COMP3610/6361
Principles of Programming Languages

Peter Höfner

Oct 25, 2023

Section 0

Admin

Lecturer

- **A/Prof. Peter Höfner**
  CSIT, Room N234 (Building 108)
  Peter.Hoefner@anu.edu.au
  +61 2 6125 0159

  Consultation
  Thursday 12pm – 1pm, or by appointment

CoLecturer and Tutors

- **Dr Fabian Muelboeck**
  Fabian.Muehlboeck@anu.edu.au

- **Abhaas Goyal**
  Abhaas.Goyal@anu.edu.au

- **Weiyou Wang**
  Weiyou.Wang@anu.edu.au

Lectures

- Wednesday, 3 pm – 5 pm
  Thursday, 11 am – 12 pm
- Rm 5.02 Marie Reay, Bldg 155
- Q/A session in Week 12

- **Etiquette**
  - engage
  - feel free to ask questions
  - we reject behaviour that strays into harassment, no matter how mild

Tutorials

- join one of the 2 tutorials
- Thursday, 3pm – 5pm (Rm 5.02 Marie Reay)
  Friday, 1pm – 2pm (Rm 4.03 Marie Reay)
- from Week 2 onwards

- **Summary**
  - your chance to discuss problems
  - discuss home work
  - discuss additional exercises

Plan/Schedule I

Resources
web: `https://cs.anu.edu.au/courses/comp3610/`
wattle: `https://wattlecourses.anu.edu.au/course/view.php?id=41142`
edstem: `https://edstem.org/`
(you will be registered at the end of the week)

Workload
The average student workload is 130 hours for a six unit course.
That is roughly **11 hours/week**.
`https://policies.anu.edu.au/ppl/document/ANUP_000691`

Plan/Schedule II

Assessment criteria

- Quizz: 0% (for feedback only)
- Assignments: 35%, 4 assignments (35marks)
- Oral exam: 65% (65 marks) **[hurdle]**
- **hurdle:** minimum of 40% in the final exam

Assessments (tentative)

No	Hand Out	Hand In	Marks
0	31/07	03/08	0
1	02/08	10/08	5
2	16/08	31/08	10
3	20/09	12/10	10
4	18/10	02/11	10

About the Course I

This course is an introduction to the theory and design of programming languages.

About the Course II

Topics (tentative)

The following schedule is tentative and likely to change.

	Topic
0	Admin
1	introduction
2	IMP and its Operational Semantics
3	Types
4	Derivation and Proofs
5	Functions, Call-by-Value, Call-by-Name
6	Typing for Call-By-Value
7	Data Types and Subtyping
8	Denotational Semantics
9	Axiomatic Semantics
10	Concurrency
11	Formal Verification

About the Course IV

Disclaimer

This is has been redesigned fairly recently.
The material in these notes has been drawn from several different sources, including the books and similar courses at some other universities. Any errors are of course all the author's own work.
As it is a newly designed course, changes in timetabling are quite likely.
Feedback (oral, email, survey, . . .) is highly appreciated.

Academic Integrity

- never misrepresent the work of others as your own
- if you take ideas from elsewhere
  you must say so with utmost clarity

Reading Material

- Glynn Winskel. *The Formal Semantics of Programming Languages – An Introduction*. MIT Press, 1993. ISBN 978-0-262-73103-4
- Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016. ISBN 978-1-107-15030-0
- Shriram Krishnamurthi. *Programming Languages: Application and Interpretation (2nd edition)* Open Textbook Library, 2017
- additional reading material can be found online

Section 1

Introduction

Foundational Knowledge of Disciplines

Mechanical Engineering

Students learn about *torque*

$$\frac{\mathrm{d}(r \times \omega)}{\mathrm{d}t} = r \times \frac{\mathrm{d}\omega}{\mathrm{d}t} + \frac{\mathrm{d}r}{\mathrm{d}t} \times \omega$$

Figure: Sydney Harbour Bridge under construction [NMA]

Foundational Knowledge of Disciplines

Electrical Engineering / Astro Physics

Students learn about *complex impedance*

$$e^{j\omega t} = \cos(\omega t) + j\sin(\omega t)$$

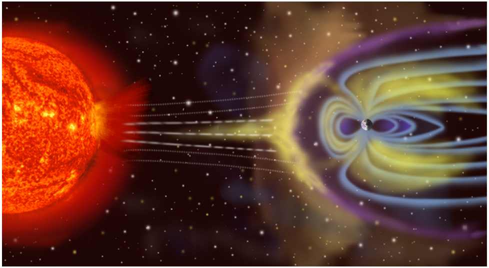


Figure: Geomagnetic Storm alters Earth's Magnetic field [Wikipedia]

Foundational Knowledge of Disciplines

Civil Engineering / Surveying

Students learn about *trigonometry*

$$\sin(\theta + \phi) = \sin\theta\cos\phi + \cos\theta\sin\phi$$

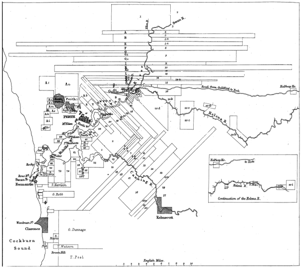


Figure: Surveying Swan River, WA [Wikipedia]

Foundational Knowledge of Disciplines

Software Engineering / Computer Science

Students learn about *???*

Figure: First Ariane 5 Flight, 1996 [ESA]

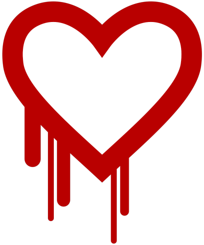


Figure: Heartbleed, 2014 [Wikipedia]

Programming Languages

Programming Languages: basic tools of computing

- what are programming languages?
- do they provide basic laws of software engineering?
- do they allow formal reasoning in the sense of above laws?

Constituents

- the *syntax* of programs:
  the alphabet of symbols and a description of the well-formed expressions, phrases, programs, etc.
- the *semantics*:
  the meaning of programs, or how they behave
- often also the *pragmatics*:
  description and examples of how the various features of the language are intended to be used

Use of Semantics

- understand a particular language
  what you can depend on as a programmer;
  what you must provide as a compiler writer
- as a tool for language design:
  - clear language design
  - express design choices, understand language features and interaction
  - for proving properties of a language, eg type safety, decidability of type inference.
- prove properties of particular programs

Style of Description (Syntax and Semantics)

- natural language
- definition ‘by’ compiler behaviour
- **mathematically**

Introductory Examples: C

In C, if initially x has value 3, what is the value of the following?

```
x++ + x++ + x++ + x++
```

Is it different to the following?

```
x++ + x++ + ++x + ++x
```

Introductory Examples: C$^\sharp$

In C$^\sharp$, what is the output of the following?

```
delegate int IntThunk();
class C {
  public static void Main() {
    IntThunk[] funcs = new IntThunk[11];
    for (int i = 0; i <= 10; i++)
    {
        funcs[i] = delegate() { return i; } ;
    }
    foreach (IntThunk f in funcs)
    {
        System.Console.WriteLine(f());
    }
  }
}
```

Introductory Examples: JavaScript

```
function bar(x) {
    return function () {
        var x = x;
        return x;
    };
}

var f = bar(200);

f()
```

About This Course

- background: mathematical description of syntax by means of formal grammars, e.g. BNF (see COMP1600)
  clear, concise and precise
- aim I: mathematical definitions of semantics/behaviour
- aim II: understand principles of program design
  (for a toy language)
- aim III: reasoning about programs

Use of formal, mathematical semantics

Implementation issues
Machine-independent specification of behaviour. Correctness of program analyses and optimisations.

Language design
Can bring to light ambiguities and unforeseen subtleties in programming language constructs. Mathematical tools used for semantics can suggest useful new programming styles. (E.g. influence of Church's lambda calculus (circa 1934) on functional programming).

Verification
Basis of methods for reasoning about program properties and program specifications.

Styles of semantics

Operational
Meanings for program phrases defined in terms of the steps of computation they can take during program execution.

Denotational
Meanings for program phrases defined abstractly as elements of some suitable mathematical structure.

Axiomatic
Meanings for program phrases defined indirectly via the axioms and rules of some logic of program properties.

Section 2

IMP
and its Operational Semantics

‘Toy’ languages

- real programming languages are large
  many features, redundant constructs
- focus on particular aspects and abstract from others (scale up later)
- even small languages can involve delicate design choices.

Design choices, from Micro to Macro

- basic values
- evaluation order
- what is guaranteed at compile-time and run-time
- how effects are controlled
- how concurrency is supported
- how information hiding is enforceable
- how large-scale development and re-use are supported
- . . .

IMP[1]– Introductory Example

IMP is an imperative language with store locations, conditionals and while loop.
For example

$$
\begin{array}{l}
l_2 := 0 \; ; \\
\textbf{while} \; !l_1 \geq 1 \; \textbf{do} \; (\\
\quad l_2 := !l_2 + !l_1 \; ; \\
\quad l_1 := !l_1 + -1 \\
)
\end{array}
$$

with initial store $\{l_1 \mapsto 3, l_2 \mapsto 0\}$.

[1] Basically the same as in Winskel 1993 (IMP) and in Hennessy 1990 (WhileL)

IMP – Syntax

Booleans $b \in \mathbb{B} = \{\texttt{true}, \texttt{false}\}$

Integers (Values) $n \in \mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$

Locations $l \in \mathbb{L} = \{l, l_0, l_1, l_2, \dots\}$

Operations $op ::= \ + \mid \geq$

Expressions

$$
\begin{aligned}
E ::= {} & n \mid b \mid E \ op \ E \mid \\
& l := E \mid\ !l \mid \\
& \textbf{skip} \mid E \ ; E \mid \\
& \textbf{if } E \textbf{ then } E \textbf{ else } E \\
& \textbf{while } E \textbf{ do } E
\end{aligned}
$$

Transition systems

A *transition system* consists of

- a set Config of configurations (or states), and
- a binary relation $\longrightarrow \subseteq$ Config $\times$ Config.

The relation $\longrightarrow$ is called the transition or reduction relation:
$c \longrightarrow c'$ reads as 'state c can make a transition to state c''.
(see DFA/NFA)

IMP Semantics (1 of 4) – Configurations

Stores are (finite) partial functions $\mathbb{L} \rightharpoonup \mathbb{Z}$.
For example, $\{l_1 \mapsto 3, l_3 \mapsto 42\}$

Configurations are pairs $\langle E\,,\ s\rangle$ of an expression E and a store s.
For example, $\langle l := 2 + !l\,,\ \{l \mapsto 3\}\rangle$.

Transitions have the form $\langle E\,,\ s\rangle \longrightarrow \langle E'\,,\ s'\rangle$.
For example, $\langle l := 2 + !l\,,\ \{l \mapsto 3\}\rangle \longrightarrow \langle l := 2 + 3\,,\ \{l \mapsto 3\}\rangle$

Transitions – Examples

Transitions are single computation steps.
For example

$$
\begin{aligned}
& \langle l := 2 + !l \,,\, \{l \mapsto 3\}\rangle \\
\longrightarrow\ & \langle l := 2 + 3 \,,\, \{l \mapsto 3\}\rangle \\
\longrightarrow\ & \langle l := 5 \,,\, \{l \mapsto 3\}\rangle \\
\longrightarrow\ & \langle \textbf{skip} \,,\, \{l \mapsto 5\}\rangle \\
\not\longrightarrow\ &
\end{aligned}
$$

Keep going until reaching a value v, an expression in $\mathbb{V} = \mathbb{B} \cup \mathbb{Z} \cup \{\textbf{skip}\}$.
A configuration $\langle E \,,\, s\rangle$ is stuck if E is not a value and $\langle E \,,\, s\rangle \not\longrightarrow$.

IMP Semantics (2 of 4) – Rules (basic operations)

(op+) $\langle n_1 + n_2\,,\, s\rangle \longrightarrow \langle n\,,\, s\rangle$ if $n = n_1 + n_2$

(op≥) $\langle n_1 \geq n_2\,,\, s\rangle \longrightarrow \langle b\,,\, s\rangle$ if $b = (n_1 \geq n_2)$

(op1) $$\frac{\langle E_1\,,\, s\rangle \longrightarrow \langle E_1'\,,\, s'\rangle}{\langle E_1\ op\ E_2\,,\, s\rangle \longrightarrow \langle E_1'\ op\ E_2\,,\, s'\rangle}$$

(op2) $$\frac{\langle E_2\,,\, s\rangle \longrightarrow \langle E_2'\,,\, s'\rangle}{\langle v\ op\ E_2\,,\, s\rangle \longrightarrow \langle v\ op\ E_2'\,,\, s'\rangle}$$

Rules (basic operations) – Examples

Find the possible sequences of transitions for

$$\langle (2+3)+(4+5)\,,\,\emptyset \rangle$$

The answer is 14 – but how do we show this formally?

IMP Semantics (3 of 4) – Store and Sequencing

(deref) $\langle !l\,,\ s\rangle \longrightarrow \langle n\,,\ s\rangle$ if $l \in \text{dom}(s)$ and $s(l) = n$

(assign1) $\langle l := n\,,\ s\rangle \longrightarrow \langle \textbf{skip}\,,\ s + \{l \mapsto n\}\rangle$ if $l \in \text{dom}(s)$

(assign2) $$\frac{\langle E\,,\ s\rangle \longrightarrow \langle E'\,,\ s'\rangle}{\langle l := E\,,\ s\rangle \longrightarrow \langle l := E'\,,\ s'\rangle}$$

(seq1) $\langle \textbf{skip}\ ;\ E_2\,,\ s\rangle \longrightarrow \langle E_2\,,\ s\rangle$

(seq2) $$\frac{\langle E_1\,,\ s\rangle \longrightarrow \langle E_1'\,,\ s'\rangle}{\langle E_1\ ;\ E_2\,,\ s\rangle \longrightarrow \langle E_1'\ ;\ E_2\,,\ s'\rangle}$$

Store and Sequencing – Examples

$$\begin{aligned}\langle l := 3\,;\,!l\,,\ \{l \mapsto 0\}\rangle &\longrightarrow \langle \textbf{skip}\,;\,!l\,,\ \{l \mapsto 3\}\rangle\\ &\longrightarrow \langle !l\,,\ \{l \mapsto 3\}\rangle\\ &\longrightarrow \langle 3\,,\ \{l \mapsto 3\}\rangle\end{aligned}$$

Australian National University

Store and Sequencing – Examples

$$\langle l := 3\; ;\; l := !l\,,\ \{l \mapsto 0\}\rangle \longrightarrow \quad ?$$

$$\langle 42 + !l\,,\ \emptyset\rangle \longrightarrow \quad ?$$

Australian National University

IMP Semantics (4 of 4) – Conditionals and While

(if1) $$\langle \textbf{if } \texttt{true} \textbf{ then } E_2 \textbf{ else } E_3 \, , \, s \rangle \longrightarrow \langle E_2 \, , \, s \rangle$$

(if2) $$\langle \textbf{if } \texttt{false} \textbf{ then } E_2 \textbf{ else } E_3 \, , \, s \rangle \longrightarrow \langle E_3 \, , \, s \rangle$$

(if3) $$\frac{\langle E_1 \, , \, s \rangle \longrightarrow \langle E_1' \, , \, s' \rangle}{\langle \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 \, , \, s \rangle \longrightarrow \langle \textbf{if } E_1' \textbf{ then } E_2 \textbf{ else } E_3 \, , \, s' \rangle}$$

(while)

$$\langle \textbf{while } E_1 \textbf{ do } E_2 \, , \, s \rangle \longrightarrow \langle \textbf{if } E_1 \textbf{ then } (E_2 \, ; \textbf{while } E_1 \textbf{ do } E_2) \textbf{ else skip} \, , \, s \rangle$$

IMP – Examples

If

$$E = \big(l_2 := 0 \;;\; \textbf{while}\ !l_1 \geq 1\ \textbf{do}\ (l_2 := !l_2 + !l_1 \;;\; l_1 := !l_1 + -1)\big)$$
$$s = \{l_1 \mapsto 3, l_2 \mapsto 0\}$$

then

$$\langle E\,,\, s\rangle \longrightarrow^* \quad ?$$

Determinacy

Theorem (Determinacy)

If $\langle E\,,\,s\rangle \longrightarrow \langle E_1\,,\,s_1\rangle$ *and* $\langle E\,,\,s\rangle \longrightarrow \langle E_2\,,\,s_2\rangle$
then $\langle E_1\,,\,s_1\rangle = \langle E_2\,,\,s_2\rangle$*.*

Proof.

later $\square$

Reminder

- basic and simple imperative while-language
- with *formal* semantics
- given in the format structural operational semantics
- rules usually have the form $\dfrac{A \qquad B}{C}$

  (special rule is $\overline{C}$, which we often write as C)
- derivation tree

$$
(\textsc{r}1)\ \dfrac{(\textsc{r}3)\ \dfrac{}{A} \qquad \dfrac{(\textsc{r}4)\ \dfrac{}{B_1} \quad (\textsc{r}5)\ \dfrac{}{B_2}}{B}\ (\textsc{r}2)}{C}
$$

Language design I

Order of Evaluation

IMP uses left-to-right evaluation. For example

$$\langle (l := 1 \; ; 0) + (l := 2 \; ; 0) \, , \{l \mapsto 0\} \rangle \longrightarrow^5 \langle 0 \, , \{l \mapsto \mathbf{2}\} \rangle$$

For right-to-left we could use

(op1') $$\frac{\langle E_2 \, , \, s \rangle \longrightarrow \langle E_2' \, , \, s' \rangle}{\langle E_1 \; op \; E_2 \, , \, s \rangle \longrightarrow \langle E_1 \; op \; E_2' \, , \, s' \rangle}$$

(op2') $$\frac{\langle E_1 \, , \, s \rangle \longrightarrow \langle E_1' \, , \, s' \rangle}{\langle E_1 \; op \; v \, , \, s \rangle \longrightarrow \langle E_1' \; op \; v \, , \, s' \rangle}$$

In this language

$$\langle (l := 1 \; ; 0) + (l := 2 \; ; 0) \, , \{l \mapsto 0\} \rangle \longrightarrow^5 \langle 0 \, , \{l \mapsto \mathbf{1}\} \rangle$$

Language design II

Assignment results

Recall

(assign1) $\langle l := n \,,\, s\rangle \longrightarrow \langle \textbf{skip} \,,\, s + \{l \mapsto n\}\rangle$ if $l \in \mathrm{dom}(s)$

(seq1) $\langle \textbf{skip} \;;\, E_2 \,,\, s\rangle \longrightarrow \langle E_2 \,,\, s\rangle$

We have chosen to map an assignment to **skip**, and $e_1 \;;\, e_2$ to progress iff $e_1 = \textbf{skip}$.

Instead we could have chosen the following.

(assign1’) $\langle l := n \,,\, s\rangle \longrightarrow \langle n \,,\, s + \{l \mapsto n\}\rangle$ if $l \in \mathrm{dom}(s)$

(seq1’) $\langle v \;;\, E_2 \,,\, s\rangle \longrightarrow \langle E_2 \,,\, s\rangle$

Language design III

Store initialisation

Recall

(deref) $\langle !l\,,\ s\rangle \longrightarrow \langle n\,,\ s\rangle$ if $l \in \mathrm{dom}(s)$ and $s(l) = n$

Assumes $l \in \mathrm{dom}(s)$.

Instead we could have

- initialise *all* locations to 0, or
- allow assignments to an $l \notin \mathrm{dom}(s)$.

Language design IV

Storable values

- our language only allows integer values (store: $\mathbb{L} \rightharpoonup \mathbb{Z}$)
- could we store any value? Could we store locations, or even programs?
- store is global and cannot create new locations

Australian National University

Language design V

Operators and Basic values

- Booleans are different from integers (unlike in C)
- Implementation is (probably) different to semantics
  Exercise: fix the semantics to match 32-bit integers

Expressiveness

Is our language expressive enough to write ‘interesting’ programs?

- **yes**: it is Turing-powerful
  Exercise: try to encode an arbitrary Turing machine in IMP
- **no**: no support for standard feature, such as functions, lists, trees, objects, modules, ...

Is the language too expressive?

- **yes**: we would like to exclude programs such as $3 + \texttt{true}$
  clearly 3 and `true` are of different type

Section 3

Types

Type systems

- describe when programs make sense
- prevent certain kinds of errors
- structure programs
- guide language design

Ideally, **well-typed programs do not get stuck**.

Run-time errors

Trapped errors

Cause execution to halt immediately.
Examples: jumping to an illegal address, raising a top-level exception.
Innocuous?

Untrapped errors

May go unnoticed for a while and later cause arbitrary behaviour.
Examples: accessing data past the end of an array, security loopholes in Java abstract machines.
Insidious!

Given a precise definition of what constitutes an untrapped run-time error, then a language is safe if all its syntactically legal programs cannot cause such errors. Usually, safety is desirable. Moreover, we'd like as few trapped errors as possible.

Formal type systems

We define a ternary relation $\Gamma \vdash E : T$

expression E has type T, under assumptions Γ on the types of locations that may occur in E.

For example (according to the definition coming up):

- $\{\} \ \vdash$ **if** $\texttt{true}$ **then** 2 **else** $3 + 4 \ :$ int
- l_1 : intref $\ \vdash$ **if** $!l_1 \geq 3$ **then** $!l_1$ **else** $3 \ :$ int
- $\{\} \ \nvdash \ 3 + \texttt{true} \ : \ T$ for any type T
- $\{\} \ \nvdash$ **if** $\texttt{true}$ **then** 3 **else** $\texttt{true} \ :$ int

Types of IMP

Types of expressions

$$T \ ::= \text{int} \mid \text{bool} \mid \text{unit}$$

Types of locations

$$T_{loc} \ ::= \text{intref}$$

We write T and T_{loc} for the sets of all terms of these grammars.

- Γ ranges over TypeEnv, the finite partial function from $\mathbb{L} \rightharpoonup \mathbb{Z}$
- notation: write $l_1 : \text{intref}, \ldots, l_k : \text{intref}$ instead of $\{l_1 \mapsto \text{intref}, \ldots, l_k \mapsto \text{intref}\}$

Type Judgement (1 of 3)

(int) $\Gamma \vdash n : \text{int} \qquad \text{if } n \in \mathbb{Z}$

(bool) $\Gamma \vdash b : \text{bool} \qquad \text{if } b \in \mathbb{B} = \{\texttt{true}, \texttt{false}\}$

(op+) $$\frac{\Gamma \vdash E_1 : \text{int} \qquad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 + E_2 : \text{int}}$$

(op$\geq$) $$\frac{\Gamma \vdash E_1 : \text{int} \qquad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 \geq E_2 : \text{bool}}$$

(if) $$\frac{\Gamma \vdash E_1 : \text{bool} \qquad \Gamma \vdash E_2 : T \qquad \Gamma \vdash E_3 : T}{\Gamma \vdash \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 : T}$$

Type Judgement – Example

Prove that $\{\} \vdash \textbf{if}\ \texttt{false}\ \textbf{then}\ 2\ \textbf{else}\ 3+4 : \mathsf{int}$.

$$
\dfrac{
\dfrac{}{\{\} \vdash \texttt{false} : \mathsf{bool}}\,(\mathrm{BOOL})
\qquad
\dfrac{}{\{\} \vdash 2 : \mathsf{int}}\,(\mathrm{INT})
\qquad
\dfrac{
(\mathrm{INT})\,\dfrac{}{\{\} \vdash 3 : \mathsf{int}}
\quad
\dfrac{}{\{\} \vdash 4 : \mathsf{int}}\,(\mathrm{INT})
}{\{\} \vdash 3+4 : \mathsf{int}}\,(\mathrm{OP}+)
}{\{\} \vdash \textbf{if}\ \texttt{false}\ \textbf{then}\ 2\ \textbf{else}\ 3+4 : \mathsf{int}}\,(\mathrm{IF})
$$

Type Judgement (2 of 3)

(assign) $$\frac{\Gamma(l) = \mathsf{intref} \qquad \Gamma \vdash E : \mathsf{int}}{\Gamma \vdash l := E : \mathsf{unit}}$$

(deref) $$\frac{\Gamma(l) = \mathsf{intref}}{\Gamma \vdash\, !l : \mathsf{int}}$$

Here, (for the moment) $\Gamma(l) = \mathsf{intref}$ means $l \in \mathsf{dom}(\Gamma)$

Type Judgement (3 of 3)

(skip) $\Gamma \vdash \textbf{skip} : \text{unit}$

(seq) $$\frac{\Gamma \vdash E_1 : \text{unit} \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 \; ; \; E_2 : T}$$

(while) $$\frac{\Gamma \vdash E_1 : \text{bool} \qquad \Gamma \vdash E_2 : \text{unit}}{\Gamma \vdash \textbf{while } E_1 \textbf{ do } E_2 : \text{unit}}$$

Type Judgement – Properties

Theorem (Progress)

If $\Gamma \vdash E : T$ and dom$(\Gamma) \subseteq$ dom(s) then either E is a value or there exist E' and s' such that $\langle E\, ,\, s\rangle \longrightarrow \langle E'\, ,\, s'\rangle$.

Theorem (Type Preservation)

If $\Gamma \vdash E : T$, dom$(\Gamma) \subseteq$ dom(s) and $\langle E\, ,\, s\rangle \longrightarrow \langle E'\, ,\, s'\rangle$ then $\Gamma \vdash E' : T$ and dom$(\Gamma) \subseteq$ dom(s').

Type Safety

Main result: Well-typed programs do not get stuck.

Theorem (Type Safety)

If $\Gamma \vdash E : T$, *dom*$(\Gamma) \subseteq$ *dom*(s), *and* $\langle E\ , \ s\rangle \longrightarrow^* \langle E'\ , \ s'\rangle$ *then either* E' *is a value with* $\Gamma \vdash E' : T$, *or there exist* E'', s'' *such that* $\langle E'\ , \ s'\rangle \longrightarrow \langle E''\ , \ s''\rangle$, $\Gamma \vdash E'' : T$ *and dom*$(\Gamma) \subseteq$ *dom*(s'').

Here, $\longrightarrow^*$ means arbitrary many steps in the transition system.

Type checking, typeability, and type inference

Type checking problem for a type system:
given Γ, E and T, is $\Gamma \vdash E : T$ derivable?

Type inference problem:
given Γ and E, find a type T such that $\Gamma \vdash E : T$ is derivable, or show there is none.

Type inference is usually harder than type checking, for a type T needs to be computed.

For our type system, though, both are easy.

Properties

Theorem (Type inference)

Given Γ and E , one can find T such that $\Gamma \vdash E : T$, or show that there is none.

Theorem (Decidability of type checking)

Given Γ, E and T, one can decide whether $\Gamma \vdash E : T$ holds.

Moreover

Theorem (Uniqueness of typing)

If $\Gamma \vdash E : T$ and $\Gamma \vdash E : T'$ then $T = T'$.

Section 4

Proofs (Structural Induction)

Why Proofs

- how do we know that the stated theorems are actually true? intuition is often wrong – we need proof
- proofs strengthen intuition about language features
- examines all the various cases
- can guarantee items such as type safety
- most of our definitions are **inductive**; we use *structural* induction

(Mathematical) Induction

Mathematical induction proves that we can climb as high as we like on a ladder, by proving that we can climb onto the bottom rung (the basis) and that from each rung we can climb up to the next one (the step).

[Concrete Mathematics (1994), R. Graham]

Natural Induction I

A proof by (natural) induction consists of **two cases**.

The **base case** proves the statement for $n = 0$ without assuming any knowledge of other cases.
The **induction step** proves that if the statement holds for any given case $n = k$, then it must also hold for the next case $n = k + 1$.

Natural Induction II

Theorem

$\forall n \in \mathbb{N} . \Phi(n)$.

Proof.

Base case: show $\Phi(0)$
Induction step: $\forall k.\ \Phi(k) \implies \Phi(k+1)$
For that we fix an arbitrary k.
Assume $\Phi(k)$ derive $\Phi(k+1)$. $\square$

Example: $0 + 1 + 2 + \cdots + n = \frac{n \cdot (n+1)}{2}$.

Australian National University

Natural Induction III

Theorem

$\forall n \in \mathbb{N} . \Phi(n)$.

Proof.

Base case: show $\Phi(0)$
Induction step: $\forall i, k . 0 \leq i \leq k . \ \Phi(i) \Longrightarrow \Phi(k+1)$
For that we fix an arbitrary k.
Assume $\phi(i)$ *for all* $i \leq k$ derive $\phi(k+1)$. $\square$

Example: $F_n = \frac{\varphi^n - \psi^n}{\varphi - \psi}$,
with F_n is the n-th Fibonacci number, $\varphi = \frac{1+\sqrt{5}}{2}$ (the golden ratio) and $\psi = \frac{1-\sqrt{5}}{2}$.

Structural Induction I

- generalisation of natural induction
- prove that some proposition $\Phi(x)$ holds for all x of some sort of recursively defined structure
- requires well-founded partial order

Examples: lists, formulas, trees

Structural Induction II

Determinacy	structural induction for E
Progress	rule induction for $\Gamma \vdash E : T$
	(induction over the height of derivation tree)
Type Preservation	rule induction for $\langle E\,,\,s\rangle \longrightarrow \langle E'\,,\,s'\rangle$
Safety	mathematical induction on $\longrightarrow^n$
Uniqueness of typing	$\ldots$
Decidability of typability	exhibiting an algorithm
Decidability of type checking	corollary of other results

Structural Induction over Expressions

Prove facts about all expressions, e.g. Determinacy?

Theorem (Determinacy)

If $\langle E \,,\, s \rangle \longrightarrow \langle E_1 \,,\, s_1 \rangle$ *and* $\langle E \,,\, s \rangle \longrightarrow \langle E_2 \,,\, s_2 \rangle$
then $\langle E_1 \,,\, s_1 \rangle = \langle E_2 \,,\, s_2 \rangle$.

Do not forget the elided universal quantifiers.

Theorem (Determinacy)

For all E***,*** s***,*** E_1***,*** s_1***,*** E_2 ***and*** s_2***,***
if $\langle E \,,\, s \rangle \longrightarrow \langle E_1 \,,\, s_1 \rangle$ *and* $\langle E \,,\, s \rangle \longrightarrow \langle E_2 \,,\, s_2 \rangle$
then $\langle E_1 \,,\, s_1 \rangle = \langle E_2 \,,\, s_2 \rangle$.

Abstract Syntax

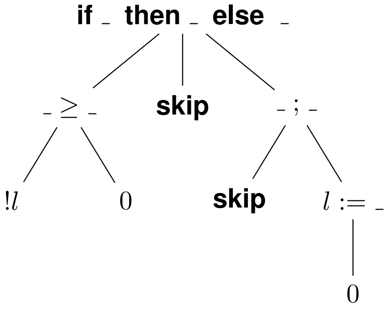
Remember the definition of expressions:

$$
\begin{aligned}
E ::= {} & n \mid b \mid E \; op \; E \mid \\
& l := E \mid !l \mid \\
& \textbf{if } E \textbf{ then } E \textbf{ else } E \mid \\
& \textbf{skip} \mid E \, ; E \mid \\
& \textbf{while } E \textbf{ do } E
\end{aligned}
$$

This defines an (infinite) set of expressions.

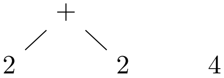
Abstract Syntax Tree I

Example: **if** $!l \geq 0$ **then skip else** (**skip** ; $l := 0$)

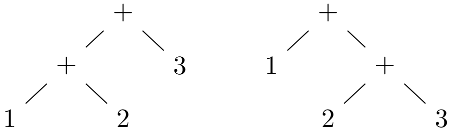


Abstract Syntax Tree II

- equivalent expressions are not the same, e.g., $2 + 2 \neq 4$



- ambiguity, e.g., $(1 + 2) + 3 \neq 1 + (2 + 3)$



Parentheses are only used for disambiguation – they are not part of the grammar

Structural Induction (for abstract syntax)

Theorem
$\forall E \in$ *IMP*. $\Phi(E)$

Proof.
Base case(s): show $\Phi(E)$ for each unary tree constructor (leaf)
Induction step(s): show it for the remaining constructors

$$\forall c.\ \forall E_1, \ldots E_k.\ (\Phi(E_1) \wedge \cdots \wedge \Phi(E_k)) \Longrightarrow \Phi(c(E_1, \ldots, E_k))$$

$\square$

Structural Induction (syntax IMP)

To show $\forall E \in \text{IMP}.\ \Phi(E)$.

base cases

nullary: $\Phi(\textbf{skip})$

$\forall b \in \mathbb{B}.\ \Phi(b)$

$\forall n \in \mathbb{Z}.\ \Phi(n)$

$\forall l \in \mathbb{L}.\ \Phi(!l)$

steps

unary: $\forall l \in \mathbb{L}.\ \forall E.\ \Phi(E) \implies \Phi(l := E)$

binary: $\forall op.\ \forall E_1, E_2.\ (\Phi(E_1) \wedge \Phi(E_2)) \implies \Phi(E_1\ op\ E_2)$

$\forall E_1, E_2.\ (\Phi(E_1) \wedge \Phi(E_2)) \implies \Phi(E_1\ ;\ E_2)$

$\forall E_1, E_2.\ (\Phi(E_1) \wedge \Phi(E_2)) \implies \Phi(\textbf{while}\ E_1\ \textbf{do}\ E_2)$

ternary: $\forall E_1, E_2, E_3.\ (\Phi(E_1) \wedge \Phi(E_2) \wedge \Phi(E_3))$

$\implies \Phi(\textbf{if}\ E_1\ \textbf{then}\ E_2\ \textbf{else}\ E_3)$

Proving Determinacy – Outline

Theorem (Determinacy)

For all** E**,** s**,** E_1**,** s_1**,** E_2 **and** s_2**,
if $\langle E\ ,\ s\rangle \longrightarrow \langle E_1\ ,\ s_1\rangle$ *and* $\langle E\ ,\ s\rangle \longrightarrow \langle E_2\ ,\ s_2\rangle$
then $\langle E_1\ ,\ s_1\rangle = \langle E_2\ ,\ s_2\rangle$*.*

Proof.

Choose

$$\begin{aligned}\Phi(E) \overset{\text{def}}{=} \forall s, E', s', E'', s''.& \\ &(\langle E\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle \ \wedge\ \langle E\ ,\ s\rangle \longrightarrow \langle E''\ ,\ s''\rangle) \\ &\quad\Longrightarrow \langle E'\ ,\ s'\rangle = \langle E''\ ,\ s''\rangle\end{aligned}$$

and show $\Phi(E)$ by structural induction over E. $\square$

Proving Determinacy – Sketch

Some cases on whiteboard

Proving Determinacy – auxiliary lemma

Values do not reduce.

Lemma

For all $E \in$ IMP, if E is a value then
$\forall s.\ \neg(\exists E', s'.\ \langle E\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle)$*.*

Proof.

- E is a value iff it is of the form n, b, **skip**
- By examination of the rules . . .
  there is no rule with conclusion of the form $\langle E\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle$ for E a value

$\square$

Inversion I

In proofs involving inductive definitions. one often needs an *inversion property*.
Given a tuple in one inductively defined relation, gives you a case analysis of the possible "last rule" used.

Lemma (Inversion for $\longrightarrow$)

If $\langle E\,,\,s\rangle \longrightarrow \langle \hat{E}\,,\,\hat{s}\rangle$ *then either*

1. *(op+): there exists* n_1, n_2 *and* n *such that* $E = n_1\ op\ n_2$, $\hat{E} = n$, $\hat{s} = s$ *and* $n = n_1 + n_2$,
   (Note: $+$*s have different meanings in this statements), or*
2. *(op1): there exists* E_1, E_2, op *and* E_1' *such that* $E = E_1\ op\ E_2$, $\hat{E} = E_1'\ op\ E_2$ *and* $\langle E_1\,,\,s\rangle \longrightarrow \langle E_1'\,,\,s'\rangle$*, or*
3. *…*

Inversion II

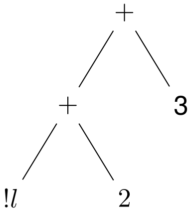
Lemma (Inversion for $\vdash$)

If $\Gamma \vdash E : T$ *then either*

- $\ldots$

Determinacy – Intuition

The intuition behind structural induction over expressions. Consider $(!l + 2) + 3$. How can we see that $\Phi((!l + 2) + 3)$ holds?



Rule Induction

How to prove the following theorems?

Theorem (Progress)

If $\Gamma \vdash E : T$ and $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$ then either E is a value or there exist E' and s' such that $\langle E \,,\, s \rangle \longrightarrow \langle E' \,,\, s' \rangle$.

Theorem (Type Preservation)

If $\Gamma \vdash E : T$*, dom*$(\Gamma) \subseteq$ *dom*(s) *and* $\langle E \,,\, s \rangle \longrightarrow \langle E' \,,\, s' \rangle$ *then* $\Gamma \vdash E' : T$ *and dom*$(\Gamma) \subseteq$ *dom*(s')*.*

Inductive Definition of $\longrightarrow$

What does $\langle E\,,\,s\rangle \longrightarrow \langle E'\,,\,s'\rangle$ actually mean?

We defined the transition relation by providing some rules, such as

(op+) $\quad \langle n_1 + n_2\,,\,s\rangle \longrightarrow \langle n\,,\,s\rangle \qquad$ if $n = n_1 + n_2$

(op1) $$\frac{\langle E_1\,,\,s\rangle \longrightarrow \langle E_1'\,,\,s'\rangle}{\langle E_1\ op\ E_2\,,\,s\rangle \longrightarrow \langle E_1'\ op\ E_2\,,\,s'\rangle}$$

These rules (their concrete instances) inductively/recursively define a set of derivation trees. The last step in the derivation tree defines a step in the transition system.

We define the (infinite) set of all finite derivation trees

Derivation Tree (Transition Relation) – Example

$$\dfrac{\dfrac{\dfrac{}{\langle 2+2\,,\,\{\}\rangle \longrightarrow \langle 4\,,\,\{\}\rangle}\ (\text{OP}+)}{\langle (2+2)+3\,,\,\{\}\rangle \longrightarrow \langle 4+3\,,\,\{\}\rangle}\ (\text{OP}1)}{\langle (2+2)+3 \geq 5\,,\,\{\}\rangle \longrightarrow \langle 4+3 \geq 5\,,\,\{\}\rangle}\ (\text{OP}1)$$

Australian National University

Derivation Tree (Typing Judgement) – Example

$$
\dfrac{\dfrac{\dfrac{\Gamma(l) = \mathsf{intref}}{\Gamma \vdash !l : \mathsf{int}}\;(\textsc{derref}) \quad \dfrac{}{\Gamma \vdash 2 : \mathsf{int}}\;(\textsc{int})}{\Gamma \vdash\, !l + 2 : \mathsf{int}}\;(\textsc{op+}) \qquad \dfrac{}{\Gamma \vdash 3 : \mathsf{int}}\;(\textsc{int})}{\Gamma \vdash (!l + 2) + 3 : \mathsf{int}}\;(\textsc{op+})
$$

Principle of Rule Induction I

For any property $\Phi(a)$ of elements a of A, and any set of rules which define a subset S_R of A, to prove

$$\forall a \in S_R.\ \Phi(a)$$

it is enough to prove that $\{a \mid \Phi(a)\}$ is closed under the rules, i.e., for each

$$\frac{h_1 \quad \ldots h_k}{c}$$

if $\Phi(h_1) \wedge \cdots \wedge \Phi(h_k)$ then $\Phi(c)$.

Principle of Rule Induction II

For any property $\Phi(a)$ of elements a of A, and any set of rules which define a subset S_R of A, to prove

$$\forall a \in S_R.\ \Phi(a)$$

it is enough to prove that for each

$$\frac{h_1 \quad \ldots h_k}{c}$$

if $\Phi(h_1) \wedge \cdots \wedge \Phi(h_k)$ then $\Phi(c)$.

Proving Progress I

Theorem (Progress)

If $\Gamma \vdash E : T$ and $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$ then either E is a value or there exist E' and s' such that $\langle E\,,\ s\rangle \longrightarrow \langle E'\,,\ s'\rangle$.

Proof.

Choose

$$\begin{aligned}\Phi(\Gamma, E, T) = \ & \forall s.\ \mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(s) \\ & \Longrightarrow \mathsf{value}(E) \vee (\exists E', s'.\ \langle E\,,\ s\rangle \longrightarrow \langle E'\,,\ s'\rangle)\end{aligned}$$

We show that for all Γ, E, T, if $\Gamma \vdash E : T$ then $\Phi(\Gamma, E, T)$, by rule induction on the definition of $\vdash$. $\square$

Proving Progress II

Rule induction for our typing rules means:

(int) $\forall \Gamma, n.\ \Phi(\Gamma, n, \mathsf{int})$

(deref) $\forall \Gamma, l.\ \Gamma(l) = \mathsf{intref} \Longrightarrow \Phi(\Gamma, !l, \mathsf{int})$

(op+) $\forall \Gamma, E_1, E_2.\ \big(\Phi(\Gamma, E_1, \mathsf{int}) \wedge \Phi(\Gamma, E_2, \mathsf{int}) \wedge \Gamma \vdash E_1 : \mathsf{int} \wedge \Gamma \vdash E_2 : \mathsf{int}\big)$
$\Longrightarrow \Phi(\Gamma, E_1 + E_2, \mathsf{int})$

(seq) $\forall \Gamma, E_1, E_2.\ \big(\Phi(\Gamma, E_1, \mathsf{unit}) \wedge \Phi(\Gamma, E_2, T) \wedge \Gamma \vdash E_1 : \mathsf{unit} \wedge \Gamma \vdash E_2 : T\big)$
$\Longrightarrow \Phi(\Gamma, E_1; E_2, \mathsf{int})$

… [10 rules in total]

Proving Progress III

$$\Phi(\Gamma, E, T) = \forall s.\ \mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(s) \\ \Longrightarrow \mathsf{value}(E) \vee (\exists E', s'.\ \langle E\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle)$$

Case (op$+$):

$$(\mathsf{op}+)\quad \frac{\Gamma \vdash E_1 : \mathsf{int} \qquad \Gamma \vdash E_2 : \mathsf{int}}{\Gamma \vdash E_1 + E_2 : \mathsf{int}}$$

- assume $\Phi(\Gamma, E_1, \mathsf{int})$, $\Phi(\Gamma, E_2, \mathsf{int})$, $\Gamma \vdash E_1 : \mathsf{int}$ and $\Gamma \vdash E_2 : \mathsf{int}$
- we have to show $\Phi(\Gamma, E_1 + E_2, \mathsf{int})$
- assume an arbitrary but fixed s, and $\mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(s)$
- $E_1 + E_2$ is not a value; hence we have to show

$$\exists E', s'.\ \langle E_1 + E_2\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle$$

Proving Progress IV

Case (op$+$) (cont'd)**:**

- we have to show

$$\exists E', s'.\ \langle E_1 + E_2\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle$$

- Using $\Phi(\Gamma, E_1, \text{int})$ and $\Phi(\Gamma, E_2, \text{int})$ we have

  case E_1 reduces. Then $E_1 + E_2$ does, by (op1).
  case E_1 is a value and E_2 reduces. Then $E_1 + E_2$ does, by (op2).
  case E_1 and E_2 are values; we want to use

$$\text{(op+)} \qquad \langle n_1 + n_2\ ,\ s\rangle \longrightarrow \langle n\ ,\ s\rangle \qquad \text{if } n = n_1 + n_2$$

  we assumed $\Gamma \vdash E_1 : \text{int}$ and $\Gamma \vdash E_2 : \text{int}$ we need $E_1 = n_1$ and $E_2 = n_2$; then $E_1 + E_2$ reduces, by (op$+$).

Proving Progress V

Lemma

For all Γ, E, T, if $\Gamma \vdash E : T$ is a value with $T =$ int then there exists $n \in \mathbb{Z}$ with $E = n$.

Derivation Tree (Typing Judgement) – Example

$$\dfrac{\dfrac{\dfrac{\Gamma(l) = \mathsf{intref}}{\Gamma \vdash\ !l : \mathsf{int}}\ (\textsc{deref}) \quad \dfrac{}{\Gamma \vdash 2 : \mathsf{int}}\ (\textsc{int})}{\Gamma \vdash\ !l + 2 : \mathsf{int}}\ (\textsc{op+}) \qquad \dfrac{}{\Gamma \vdash 3 : \mathsf{int}}\ (\textsc{int})}{\Gamma \vdash (!l + 2) + 3 : \mathsf{int}}\ (\textsc{op+})$$

Which Induction Principle to Use?

- matter of convenience (all equivalent)
- use an induction principle that matches the definitions

Structural Induction (Repetition)

Determinacy	structural induction for E
Progress	rule induction for $\Gamma \vdash E : T$
	(induction over the height of derivation tree)
Type Preservation	rule induction for $\langle E\,,\ s\rangle \longrightarrow \langle E'\,,\ s'\rangle$
Safety	mathematical induction on $\longrightarrow^n$
Uniqueness of typing	$\ldots$
Decidability of typability	exhibiting an algorithm
Decidability of type checking	corollary of other results

Why care about Proofs?

1. sometimes it seems hard or pointless to prove things because they are ‘obvious’, . . .
   (in particular with our language)
2. proofs illustrate (and explain) why ‘things are obvious’
3. sometimes the obvious facts are false . . .
4. sometimes the obvious facts are not obvious at all
   (in particular for ‘real’ languages)
5. sometimes a proof contains or suggests an algorithm that you need
   (proofs that type inference is decidable (for fancier type systems))
6. force a clean language design

Section 5

Functions

Functions, Methods, Procedures, . . .

- so far IMP was really minimalistic
- the most important ‘add-on’ are functions
- this requires variables and other concepts

Examples

```
add_one :: Int -> Int
add_one n = n + 1
```

```
public int add_one (int x) {
  return (x+1);
}
```

```
<script type="text/vbscript">
function addone(x)
    addone = x+1
end function
</script>
```

Introductory Examples: C$^\sharp$

In C$^\sharp$, what is the output of the following?

```
delegate int IntThunk();
class C {
  public static void Main() {
    IntThunk[] funcs = new IntThunk[11];
    for (int i = 0; i <= 10; i++)
    {
        funcs[i] = delegate() { return i; } ;
    }
    foreach (IntThunk f in funcs)
    {
        System.Console.WriteLine(f());
    }
  }
}
```

In my opinion, the design was wrong.

Functions – Examples

We want include the following expressions:

$$(\textbf{fn}\ x : \mathsf{int} \Rightarrow x + 1)$$

$$(\textbf{fn}\ x : \mathsf{int} \Rightarrow x + 1)\ 7$$

$$(\textbf{fn}\ y : \mathsf{int} \Rightarrow (\textbf{fn}\ x : \mathsf{int} \Rightarrow x + y))$$

$$(\textbf{fn}\ y : \mathsf{int} \Rightarrow (\textbf{fn}\ x : \mathsf{int} \Rightarrow x + y))\ 1$$

$$(\textbf{fn}\ x : \mathsf{int} \to \mathsf{int} \Rightarrow (\textbf{fn}\ y : \mathsf{int} \Rightarrow x\ (x\ y)))$$

$$(\textbf{fn}\ x : \mathsf{int} \to \mathsf{int} \Rightarrow (\textbf{fn}\ y : \mathsf{int} \Rightarrow x\ (x\ y)))\ (\textbf{fn}\ x : \mathsf{int} \Rightarrow x + 1)$$

$$\big((\textbf{fn}\ x : \mathsf{int} \to \mathsf{int} \Rightarrow (\textbf{fn}\ y : \mathsf{int} \Rightarrow x\ (x\ y)))\ (\textbf{fn}\ z : \mathsf{int} \Rightarrow z + 1)\big)\ 7$$

Functions – Syntax

We extend our syntax:

Variables $x \in \mathbb{X}$ for a set $\mathbb{X} = \{x, y, z, \dots\}$ (countable)

Expressions

$$E ::= \dots \mid (\textbf{fn}\ x : T \Rightarrow E) \mid E\ E \mid x$$

Types

$$T ::= \text{int} \mid \text{bool} \mid \text{unit} \mid T \to T$$
$$T_{loc} ::= \text{intref}$$

Australian National University

Variable Shadowing

$(\textbf{fn}\ x : \mathsf{int} \Rightarrow (\textbf{fn}\ x : \mathsf{int} \Rightarrow x + 1))$

Alpha conversion

In expressions ($\textbf{fn}\ x : T \Rightarrow E$), variable x is a binder

- inside E, any x (not being a binder themselves and not inside another ($\textbf{fn}\ x : T' \Rightarrow \ldots$)) mean the same
- it is the formal parameter of this function
- outside ($\textbf{fn}\ x : T \Rightarrow E$), it does not matter which variable we use – in fact, we should not be able to tell
  For example, ($\textbf{fn}\ x : \text{int} \Rightarrow x + 2$) should be the same as ($\textbf{fn}\ y : \text{int} \Rightarrow y + 2$)

Binders are known from many areas of mathematics/logics.

Alpha conversion: free and bound variables

An occurrence x in an expression E is *free* if it is not inside any $(\textbf{fn } x : T \Rightarrow \ldots)$.
For example:

$$17$$

$$x + y$$

$$(\textbf{fn } x : \text{int} \Rightarrow x + 2)$$

$$(\textbf{fn } x : \text{int} \Rightarrow x + z)$$

$$\textbf{if } y \textbf{ then } 2 + x \textbf{ else } ((\textbf{fn } x : \text{int} \Rightarrow x + 2)\ z)$$

Alpha Conversion – Binding Examples

$$(\textbf{fn } x : \textsf{int} \Rightarrow x + 2)$$

$$(\textbf{fn } x : \textsf{int} \Rightarrow x + z)$$

$$(\textbf{fn } y : \textsf{int} \Rightarrow y + z)$$

$$(\textbf{fn } z : \textsf{int} \Rightarrow z + z)$$

$$(\textbf{fn } x : \textsf{int} \Rightarrow (\textbf{fn } x : \textsf{int} \Rightarrow x + 2))$$

Alpha Conversion – Convention

- we want to allow to replace binder x (and all occurrences of x bound by that x) by another binder y
- *if* it does not change the binding graph

For example

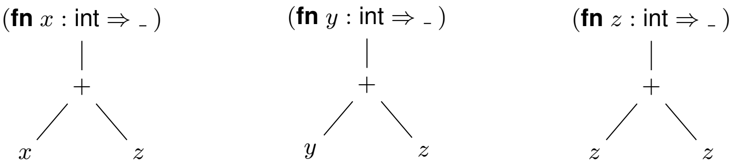
$$(\textbf{fn}\ x : \mathsf{int} \Rightarrow x + z) \ = \ (\textbf{fn}\ y : \mathsf{int} \Rightarrow y + z) \ \neq \ (\textbf{fn}\ z : \mathsf{int} \Rightarrow z + z)$$

- called 'working up to alpha conversion'
- extend abstract syntax trees by pointers

Syntax Trees up to Alpha Conversion

$$(\textbf{fn}\ x : \mathsf{int} \Rightarrow x + z) \;=\; (\textbf{fn}\ y : \mathsf{int} \Rightarrow y + z) \;\neq\; (\textbf{fn}\ z : \mathsf{int} \Rightarrow z + z)$$

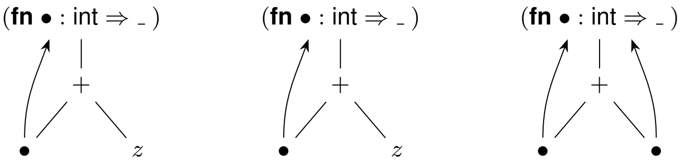
Standard abstract syntax trees



Syntax Trees up to Alpha Conversion II

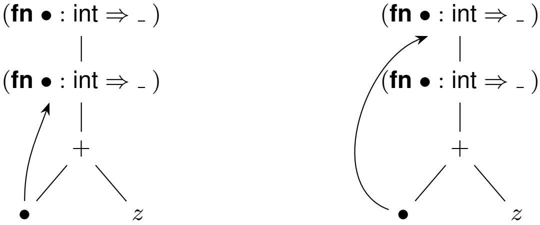
$$(\textbf{fn } x : \mathsf{int} \Rightarrow x + z) \;=\; (\textbf{fn } y : \mathsf{int} \Rightarrow y + z) \;\neq\; (\textbf{fn } z : \mathsf{int} \Rightarrow z + z)$$

Add pointers



Syntax Trees up to Alpha Conversion III

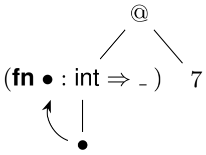
$$
\begin{aligned}
&(\textbf{fn } x : \mathsf{int} \Rightarrow (\textbf{fn } x : \mathsf{int} \Rightarrow x + 2)) \\
= {} &(\textbf{fn } y : \mathsf{int} \Rightarrow (\textbf{fn } z : \mathsf{int} \Rightarrow z + 2)) \quad \neq \quad (\textbf{fn } z : \mathsf{int} \Rightarrow (\textbf{fn } y : \mathsf{int} \Rightarrow z + 2))
\end{aligned}
$$

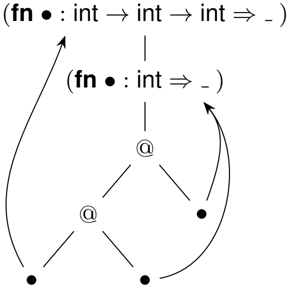


Syntax Trees up to Alpha Conversion IV

Application and function type

$(\textbf{fn}\ x : \textsf{int} \Rightarrow x)\ 7$ $\qquad$ $(\textbf{fn}\ z : \textsf{int} \to \textsf{int} \to \textsf{int} \Rightarrow (\textbf{fn}\ y : \textsf{int} \Rightarrow z\ y\ y))$

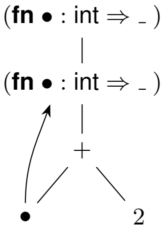


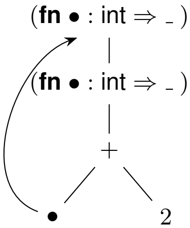


De Bruijn indices

- these pointers are known as *De Bruijn indices*
- each occurrence of a bound variable is represented by the number of **fn**-nodes you have to pass

$$(\textbf{fn}\ \bullet : \mathsf{int} \Rightarrow (\textbf{fn}\ \bullet : \mathsf{int} \Rightarrow v_0 + 2)) \quad \neq \quad (\textbf{fn}\ \bullet : \mathsf{int} \Rightarrow (\textbf{fn}\ \bullet : \mathsf{int} \Rightarrow v_1 + 2))$$





Free Variables

- *free variables* of an expression E are the set of variables for which there is an occurrence of x free in E

$$\begin{aligned}\mathsf{fv}(x) &= \{x\}\\ \mathsf{fv}(E_1\ op\ E_2) &= \mathsf{fv}(E_1) \cup \mathsf{fv}(E_2)\\ \mathsf{fv}((\textbf{fn}\ x : T \Rightarrow E)) &= \mathsf{fv}(E) - \{x\}\end{aligned}$$

- an expression E is closed if $\mathsf{fv}(E) = \emptyset$
- For a set $\mathbb{E}$ of expressions $\mathsf{fv}(\mathbb{E}) = \bigcup_{E\in\mathbb{E}} \mathsf{fv}(E)$
- these definitions are alpha-invariant
(all forthcoming definitions should be)

Substitution – Examples

- semantics of functions will involve substitution (replacement)
- $\{E/x\}\, E'$ denotes the expression E' where all *free* occurrences of x are substituted by E

Examples

$$\{3/x\}\,(x \geq x) = (3 \geq 3)$$

$$\{3/x\}\,((\textbf{fn}\ x : \text{int} \Rightarrow x + y)\ x) = (\textbf{fn}\ x : \text{int} \Rightarrow x + y)\ 3$$

$$\{y + 2/x\}\,(\textbf{fn}\ y : \text{int} \Rightarrow x + y) = (\textbf{fn}\ z : \text{int} \Rightarrow (y + 2) + z)$$

Substitution

Definition

$$\{E/z\}\, x \stackrel{\text{def}}{=} \begin{cases} E & \text{if } x = z \\ x & \text{otherwise} \end{cases}$$

$$\{E/z\}\, (\textbf{fn}\ x : T \Rightarrow E_1) \stackrel{\text{def}}{=} (\textbf{fn}\ x : T \Rightarrow (\{E/z\}\, E_1)) \quad \text{if } x \neq z \text{ and } x \notin \text{fv}(E) (^*)$$

$$\{E/z\}\, (E_1\ E_2) \stackrel{\text{def}}{=} (\{E/z\}\, E_1) \quad (\{E/z\}\, E_2)$$

$$\ldots$$

if $(^*)$ is false, apply alpha conversion to generate a variant of $(\textbf{fn}\ x : T \Rightarrow E_1)$ to make $(^*)$ true

Substitution – Example

Substitution – Example Again

$$
\begin{aligned}
&\{y+2/x\}\,(\textbf{fn}\ y : \mathsf{int} \Rightarrow x + y)\\
&= \{y+2/x\}\,(\textbf{fn}\ z : \mathsf{int} \Rightarrow x + z)\\
&= (\textbf{fn}\ z : \mathsf{int} \Rightarrow \{y+2/x\}\,(x + z))\\
&= (\textbf{fn}\ z : \mathsf{int} \Rightarrow \{y+2/x\}\,x + \{y+2/x\}\,z)\\
&= (\textbf{fn}\ z : \mathsf{int} \Rightarrow (y+2) + z)
\end{aligned}
$$

Simultaneous Substitution

- a *substitution* σ is a *finite* partial function from variables to expressions
- notation: $\{E_1/x_1, \dots, E_k/x_k\}$ instead of $\{x_1 \mapsto E_1, \dots, x_k \mapsto E_k\}$
- the formal definition is straight forward

Definition Substitution [for completeness]

Let σ be $\{E_1/x_1, \ldots, E_k/x_k\}$.
Moreover, $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_k\}$ and $\mathrm{ran}(\sigma) = \{E_1, \ldots, E_k\}$.

$$
\begin{array}{lcll}
\sigma\ x & = & \begin{cases} E_i & \text{if } x = x_i \text{ (and } x_i \in \mathrm{dom}(\sigma)) \\ x & \text{otherwise} \end{cases} \\
\sigma\ (\textbf{fn}\ x : T \Rightarrow E) & = & (\textbf{fn}\ x : T \Rightarrow (\sigma\ E)) & \text{if } x \notin \mathrm{dom}(\sigma) \text{ and } x \notin \mathrm{fv}(\mathrm{ran}(\sigma))\ (^*) \\
\sigma\ (E_1\ E_2) & = & (\sigma\ E_1)\ (\sigma\ E_2) \\
\sigma\ n & = & n \\
\sigma\ (E_1\ op\ E_2) & = & (\sigma\ E_1)\ op\ (\sigma\ E_2) \\
\sigma\ (\textbf{if}\ E_1\ \textbf{then}\ E_2\ \textbf{else}\ E_3) & = & \textbf{if}\ (\sigma\ E_1)\ \textbf{then}\ (\sigma\ E_2)\ \textbf{else}\ (\sigma\ E_3) \\
\sigma\ b & = & b \\
\sigma\ \textbf{skip} & = & \textbf{skip} \\
\sigma\ (l := E) & = & l := (\sigma\ E) \\
\sigma\ (!l) & = & !l \\
\sigma\ (E_1\ ;\ E_2) & = & (\sigma\ E_1)\ ;\ (\sigma\ E_2) \\
\sigma\ (\textbf{while}\ E_1\ \textbf{do}\ E_2) & = & \textbf{while}\ (\sigma\ E_1)\ \textbf{do}\ (\sigma\ E_2)
\end{array}
$$

Function Behaviour

- we are now ready to define the semantics of functions
- there are some choices to be made
  - call-by-value
  - call-by-name
  - call-by-need

Function Behaviour

Consider the expression

$$E = (\textbf{fn}\ x : \mathsf{unit} \Rightarrow (l := 1)\ ;\ x)\ (l := 2)$$

What is the transition relation

$$\langle E\, ,\ \{l \mapsto 0\}\rangle \longrightarrow^* \langle \textbf{skip}\, ,\ \{l \mapsto \textbf{???}\}\rangle$$

Choice 1: Call-by-Value

Idea: reduce left-hand-side of application to an **fn**-term;
then reduce argument to a value;
then replace all occurrences of the formal parameter in the **fn**-term by that value.

$$E = (\textbf{fn}\ x : \textsf{unit} \Rightarrow (l := 1)\ ;\ x)\ (l := 2)$$

$$\begin{aligned}
& \langle E\,,\ \{l \mapsto 0\}\rangle \\
\longrightarrow\ & \langle (\textbf{fn}\ x : \textsf{unit} \Rightarrow (l := 1)\ ;\ x)\ \textbf{skip}\,,\ \{l \mapsto 2\}\rangle \\
\longrightarrow\ & \langle (l := 1)\ ;\ \textbf{skip}\,,\ \{l \mapsto 2\}\rangle \\
\longrightarrow\ & \langle \textbf{skip}\ ;\ \textbf{skip}\,,\ \{l \mapsto 1\}\rangle \\
\longrightarrow\ & \langle \textbf{skip}\,,\ \{l \mapsto 1\}\rangle
\end{aligned}$$

Call-by-Value – Semantics

Values

$v ::= b \mid n \mid \textbf{skip} \mid (\textbf{fn}\ x : T \Rightarrow E)$

SOS rules

all sos rules we used so far, plus the following

(app1) $$\frac{\langle E_1\ ,\ s\rangle \longrightarrow \langle E_1'\ ,\ s'\rangle}{\langle E_1\ E_2\ ,\ s\rangle \longrightarrow \langle E_1'\ E_2\ ,\ s'\rangle}$$

(app2) $$\frac{\langle E_2\ ,\ s\rangle \longrightarrow \langle E_2'\ ,\ s'\rangle}{\langle v\ E_2\ ,\ s\rangle \longrightarrow \langle v\ E_2'\ ,\ s'\rangle}$$

(fn) $$\langle (\textbf{fn}\ x : T \Rightarrow E)\ v\ ,\ s\rangle \longrightarrow \langle \{v/x\}\,E\ ,\ s\rangle$$

Call-by-Value – Example I

$$
\begin{array}{rl}
& \langle (\textbf{fn}\ x : \mathsf{int} \Rightarrow (\textbf{fn}\ y : \mathsf{int} \Rightarrow x + y))\ (3+4)\ 5\ ,\ s\rangle \\
= & \langle \big((\textbf{fn}\ x : \mathsf{int} \Rightarrow (\textbf{fn}\ y : \mathsf{int} \Rightarrow x + y))\ (3+4)\big)\ 5\ ,\ s\rangle \\
\longrightarrow & \langle \big((\textbf{fn}\ x : \mathsf{int} \Rightarrow (\textbf{fn}\ y : \mathsf{int} \Rightarrow x + y))\ 7\big)\ 5\ ,\ s\rangle \\
\longrightarrow & \langle \big(\{7/x\}\ (\textbf{fn}\ y : \mathsf{int} \Rightarrow x + y)\big)\ 5\ ,\ s\rangle \\
= & \langle (\textbf{fn}\ y : \mathsf{int} \Rightarrow 7 + y)\ 5\ ,\ s\rangle \\
\longrightarrow & \langle \{5/y\}\ 7 + y\ ,\ s\rangle \\
= & \langle 7 + 5\ ,\ s\rangle \\
\longrightarrow & \langle 12\ ,\ s\rangle
\end{array}
$$

Call-by-Value – Example II

$$(\textbf{fn } f : \mathsf{int} \to \mathsf{int} \Rightarrow f\ 3)\ (\textbf{fn } x : \mathsf{int} \Rightarrow (1+2)+x) \quad \longrightarrow^* \quad ???$$

Choice 2: Call-by-Name

Idea: reduce left-hand-side of application to an **fn**-term; then replace all occurrences of the formal parameter in the **fn**-term by that argument.

$$E = (\textbf{fn}\ x : \mathsf{unit} \Rightarrow (l := 1)\ ;\ x)\ (l := 2)$$

$$\begin{array}{rl}
 & \langle E\,,\ \{l \mapsto 0\}\rangle \\
\longrightarrow & \langle (l := 1)\ ;\ (l := 2)\,,\ \{l \mapsto 0\}\rangle \\
\longrightarrow & \langle \textbf{skip}\ ;\ (l := 2)\,,\ \{l \mapsto 1\}\rangle \\
\longrightarrow & \langle l := 2\,,\ \{l \mapsto 1\}\rangle \\
\longrightarrow & \langle \textbf{skip}\,,\ \{l \mapsto 2\}\rangle
\end{array}$$

Call-by-Name – Semantics

SOS rules

(CBN-app) $$\frac{\langle E_1 \, , \, s \rangle \longrightarrow \langle E_1' \, , \, s' \rangle}{\langle E_1 \; E_2 \, , \, s \rangle \longrightarrow \langle E_1' \; E_2 \, , \, s' \rangle}$$

(CBN-fn) $$\langle (\textbf{fn}\ x : T \Rightarrow E_1)\ E_2 \, , \, s \rangle \longrightarrow \langle \{E_2/x\}\, E_1 \, , \, s \rangle$$

No evaluation unless needed

$$\begin{array}{rl} & \langle (\textbf{fn}\ x : \text{unit} \Rightarrow \textbf{skip})\ (l := 2) \, , \, \{l \mapsto 0\} \rangle \\ \longrightarrow & \langle \{l := 2/x\}\, \textbf{skip} \, , \, \{l \mapsto 0\} \rangle \\ = & \langle \textbf{skip} \, , \, \{l \mapsto 0\} \rangle \end{array}$$

but if it is needed, repeated evaluation possible.

Choice 3: Full Beta

Idea: allow reductions on left-hand-side and right-hand-side;
any time if left-hand-side is an **fn**-term;
replace all occurrences of the formal parameter in the **fn**-term by that argument; allow reductions inside functions

$$\langle(\textbf{fn}\ x : \mathsf{int} \Rightarrow 2 + 2)\,,\ s\rangle \longrightarrow \langle(\textbf{fn}\ x : \mathsf{int} \Rightarrow 4)\,,\ s\rangle$$

Full Beta – Semantics

Values

$v ::= b \mid n \mid \textbf{skip} \mid (\textbf{fn}\ x : T \Rightarrow E)$

SOS rules

(beta-app1) $$\frac{\langle E_1\ ,\ s\rangle \longrightarrow \langle E_1'\ ,\ s'\rangle}{\langle E_1\ E_2\ ,\ s\rangle \longrightarrow \langle E_1'\ E_2\ ,\ s'\rangle}$$

(beta-app2) $$\frac{\langle E_2\ ,\ s\rangle \longrightarrow \langle E_2'\ ,\ s'\rangle}{\langle E_1\ E_2\ ,\ s\rangle \longrightarrow \langle E_1\ E_2'\ ,\ s'\rangle}$$

(beta-fn1) $$\langle (\textbf{fn}\ x : T \Rightarrow E_1)\ E_2\ ,\ s\rangle \longrightarrow \langle \{E_2/x\}\ E_1\ ,\ s\rangle$$

(beta-fn2) $$\frac{\langle E\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle}{\langle (\textbf{fn}\ x : T \Rightarrow E)\ ,\ s\rangle \longrightarrow \langle (\textbf{fn}\ x : T \Rightarrow E')\ ,\ s'\rangle}$$

Full Beta – Example

$$
\begin{tikzcd}
 & (\textsf{\textbf{fn}}\ x : \textsf{int} \Rightarrow x + x)\ (2+2) \arrow[dl] \arrow[dr] & & \\
(\textsf{\textbf{fn}}\ x : \textsf{int} \Rightarrow x + x)\ 4 \arrow[ddr] & & (2+2) + (2+2) \arrow[dl] \arrow[dr] & \\
 & 4 + (2+2) \arrow[d] & & (2+2) + 4 \arrow[dll] \\
 & 4 + 4 \arrow[d] & & \\
 & 8 & &
\end{tikzcd}
$$

Choice 4: Normal-Order Reduction

Idea: leftmost, outermost variant of full beta.

Section 6

Typing for Call-By-Value

Typing Functions - TypeEnvironment

- so far Γ ranges over TypeEnv, the finite partial function from $\mathbb{L} \rightharpoonup T_{loc}$
- with functions, it summarises assumptions on the types of variables
- type environments Γ are now pairs of a Γ_{loc} ($\mathbb{L} \rightharpoonup T_{loc}$) and a Γ_{var}, a partial function from $\mathbb{X}$ to T ($\mathbb{X} \rightharpoonup T$).

  For example, $\Gamma_{loc} = \{l_1 : \text{intref}\}$ and $\Gamma_{var} = \{x : \text{int}, y : \text{bool} \rightarrow \text{int}\}$.
- $\text{dom}(\Gamma) = \text{dom}(\Gamma_{loc}) \cup \text{dom}(\Gamma_{var})$.
- notation: if $x \notin \text{dom}(\Gamma_{var})$, we write $\Gamma, x : T$, which adds $x : T$ to Γ_{var}

Typing Functions

(var) $\Gamma \vdash x : T$ if $\Gamma(x) = T$

(fn) $$\frac{\Gamma, x : T \ \vdash E : T'}{\Gamma \vdash (\textbf{fn}\ x : T \Rightarrow E) : T \to T'}$$

(app) $$\frac{\Gamma \vdash E_1 : T \to T' \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1\ E_2 : T'}$$

Typing Functions – Example I

$$
\dfrac{\dfrac{(\text{VAR})\ \dfrac{}{x:\mathsf{int}\vdash x:\mathsf{int}} \qquad \dfrac{}{x:\mathsf{int}\vdash 2:\mathsf{int}}\ (\text{INT})}{x:\mathsf{int}\vdash x+2:\mathsf{int}}\ (\text{OP}+)}{\{\}\vdash (\textbf{fn}\ x:\mathsf{int}\Rightarrow x+2):\mathsf{int}\to\mathsf{int}}\ (\text{FN}) \qquad \dfrac{}{\{\}\vdash 2:\mathsf{int}}\ (\text{INT})
$$

$$
\dfrac{\{\}\vdash (\textbf{fn}\ x:\mathsf{int}\Rightarrow x+2):\mathsf{int}\to\mathsf{int} \qquad \{\}\vdash 2:\mathsf{int}}{\{\}\vdash (\textbf{fn}\ x:\mathsf{int}\Rightarrow x+2)\ 2:\mathsf{int}}\ (\text{APP})
$$

Typing Functions – Example II

Determine the type of

$$(\textbf{fn } x : \mathsf{int} \to \mathsf{int} \Rightarrow x\ (\textbf{fn } x : \mathsf{int} \Rightarrow x)\ 3)$$

Properties Typing

We only consider *closed* programs, with *no* free variables.

Theorem (Progress)

If E closed, $\Gamma \vdash E : T$ and $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$ then either E is a value or there exist E' and s' such that $\langle E\,,\,s\rangle \longrightarrow \langle E'\,,\,s'\rangle$.

There are more configurations that get stuck, e.g. $(3\ 4)$.

Theorem (Type Preservation)

If E *closed,* $\Gamma \vdash E : T$*, dom*$(\Gamma) \subseteq$ *dom*(s) *and* $\langle E\,,\,s\rangle \longrightarrow \langle E'\,,\,s'\rangle$ *then* $\Gamma \vdash E' : T$ *and dom*$(\Gamma) \subseteq$ *dom*(s')*.*

Proving Type Preservation

Theorem (Type Preservation)

If E closed, $\Gamma \vdash E : T$, dom$(\Gamma) \subseteq$ dom(s) and $\langle E\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle$ then $\Gamma \vdash E' : T$ and dom$(\Gamma) \subseteq$ dom(s').

Proof outline.

Choose

$$\Phi(E, s, E', s') = \forall \Gamma, T.\ \big(\Gamma \vdash E : T \land \mathsf{closed}(E) \land \mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(s) \\ \implies \Gamma \vdash E' : T \land \mathsf{closed}(E') \land \mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(s')\big)$$

show $\forall E, s, E', s'.\ \langle E\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle \implies \Phi(E, s, E', s')$, by rule induction $\square$

Proving Type Preservation – Auxiliary Lemma

Lemma (Substitution)

If E closed, $\Gamma \vdash E : T$ and $\Gamma, x : T \vdash E' : T'$ with $x \notin$ dom(Γ) then $\Gamma \vdash \{E/x\}\, E' : T'$.

Type Safety

Main result: Well-typed programs do not get stuck.

Theorem (Type Safety)

If $\Gamma \vdash E : T$*, dom*$(\Gamma) \subseteq$ *dom*(s)*, and* $\langle E\,,\, s\rangle \longrightarrow^* \langle E'\,,\, s'\rangle$ *then either* E' *is a value with* $\Gamma \vdash E' : T$*, or there exist* E''*,* s'' *such that* $\langle E'\,,\, s'\rangle \longrightarrow \langle E''\,,\, s''\rangle$*,* $\Gamma \vdash E'' : T$ *and dom*$(\Gamma) \subseteq$ *dom*(s'')*.*

Here, $\longrightarrow^*$ means arbitrary many steps in the transition system.

Normalisation

Theorem (Normalisation)

In the sublanguage without while loops, if $\Gamma \vdash E : T$ *and* E *closed then there does not exist an infinite reduction sequence*

$$\langle E \,,\, \{\} \rangle \longrightarrow \langle E_1 \,,\, \{\} \rangle \longrightarrow \langle E_2 \,,\, \{\} \rangle \longrightarrow \ldots$$

Proof.

See B. Pierce, Types and Programming Languages, Chapter 12. $\square$

Section 7

Recursion

Scoping

Name Definitions
restrict the scope of variables

$$E ::= \ldots \mid \textbf{let val } x : T = E_1 \textbf{ in } E_2 \textbf{ end}$$

- x is a binder for E_2
- can be seen as syntactic sugar:

$$\textbf{let val } x : T = E_1 \textbf{ in } E_2 \textbf{ end} \quad \equiv \quad (\textbf{fn } x : T \Rightarrow E_2)\ E_1$$

Derived sos-rules and typing

$$\textbf{let val } x : T = E_1 \textbf{ in } E_2 \textbf{ end} \equiv (\textbf{fn } x : T \Rightarrow E_2)\ E_1$$

(let) $$\frac{\Gamma \vdash E_1 : T \qquad \Gamma, x : T \vdash E_2 : T'}{\Gamma \vdash \textbf{let val } x : T = E_1 \textbf{ in } E_2 \textbf{ end} : T'}$$

(let1) $$\frac{\langle E_1\,,\ s\rangle \longrightarrow \langle E_1'\,,\ s'\rangle}{\langle \textbf{let val } x{:}T{=}E_1 \textbf{ in } E_2 \textbf{ end}\,,\ s\rangle \longrightarrow \langle \textbf{let val } x{:}T{=}E_1' \textbf{ in } E_2 \textbf{ end}\,,\ s'\rangle}$$

(let2) $$\langle \textbf{let val } x : T = v \textbf{ in } E_2 \textbf{ end}\,,\ s\rangle \longrightarrow \langle \{v/x\}\, E_2\,,\ s\rangle$$

Recursion – An Attempt

Consider

$$r = (\textbf{fn}\ y : \text{int} \Rightarrow \textbf{if}\ y \geq 1\ \textbf{then}\ y + (r\,(y + \ -1))\ \textbf{else}\ 0)$$

where r is the recursive call (variable occurring in itself).
What is the evaluation of $r\ 3$?

We could try

$$E ::= \ldots \mid \textbf{let val rec}\ x : T = E\ \textbf{in}\ E'\ \textbf{end}$$

where x is a binder for both E and E'.

$$\begin{array}{l} \textbf{let val rec}\ r : \text{int} \to \text{int} = \\ \qquad (\textbf{fn}\ y : \text{int} \Rightarrow \textbf{if}\ y \geq 1\ \textbf{then}\ y + (r\,(y + \ -1))\ \textbf{else}\ 0) \\ \ \textbf{in}\ r\ 3\ \textbf{end} \end{array}$$

Australian National University

However …

- What about **let val rec** $x : T = (x, x)$ **in** x **end**?
- What about **let val rec** x : int list $= 3 :: x$ **in** x **end**?
  Does this terminate? and if it does is it equal to
  – **let val rec** x : int list $= 3 :: 3 :: x$ **in** x **end**
- Does **let val rec** x : int list $= 3 :: (x + 1)$ **in** x **end** terminate?
- In Call-by-Name (Call-by-Need) these are reasonable
- In Call-by-Value these would usually be disallowed

Recursive Functions

Idea specialise the previous **let val rec**

- $T = T_1 \to T_2$ (recursion only at function types)
- $E = (\textbf{fn}\ y : T_1 \Rightarrow E_1)$ (and only for function values)

Recursive Functions – Syntax and Typing

$E ::= \ldots \mid$ **let val rec** $x : T_1 \to T_2 = ($**fn** $y : T_1 \Rightarrow E_1)$ **in** E_2 **end**

Here, y binds in E_1 and x bind in $($**fn** $y : T_1 \Rightarrow E_1)$ and E_2

$$\text{(recT)} \quad \frac{\Gamma, x{:}T_1 \to T_2,\ y{:}T_1 \vdash E_1 : T_2 \qquad \Gamma, x{:}T_1 \to T_2 \vdash E_2 : T}{\Gamma \vdash \textbf{let val rec}\ x : T_1 \to T_2 = (\textbf{fn}\ y : T_1 \Rightarrow E_1)\ \textbf{in}\ E_2\ \textbf{end} : T}$$

Recursive Functions – Semantics

(rec) $\langle \textbf{let val rec}\ x : T_1 \to T_2 = (\textbf{fn}\ y : T_1 \Rightarrow E_1)\ \textbf{in}\ E_2\ \textbf{end}\,,\ s\rangle$

$$\longrightarrow$$

$\langle \{(\textbf{fn}\ y : T_1 \Rightarrow \textbf{let val rec}\ x : T_1 \to T_2 = (\textbf{fn}\ y : T_1 \Rightarrow E_1)\ \textbf{in}\ E_1\ \textbf{end})/x\}\, E_2\,,\ s\rangle$

Redundancies?

- Do we need $E_1 \ ; E_2$?
  No: $E_1 \ ; E_2 \equiv (\textbf{fn}\ y : \text{unit} \Rightarrow E_2)\ E_1$

- Do we need **while** E_1 **do** E_2?
  No:

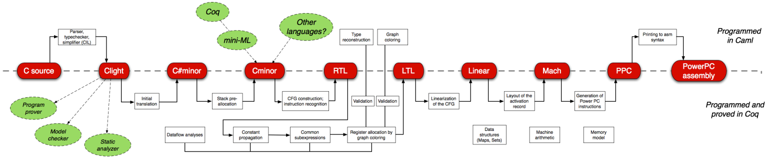
$$
\begin{aligned}
\textbf{while}\ E_1\ \textbf{do}\ E_2 \equiv\ & \textbf{let val rec}\ w : \text{unit} \rightarrow \text{unit}\ = \\
& \quad (\textbf{fn}\ y : \text{unit} \Rightarrow \textbf{if}\ E_1\ \textbf{then}\ (E_2 \ ; (w\ \textbf{skip}))\ \textbf{else skip}) \\
& \textbf{in} \\
& \qquad w\ \textbf{skip} \\
& \textbf{end}
\end{aligned}
$$

Redundancies?

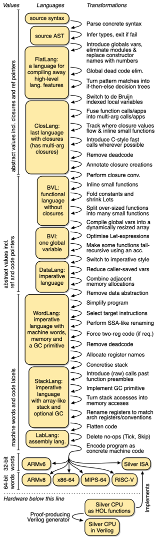
- Do we need recursion?
  Yes! Previously, normalisation theorem effectively showed that **while** adds expressive power; now, recursion is even more powerful.

Side remarks I

- naive implementations (in particular substitutions) are inefficient (more efficient implementations are shown in courses on compiler construction)
- more concrete – closer to implementation or machine code – are possible
- usually refinement to prove compiler to be correct (e.g. CompCert or CakeML)



Side remarks I – CakeML



Side remarks II: Big-step Semantics

- we have seen a **small-step semantics**

$$\langle E \,,\, s\rangle \longrightarrow \langle E' \,,\, s'\rangle$$

- alternatively, we could have looked at a **big-step semantics**

$$\langle E \,,\, s\rangle \Downarrow \langle E' \,,\, s'\rangle$$

For example

$$\frac{}{\langle n \,,\, s\rangle \Downarrow \langle n \,,\, s\rangle} \qquad \frac{\langle E_1 \,,\, s\rangle \Downarrow \langle n_1 \,,\, s'\rangle \qquad \langle E_2 \,,\, s'\rangle \Downarrow \langle n_2 \,,\, s''\rangle}{\langle E_1 + E_2 \,,\, s\rangle \Downarrow \langle n \,,\, s''\rangle}(n = n_1 + n_2)$$

- no major difference for sequential programs
- small-step much better for modelling concurrency and proving type safety

Section 8

Data

Recap and Missing Steps

- simple while language
- with functions
- **but no data structures**

Products – Syntax

$$T ::= \ldots \mid T * T$$

$$E ::= \ldots \mid (E, E) \mid \textbf{fst}\ E \mid \textbf{snd}\ E$$

Products – Typing

(pair)
$$\frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash (E_1, E_2) : T_1 * T_2}$$

(proj1)
$$\frac{\Gamma \vdash E : T_1 * T_2}{\Gamma \vdash \textbf{fst}\ E : T_1}$$

(proj2)
$$\frac{\Gamma \vdash E : T_1 * T_2}{\Gamma \vdash \textbf{snd}\ E : T_2}$$

Products – Semantics

Values

$v ::= \ldots \mid (v, v)$

SOS rules

(pair1) $$\frac{\langle E_1\,,\ s\rangle \longrightarrow \langle E_1'\,,\ s'\rangle}{\langle (E_1, E_2)\,,\ s\rangle \longrightarrow \langle (E_1', E_2)\,,\ s'\rangle}$$

(pair2) $$\frac{\langle E_2\,,\ s\rangle \longrightarrow \langle E_2'\,,\ s'\rangle}{\langle (v, E_2)\,,\ s\rangle \longrightarrow \langle (v, E_2')\,,\ s'\rangle}$$

(proj1) $\langle \textbf{fst}\ (v_1, v_2)\,,\ s\rangle \longrightarrow \langle v_1\,,\ s\rangle$

(proj2) $\langle \textbf{snd}\ (v_1, v_2)\,,\ s\rangle \longrightarrow \langle v_2\,,\ s\rangle$

(proj3) $$\frac{\langle E\,,\ s\rangle \longrightarrow \langle E'\,,\ s'\rangle}{\langle \textbf{fst}\ E\,,\ s\rangle \longrightarrow \langle \textbf{fst}\ E'\,,\ s'\rangle}$$

(proj4) $$\frac{\langle E\,,\ s\rangle \longrightarrow \langle E'\,,\ s'\rangle}{\langle \textbf{snd}\ E\,,\ s\rangle \longrightarrow \langle \textbf{snd}\ E'\,,\ s'\rangle}$$

Sums (Variants, Tagged Unions) – Syntax

$$T ::= \ldots \mid T + T$$

$$\begin{aligned} E ::= \ldots &\mid \textbf{inl}\ E : T \mid \textbf{inr}\ E : T \mid \\ &\textbf{case}\ E\ \textbf{of inl}\ x_1 : T_1 \Rightarrow E \mid \textbf{inr}\ x_2 : T_2 \Rightarrow E \end{aligned}$$

x_1 and x_2 are binders for E_1 and E_2, up to alpha-equivalence

Sums – Typing I

(inl) $$\frac{\Gamma \vdash E : T_1}{\Gamma \vdash \textbf{inl}\ E : T_1 + T_2 : T_1 + T_2}$$

(inr) $$\frac{\Gamma \vdash E : T_2}{\Gamma \vdash \textbf{inr}\ E : T_1 + T_2 : T_1 + T_2}$$

(case) $$\frac{\Gamma \vdash E : T_1 + T_2 \qquad \Gamma, x : T_1 \vdash E_1 : T \qquad \Gamma, y : T_2 \vdash E_2 : T}{\Gamma \vdash \textbf{case}\ E\ \textbf{of inl}\ x : T_1 \Rightarrow E_1 \mid \textbf{inr}\ y : T_2 \Rightarrow E_2 : T}$$

Sums – Typing II

$$\textbf{case } E \textbf{ of inl } x : T_1 \Rightarrow E_1 \mid \textbf{inr } y : T_2 \Rightarrow E_2$$

Why do we need to carry around type annotations?

- maintain the unique typing property
  Otherwise **inl** $3:$ could be of type int $+$ int or int $+$ bool
- many programming languages allow type polymorphism

Sums – Semantics

Values

$v ::= \ldots \mid \textbf{inl}\ v : T \mid \textbf{inr}\ v : T$

SOS rules

(inl) $$\frac{\langle E\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle}{\langle \textbf{inl}\ E : T\ ,\ s\rangle \longrightarrow \langle \textbf{inl}\ E' : T\ ,\ s'\rangle}$$

(inr) $$\frac{\langle E\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle}{\langle \textbf{inr}\ E : T\ ,\ s\rangle \longrightarrow \langle \textbf{inr}\ E' : T\ ,\ s'\rangle}$$

(case1) $$\frac{\langle E\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle}{\begin{array}{c}\langle \textbf{case}\ E\ \textbf{of inl}\ x : T_1 \Rightarrow E_1 \mid \textbf{inr}\ y : T_2 \Rightarrow E_2\ ,\ s\rangle \\ \longrightarrow \langle \textbf{case}\ E'\ \textbf{of inl}\ x : T_1 \Rightarrow E_1 \mid \textbf{inr}\ y : T_2 \Rightarrow E_2\ ,\ s'\rangle\end{array}}$$

(case2) $$\begin{array}{l}\langle \textbf{case inl}\ v : T\ \textbf{of inl}\ x : T_1 \Rightarrow E_1 \mid \textbf{inr}\ y : T_2 \Rightarrow E_2\ ,\ s\rangle \\ \longrightarrow \langle \{v/x\}\, E_1\ ,\ s\rangle\end{array}$$

(case3) $$\begin{array}{l}\langle \textbf{case inr}\ v : T\ \textbf{of inl}\ x : T_1 \Rightarrow E_1 \mid \textbf{inr}\ y : T_2 \Rightarrow E_2\ ,\ s\rangle \\ \longrightarrow \langle \{v/y\}\, E_2\ ,\ s\rangle\end{array}$$

Constructors and Destructors

type	constructors	destructors
$T \to T$	(**fn** $x : T \Rightarrow$ _)	_ E
$T * T$	(_, _)	**fst** _ **snd** _
$T + T$	**inl** _ $: T$ **inr** _ $: T$	**case**
bool	`true` `false`	**if** _ **then** _ **else** _

Proofs as Programs

The Curry-Howard correspondence

(var)	$\Gamma, x : T \vdash x : T$	$\Gamma, P \vdash P$
(fn)	$\dfrac{\Gamma, x : T \vdash E : T'}{\Gamma \vdash (\textbf{fn}\ x : T \Rightarrow E) : T \to T'}$	$\dfrac{\Gamma, P \vdash P'}{\Gamma \vdash P \to P'}$
(app)	$\dfrac{\Gamma \vdash E_1 : T \to T' \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1\ E_2 : T'}$	$\dfrac{\Gamma \vdash P \to P' \qquad \Gamma \vdash P}{\Gamma \vdash P'}$ (modus ponens)

...

Proofs as Programs: The Curry-Howard correspondence

(var)	$\Gamma, x{:}T \vdash x : T$	$\Gamma, P \vdash P$
(fn)	$\dfrac{\Gamma, x{:}T \vdash E : T'}{\Gamma \vdash (\textbf{fn}\ x : T \Rightarrow E) : T \to T'}$	$\dfrac{\Gamma, P \vdash P'}{\Gamma \vdash P \to P'}$
(app)	$\dfrac{\Gamma \vdash E_1 : T \to T' \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1\ E_2 : T'}$	$\dfrac{\Gamma \vdash P \to P' \qquad \Gamma \vdash P}{\Gamma \vdash P'}$ (modus ponens)
(pair)	$\dfrac{\Gamma \vdash E_1 : T_1 \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash (E_1, E_2) : T_1 * T_2}$	$\dfrac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2}$
(proj1) (proj2)	$\dfrac{\Gamma \vdash E : T_1 * T_2}{\Gamma \vdash \textbf{fst}\ E : T_1}$ $\qquad$ $\dfrac{\Gamma \vdash E : T_1 * T_2}{\Gamma \vdash \textbf{snd}\ E : T_2}$	$\dfrac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_1}$ $\qquad$ $\dfrac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_2}$
(inl) (inr)	$\dfrac{\Gamma \vdash E : T_1}{\Gamma \vdash \textbf{inl}\ E : T_1 + T_2 : T_1 + T_2}$ $\qquad$ $\dfrac{\Gamma \vdash E : T_2}{\Gamma \vdash \textbf{inr}\ E : T_1 + T_2 : T_1 + T_2}$	$\dfrac{\Gamma \vdash P_1}{\Gamma \vdash P_1 \vee P_2}$ $\qquad$ $\dfrac{\Gamma \vdash P_2}{\Gamma \vdash P_1 \vee P_2}$
(case)	$\dfrac{\Gamma \vdash E : T_1 + T_2 \qquad \Gamma, x : T_1 \vdash E_1 : T \qquad \Gamma, y : T_2 \vdash E_2 : T}{\Gamma \vdash \textbf{case}\ E\ \textbf{of inl}\ x : T_1 \Rightarrow E_1 \mid \textbf{inr}\ y : T_2 \Rightarrow E_2 : T}$	$\dfrac{\Gamma \vdash P_1 \vee P_2 \quad \Gamma, P_1 \vdash P \quad \Gamma, P_2 \vdash P}{\Gamma \vdash P}$

(unit), (zero), . . . ; **but not (letrec)**

Curry-Howard correspondence (abstract)

Programming side	**Logic side**
bottom type	false formula
unit type	true formula
sum type	disjunction
product type	conjunction
function type	implication
generalised sum type (Σ type)	existential quantification
generalised product type (Π type)	universal quantification

Datatypes in Haskell

Datatypes in Haskell generalise both sums and products

```
data Pair = P Int Double
data Either = I Int | D Double
```

The expression

```
data Expr = IntVal Int
          | BoolVal Bool
          | PairVal Int Bool
```

is (roughly) like saying

$$\text{Expr} = \text{int} + \text{bool} + (\text{int} * \text{bool})$$

More Datatypes - Records

A generalisation of products.
Labels $lab \in \mathbb{LAB}$ for a set $\mathbb{LAB} = \{p, q, ...\}$

$$
\begin{aligned}
T &::= \ldots \mid \{lab_1 : T_1, \ldots, lab_k : T_k\} \\
E &::= \ldots \mid \{lab_1 = E_1, \ldots, lab_k = E_k\} \mid \#lab\ E
\end{aligned}
$$

(where in each record (type or expression) no lab occurs more than once)

Records – Typing

(record)

$$\frac{\Gamma \vdash E_1 : T_1 \qquad \ldots \qquad \Gamma \vdash E_k : T_k}{\Gamma \vdash \{lab_1 = E_1, \ldots, lab_k = E_k\} : \{lab_1 : T_1, \ldots, lab_k : T_k\}}$$

(recordproj)

$$\frac{\Gamma \vdash E : \{lab_1 : T_1, \ldots, lab_k : T_k\}}{\Gamma \vdash \#lab_i\ E : T_i}$$

Records – Semantics

Values

$v ::= \ldots \mid \{lab_1 = v_1, \ldots, lab_k = v_k\}$

SOS rules

(record1)
$$\frac{\langle E_i\ ,\ s\rangle \longrightarrow \langle E_i'\ ,\ s'\rangle}{\begin{array}{l}\langle \{lab_1 = v_1, \ldots, lab_{i-1} = v_{i-1}, lab_i = E_i, \ldots, lab_k = E_k\}\ ,\ s\rangle \\ \longrightarrow \langle \{lab_1 = v_1, \ldots, lab_{i-1} = v_{i-1}, lab_i = E_i', \ldots, lab_k = E_k\}\ ,\ s'\rangle\end{array}}$$

(record2) $\langle \#lab_i\ \{lab_1 = v_1, \ldots, lab_k = v_k\}\ ,\ s\rangle \longrightarrow \langle v_i\ ,\ s\rangle$

(record3)
$$\frac{\langle E\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle}{\langle \#lab_i\ E\ ,\ s\rangle \longrightarrow \langle \#lab_i\ E'\ ,\ s'\rangle}$$

Mutable Store I

Most languages have some kind of mutable store.
Two main choices:

1. our approach

$$E ::= \dots \mid l := E \mid\ !l \mid x$$

- locations store mutable values
- variables refer to a previously calculated value – immutable
- explicit dereferencing and assignment
  ($\textbf{fn}\ x : \text{int} \Rightarrow l := (!l) + x$)

Mutable Store II

Most languages have some kind of mutable store.
Two main choices:

2. languages as C or Java
    - variables can refer to a previously calculated value and overwrite that value
    - implicit dereferencing
    - some limited type machinery to limit mutability

```
void foo(x:int) {
  l = l + x
  ...
}
```

References

$$
\begin{aligned}
T & ::= \ \ldots \mid T \ \mathsf{ref} \\
T_{loc} & ::= \ \cancel{\mathsf{intref}} \ \ T \ \mathsf{ref} \\
E & ::= \ \cdots \mid \cancel{\ell\ell := E} \mid \cancel{!\ell} \\
& \quad\ \ \mid E_1 := E_2 \mid {!E} \mid \mathsf{ref}\ E \mid l
\end{aligned}
$$

References – Typing

(ref)
$$\frac{\Gamma \vdash E : T}{\Gamma \vdash \text{ref } E : T \text{ ref}}$$

(assign)
$$\frac{\Gamma \vdash E_1 : T \text{ ref} \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 := E_2 : \text{unit}}$$

(deref)
$$\frac{\Gamma \vdash E : T \text{ ref}}{\Gamma \vdash !E : T}$$

(loc)
$$\frac{\Gamma(l) = T \text{ ref}}{\Gamma \vdash l : T \text{ ref}}$$

References – Semantics I

Values

A location is a value $v ::= \ldots \mid l$

Stores s were finite partial functions $\mathbb{L} \rightharpoonup \mathbb{Z}$.
We now take them to be finite partial functions from $\mathbb{L}$ to all values.

SOS rules

(ref1) $\langle \text{ref } v \,,\, s \rangle \longrightarrow \langle l \,,\, s + \{l \mapsto v\} \rangle \quad \text{if } l \notin \text{dom}(s)$

(ref2) $$\frac{\langle E \,,\, s \rangle \longrightarrow \langle E' \,,\, s' \rangle}{\langle \text{ref } E \,,\, s \rangle \longrightarrow \langle \text{ref } E' \,,\, s' \rangle}$$

References – Semantics II

(deref1) $\langle !l\,,\ s\rangle \longrightarrow \langle v\,,\ s\rangle$ if $l \in \mathrm{dom}(s)$ and $s(l) = v$

(deref2) $$\frac{\langle E\,,\ s\rangle \longrightarrow \langle E'\,,\ s'\rangle}{\langle !E\,,\ s\rangle \longrightarrow \langle !E'\,,\ s'\rangle}$$

(assign1) $\langle l := v\,,\ s\rangle \longrightarrow \langle \textbf{skip}\,,\ s + \{l \mapsto v\}\rangle$ if $l \in \mathrm{dom}(s)$

(assign2) $$\frac{\langle E\,,\ s\rangle \longrightarrow \langle E'\,,\ s'\rangle}{\langle l := E\,,\ s\rangle \longrightarrow \langle l := E'\,,\ s'\rangle}$$

(assign3) $$\frac{\langle E\,,\ s\rangle \longrightarrow \langle E'\,,\ s'\rangle}{\langle E := E_2\,,\ s\rangle \longrightarrow \langle E' := E_2\,,\ s'\rangle}$$

Type Checking the Store

- so far we used dom$(\Gamma) \subseteq$ dom(s) in theorems such as progress and type preservation
- expressed ‘all locations in Γ exist in store s’
- we need more
- for each $l \in$ dom(s) we require that $s(l)$ is typable
- moreover, $s(l)$ might contain some other locations ...

Type Checking – Example

Example

$$E \;=\; \textbf{let val } x : (\text{int} \to \text{int})\ \text{ref} = \text{ref}\ (\textbf{fn } z : \text{int} \Rightarrow z)\ \textbf{in}$$
$$(x := (\textbf{fn } z : \text{int} \Rightarrow \textbf{if } z \geq 1 \textbf{ then } z + ((!x)\ (z + -1))\ \textbf{else } 0);$$
$$(!x)\ 3)\ \textbf{end}$$

which has reductions

$$
\begin{array}{rl}
 & \langle E\, ,\, \{\}\rangle \\
\longrightarrow^* & \langle E_1\, ,\, \{l_1 \mapsto (\textbf{fn } z : \text{int} \Rightarrow z)\rangle \\
\longrightarrow^* & \langle E_2\, ,\, \{l_1 \mapsto (\textbf{fn } z : \text{int} \Rightarrow \textbf{if } z \geq 1 \textbf{ then } z + ((!l_1)(z + -1))\ \textbf{else } 0)\}\rangle \\
\longrightarrow^* & \langle 6\, ,\, \ldots\rangle
\end{array}
$$

Progress and Type Preservation

Definition (Well-type store)

Let $\Gamma \vdash s$ if $\mathrm{dom}(\Gamma) = \mathrm{dom}(s)$ and if
$\forall l \in \mathrm{dom}(s).\ \Gamma(l) = T\ \mathsf{ref} \Longrightarrow \Gamma \vdash s(l) : T$.

Theorem (Progress)

If E closed, $\Gamma \vdash E : T$ and $\Gamma \vdash s$ then either E is a value or there exist E' and s' such that $\langle E\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle$.

Theorem (Type Preservation)

If E closed, $\Gamma \vdash E : T$, $\Gamma \vdash s$ and $\langle E\ ,\ s\rangle \longrightarrow \langle E'\ ,\ s'\rangle$ then E' is closed and for some Γ' (with disjoint domain to Γ) $\Gamma, \Gamma' \vdash E' : T$ and $\Gamma, \Gamma' \vdash s'$.

Type Safety

Theorem (Type Safety)

If E closed, $\Gamma \vdash E : T$, $\Gamma \vdash s$, and $\langle E\,,\, s\rangle \longrightarrow^{*} \langle E'\,,\, s'\rangle$ then either E' is a value with $\Gamma \vdash E' : T$, or there exist E'', s'' such that $\langle E'\,,\, s'\rangle \longrightarrow \langle E''\,,\, s''\rangle$, and there exists a Γ' s.t. $\Gamma, \Gamma' \vdash E'' : T$ and $\Gamma, \Gamma' \vdash s''$.

Section 9

Exceptions

Motivation

Trapped errors

Cause execution to halt immediately.
Examples: jumping to an illegal address, raising a top-level exception.
Innocuous?

Untrapped errors

May go unnoticed for a while and later cause arbitrary behaviour.
Examples: accessing data past the end of an array, security loopholes in Java abstract machines.
Insidious!

program should signal error

- devision by zero
- index out of bound (e.g. record type)
- …
- lookup key missing
- file not found

Choice 1: Raising Exceptions

Idea: introduce term $\mathbf{error}$ that completely aborts an evaluation of a term.

$$E ::= \ldots \mid \mathbf{error}$$

(no change of values nor types)

(err) $\quad \Gamma \vdash \mathbf{error} : T$

Errors – Semantics

SOS rules

(apperr1) $\langle \textbf{error}\ E\ ,\ s\rangle \longrightarrow \langle \textbf{error}\ ,\ s\rangle$

(apperr2) $\langle v\ \textbf{error}\ ,\ s\rangle \longrightarrow \langle \textbf{error}\ ,\ s\rangle$

Errors

- $(\textbf{fn}\ x : \text{int} \Rightarrow x)\ \mathbf{error} \rightarrow ?$
- $\textbf{let val rec}\ x : \text{int} \rightarrow \text{int} = (\textbf{fn}\ y : \text{int} \Rightarrow y)\ \textbf{in}\ x\ \mathbf{error}\ \textbf{end} \rightarrow ?$

- $\mathbf{error}$ can have arbitrary type, which violates type uniqueness (can be fixed by subtyping)
- type preservation is maintained
- progress property needs adaptation (homework 2)

Choice 2: Handling Exceptions

Idea: install exception handlers (e.g. ML or Java)

$$E ::= \ldots \mid \textbf{try } E \textbf{ with } E$$

(no change of values nor types)

Handling Exceptions – Typing and Semantics

try E_1 **with** E_2 means 'return result of evaluating E_1, unless it aborts, in which case the handler E_2 is evaluated'

Typing

$$\text{(try)}\quad \frac{\Gamma \vdash E_1 : T \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash \textbf{try } E_1 \textbf{ with } E_2 : T}$$

SOS rules

$$\text{(try1)}\quad \langle \textbf{try } v \textbf{ with } E\ ,\ s\rangle \longrightarrow \langle v\ ,\ s\rangle$$

$$\text{(try2)}\quad \langle \textbf{try } \mathbf{error} \textbf{ with } E\ ,\ s\rangle \longrightarrow \langle E\ ,\ s\rangle$$

$$\text{(try3)}\quad \frac{\langle E_1\ ,\ s\rangle \longrightarrow \langle E_1'\ ,\ s'\rangle}{\langle \textbf{try } E_1 \textbf{ with } E_2\ ,\ s\rangle \longrightarrow \langle \textbf{try } E_1' \textbf{ with } E_2\ ,\ s'\rangle}$$

Choice 3: Exceptions with Values

Idea: inform user about type of error

$$E ::= \ldots \mid \cancel{\textbf{error}} \mid \textbf{raise } E \mid \textbf{try } E \textbf{ with } E$$

(no change of values)

Exceptions with Values – Typing

Typing

(try_ex) $$\frac{\Gamma \vdash E : T_{ex}}{\Gamma \vdash \textbf{raise } E : T}$$

(try_v) $$\frac{\Gamma \vdash E_1 : T \qquad \Gamma \vdash E_2 : T_{ex} \to T}{\Gamma \vdash \textbf{try } E_1 \textbf{ with } E_2 : T}$$

Exceptions with Values – Semantics

SOS rules

(apprai1) $\langle (\textbf{raise}\ v)\ E\ ,\ s\rangle \longrightarrow \langle \textbf{raise}\ v\ ,\ s\rangle$

(apprai2) $\langle v_1\ (\textbf{raise}\ v_2)\ ,\ s\rangle \longrightarrow \langle \textbf{raise}\ v_2\ ,\ s\rangle$

(rai) $$\frac{\langle E\ ,\ s\rangle \longrightarrow \langle E_1'\ ,\ s'\rangle}{\langle \textbf{raise}\ E\ ,\ s\rangle \longrightarrow \langle \textbf{raise}\ E'\ ,\ s'\rangle}$$

(rai2) $\langle \textbf{raise}\ (\textbf{raise}\ v)\ ,\ s\rangle \longrightarrow \langle \textbf{raise}\ v\ ,\ s\rangle$

(try1) $\langle \textbf{try}\ v\ \textbf{with}\ E\ ,\ s\rangle \longrightarrow \langle v\ ,\ s\rangle$

(try2) $\langle \textbf{try}\ \textbf{raise}\ v\ \textbf{with}\ E\ ,\ s\rangle \longrightarrow \langle E\ v\ ,\ s\rangle$

(try3) $$\frac{\langle E_1\ ,\ s\rangle \longrightarrow \langle E_1'\ ,\ s'\rangle}{\langle \textbf{try}\ E_1\ \textbf{with}\ E_2\ ,\ s\rangle \longrightarrow \langle \textbf{try}\ E_1'\ \textbf{with}\ E_2\ ,\ s'\rangle}$$

The Type T_{ex} (I)

- $T_{ex} =$ nat: corresponds to `errno` in Unix OSs;
  0 indicates success; other values report various exceptional conditions.
  (similar in `C++`).
- $T_{ex} =$ string: avoids looking up error codes; more descriptive; error handling may now require parsing a string
- T_{ex} could be of type record

$$
\begin{aligned}
T_{ex} ::= \{&\texttt{dividedByZero} : \mathsf{unit},\\
&\texttt{overflow} : \mathsf{unit},\\
&\texttt{fileNotFound} : \mathsf{string},\\
&\texttt{fileNotReadable} : \mathsf{string},\\
&\ldots\}
\end{aligned}
$$

The Type T_{ex} (II)

- 'T_{ex} in ML': make records more flexible to allow fields to be added, sometimes called *extensible records* or *extensible variant type*
- 'T_{ex} in Java': use of classes, uses keyword `throwable`, which allows the declaration of new errors. (We do not yet know what an object is)
- …

Section 10

Subtyping

Motivation (I)

- so far we carried around types explicitly to avoid ambiguity of types
- programming languages use polymorphisms to allow different types
- some of it can be captured by *subtyping*
- common in all object-oriented languages
- subtyping is cross-cutting extension, interacting with most other language features

Polymorphism

Ability to use expressions at many different types

- ad-hoc polymorphism (overloading),
  e.g. $+$ can be used to add two integers and two reals,
  see Haskell type classes
- Parametric Polymorphism (e.g. ML or Isabelle)
  write a function that for any type α takes an argument of type α list
  and computes its length (parametric – uniform in whatever α is)
- *Subtype polymorphism* – as in various OO languages. See here.

Motivation (II)

$$\text{(app)} \quad \frac{\Gamma \vdash E_1 : T \to T' \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1\ E_2 : T'}$$

we cannot type

$$\Gamma \nvdash (\textbf{fn}\ x : \{p : \mathsf{int}\} \Rightarrow \#p\ x)\ \{p = 3, q = 4\} : \mathsf{int}$$
$$\Gamma \nvdash (\textbf{fn}\ x : \mathsf{int} \Rightarrow\ x)\ 3 : \mathsf{int} \quad \text{(assuming } 3 \text{ is of type nat)}$$

even though the function gets a ‘better’ argument, with more structure

Subsumption

better: any term of type $\{p : \text{int}, q : \text{int}\}$ can be used wherever a term of type $\{p : \text{int}\}$ is expected.

Introduce a *subtyping relation* between types

- T is a subtype of T' (a T is useful in more contexts than a T')

$$T <: T'$$

- should include $\{p : \text{int}, q : \text{int}\} <: \{p : \text{int}\} <: \{\}$
- introduce *subsumption rule*

$$\text{(sub)} \quad \frac{\Gamma \vdash E : T \qquad T <: T'}{\Gamma \vdash E : T'}$$

Example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \{p{:}\mathsf{int}\} \vdash x : \{p{:}\mathsf{int}\}}\ (\text{var})}{x : \{p{:}\mathsf{int}\} \vdash \#p\ x : \mathsf{int}}\ (\text{recordproj})}{\{\} \vdash (\mathbf{fn}\ x : \{p{:}\mathsf{int}\} \Rightarrow \#p\ x) : \{p{:}\mathsf{int}\} \to \mathsf{int}}\ (\text{fn}) \qquad \dfrac{\dfrac{\dfrac{}{\{\} \vdash 3 : \mathsf{int}}\ (\text{var}) \quad \dfrac{}{\{\} \vdash 4 : \mathsf{int}}\ (\text{var})}{\{\} \vdash \{p{=}3, q{=}4\} : \{p{:}\mathsf{int}, q{:}\mathsf{int}\}}\ (\text{record}) \qquad \{p{:}\mathsf{int}, q{:}\mathsf{int}\} <: \{p{:}\mathsf{int}\}}{\{\} \vdash \{p{=}3, q{=}4\} : \{p{:}\mathsf{int}\}}\ (\text{sub})}{\{\} \vdash (\mathbf{fn}\ x : \{p{:}\mathsf{int}\} \Rightarrow \#p\ x)\ \{p{=}3, q{=}4\} : \mathsf{int}}\ (\text{app})$$

The Subtype Relation $<:$

(s-refl) $$T <: T$$

(s-trans) $$\frac{T <: T' \qquad T' <: T''}{T <: T''}$$

the subtype order is not anti-symmetric – it is a preorder

Subtyping – Records

(s-rcd1) $\{lab_1{:}T_1, \ldots, lab_k{:}T_k, lab_{k+1}{:}T_{k+1}, .., lab_{k+n}{:}T_{k+n}\}$
$<: \{lab_1{:}T_1, \ldots, lab_k{:}T_k\}$

e.g. $\{p{:}\mathrm{int}, q{:}\mathrm{int}\} <: \{p{:}\mathrm{int}\}$

(s-rcd2) $$\frac{T_1 <: T_1' \quad \ldots T_k <: T_k'}{\{lab_1{:}T_1, \ldots, lab_k{:}T_k\} <: \{lab_1 : T_1', \ldots, lab_k{:}T_k'\}}$$

(s-rcd3) $$\frac{\pi \text{ a permutation of } 1, \ldots, k}{\{lab_1{:}T_1, \ldots, lab_k{:}T_k\} <: \{lab_{\pi(1)} : T_{\pi(1)}, \ldots, lab_{\pi(k)}{:}T_{\pi(k)}\}}$$

Subtyping – Functions (I)

$$\text{(s-fn)} \quad \frac{T_1' <: T_1 \qquad T_2 <: T_2'}{T_1 \to T_2 <: T_1' \to T_2'}$$

- *contravariant* on the left of $\to$
- *covariant* on the right of $\to$

Subtyping – Functions (II)

If $f : T_1 \to T_2$ then
– f can use any argument which is a subtype of T_1;
– the result of f can be regarded as any supertype of T_2

Example: let $f = (\textbf{fn}\ x : \{p{:}\text{int}\} \Rightarrow \{p{=}\#p\ x, q{=}42\})$
we have

$$\Gamma \vdash f : \{p{:}\text{int}\} \to \{p{:}\text{int}, q{:}\text{int}\}$$
$$\Gamma \vdash f : \{p{:}\text{int}\} \to \{p{:}\text{int}\}$$
$$\Gamma \vdash f : \{p{:}\text{int}, q{:}\text{int}\} \to \{p{:}\text{int}, q{:}\text{int}\}$$
$$\Gamma \vdash f : \{p{:}\text{int}, q{:}\text{int}\} \to \{p{:}\text{int}\}$$

Subtyping – Functions (III)

Example: let $f = (\textbf{fn}\ x : \{p\text{:int}, q\text{:int}\} \Rightarrow \{p{=}(\#p\ x) + (\#q\ x)\})$

we have

$$\begin{aligned}
&\Gamma \vdash f : \{p\text{:int}, q\text{:int}\} \rightarrow \{p\text{:int}\}\\
&\Gamma \nvdash f : \{p\text{:int}\} \rightarrow T\\
&\Gamma \nvdash f : T \rightarrow \{p\text{:int}, q\text{:int}\}
\end{aligned}$$

Subtyping – Top and Bottom

(s-top) $T <: \text{Top}$

- not strictly necessary, but convenient
- corresponds to `Object` found in most OO languages

Does it make sense to have a bottom type Bot?
(see B. Pierce for an answer)

Subtyping – Products and Sums

Products

(s-pair) $$\frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 * T_2 <: T_1' * T_2'}$$

Sums

Exercise

Subtyping – References (I)

Does one of the following make sense?

$$\frac{T <: T'}{T \text{ ref} <: T' \text{ ref}} \qquad\qquad \frac{T' <: T}{T \text{ ref} <: T' \text{ ref}}$$

No

Subtyping – References (II)

(s-ref) $$\frac{T <: T' \qquad T' <: T}{T \text{ ref} <: T' \text{ ref}}$$

- ref needs to be an *invariant*
- a more refined analysis of references is possible (using `Source` – capability to read –, and `Sink` – capability to write)

Example:

$$\{a{:}\text{int}, b{:}\text{bool}\} \text{ ref} <: \{b{:}\text{bool}, a{:}\text{int}\} \text{ ref}$$

Typing – Remarks

Semantics
no change required (we did not change the grammar for expressions)

Properties
Type preservation, progress and type safety hold

Implementation
Type inference is more complicated; good run-time is also tricky due to re-ordering

Down Casts

The rule (sub) permits up-casting. How down-casting?

$$E ::= \ldots \mid (T)\, E$$

Typing rule

$$\frac{\Gamma \vdash E : T'}{\Gamma \vdash (T)\, E : T}$$

- requires dynamic type checking
  (verify type safety of a program at runtime)
- gives flexibility, at the cost of potential run-time errors
- better handled by *parametric polymorphism*, a.k.a. *generics* (for example Java)

Section 11

(Imperative) Objects

Case Study

Motivation

- use our language with subtyping, records and references to model key keatures of OO programming
- encode/approximate concepts into our language
- OO concepts
  - *multiple representations* (object carry their methods)
    (in contrast to abstract data types (ADTs))
  - *encapsulation*
  - *subtyping*
    interface is the set of names and types of its operations
  - *inheritance* share common parts (class and subclasses)
    some languages use *delegations* (e.g. C$^\sharp$), which combine classes and objects
  - *open recursion* (`self` or `this`)

(Simple) Objects

- data structure encapsulating some internal state
- access via methods
- internal state typically a number of mutable instance variables (or fields)
- attention lies on *building*, rather than usage

Reminder

Scope Restriction

$$E ::= \ldots \mid \textbf{let val } x : T = E_1 \textbf{ in } E_2 \textbf{ end}$$

- x is a binder for E_2
- can be seen as syntactic sugar:

$$\textbf{let val } x : T = E_1 \textbf{ in } E_2 \textbf{ end} \equiv (\textbf{fn } x : T \Rightarrow E_2)\ E_1$$

Objects – Example

A Counter Object

$$
\begin{array}{l}
\textbf{let val } c : \{\texttt{get} : \mathsf{unit} \to \mathsf{int},\ \texttt{inc} : \mathsf{unit} \to \mathsf{unit}\} = \\
\quad \textbf{let val } x : \mathsf{int\ ref} = \mathsf{ref}\ 0\ \textbf{in} \\
\quad\quad \{\texttt{get} = (\textbf{fn } y : \mathsf{unit} \Rightarrow !x), \\
\quad\quad\ \ \texttt{inc} = (\textbf{fn } y : \mathsf{unit} \Rightarrow x := !x + 1)\} \\
\quad \textbf{end} \\
\textbf{in} \\
\quad (\#\texttt{inc}\ c)()\ ;\ (\#\texttt{get}\ c)() \\
\textbf{end}
\end{array}
$$

$$\texttt{Counter} = \{\texttt{get} : \mathsf{unit} \to \mathsf{int},\ \texttt{inc} : \mathsf{unit} \to \mathsf{unit}\}$$

Objects – Example

Subtyping I

$$\textbf{let val } c : \{\texttt{get} : \text{unit} \to \text{int}, \ \texttt{inc} : \text{unit} \to \text{unit}, \texttt{reset} : \text{unit} \to \text{unit}\} =$$

$$\textbf{let val } x : \text{int ref} = \text{ref } 0 \textbf{ in}$$

$$\{\texttt{get} = (\textbf{fn } y : \text{unit} \Rightarrow !x),$$

$$\texttt{inc} = (\textbf{fn } y : \text{unit} \Rightarrow x := !x + 1)\}$$

$$\texttt{reset} = (\textbf{fn } y : \text{unit} \Rightarrow x := 0)\}$$

end

in

$$(\#\texttt{inc } c)() \ ; \ (\#\texttt{get } c)()$$

end

$$\texttt{ResCounter} \ = \ \{\texttt{get} : \text{unit} \to \text{int}, \ \texttt{inc} : \text{unit} \to \text{unit}, \texttt{reset} : \text{unit} \to \text{unit}\}$$

Objects – Example

Subtyping II

$$\texttt{ResCounter} <: \texttt{Counter}$$

Objects – Example

Object Generators

$$
\begin{array}{l}
\textbf{let val } \texttt{newCounter} : \text{unit} \to \texttt{Counter} = \\
\quad (\textbf{fn } y : \text{unit} \Rightarrow \\
\qquad \textbf{let val } x : \text{int ref} = \text{ref } 0 \textbf{ in} \\
\qquad\quad \{\texttt{get} = (\textbf{fn } y : \text{unit} \Rightarrow !x), \\
\qquad\quad\; \texttt{inc} = (\textbf{fn } y : \text{unit} \Rightarrow x := !x + 1)\} \\
\qquad \textbf{end}) \\
\textbf{in} \\
\quad (\#\texttt{inc}\ (\texttt{newCounter}()))() \\
\textbf{end}
\end{array}
$$

`newRCounter` defined in similar fashion

Simple Classes

- pull out common features
- ignore complex features
  such as *visibility annotations*, *static fields and methods*, *friend classes* . . .

- most primitive form, a class is a data structure that can
  – be instantiated to yields a fresh object, or – extended to yield another class

Reusing Method Code

$$\begin{aligned}\texttt{Counter} &= \{\texttt{get} : \mathsf{unit} \to \mathsf{int},\ \texttt{inc} : \mathsf{unit} \to \mathsf{unit}\} \\ \texttt{CounterRep} &= \{p : \mathsf{int\ ref}\}\end{aligned}$$

(Simple) Classes

$$
\begin{array}{l}
\textbf{let val}\ \texttt{CounterClass} : \texttt{CounterRep} \to \texttt{Counter} = \\
\quad (\textbf{fn}\ x : \texttt{CounterRep} \Rightarrow \\
\quad\quad \{\texttt{get} = (\textbf{fn}\ y : \textsf{unit} \Rightarrow !(\#p\ x)), \\
\quad\quad\ \ \texttt{inc} = (\textbf{fn}\ y : \textsf{unit} \Rightarrow (\#p\ x) := !(\#p\ x) + 1)\})
\end{array}
$$

$$
\begin{array}{l}
\textbf{let val}\ \texttt{newCounter} : \textsf{unit} \to \texttt{Counter} = \\
\quad (\textbf{fn}\ y : \textsf{unit} \Rightarrow \\
\quad\quad \textbf{let val}\ x : \texttt{CounterRep} = \{p = \textsf{ref}\ 0\}\ \textbf{in} \\
\quad\quad\quad \texttt{CounterClass}\ x \\
\quad\quad \textbf{end})
\end{array}
$$

IMP vs. Java

```
class Counter
  {  protected int p;
     Counter() { this.p=0; }
     int get () { return this.p; }
     void inc () { this.p++ ; }
  };
```

(Simple) Classes

$$
\begin{array}{l}
(\textbf{fn}\ \mathtt{ResCounterClass} : \mathtt{CounterRep} \to \mathtt{ResCounter} \Rightarrow \\
\quad (\textbf{fn}\ x : \mathtt{CounterRep} \Rightarrow \\
\quad\quad \textbf{let val}\ \mathtt{super} : \mathtt{Counter} = \mathtt{CounterClass}\ x\ \textbf{in} \\
\quad\quad\quad \{\mathtt{get} = \#\mathtt{get}\ \mathtt{super}, \\
\quad\quad\quad\ \mathtt{inc} = \#\mathtt{inc}\ \mathtt{super}, \\
\quad\quad\quad\ \mathtt{reset} = (\textbf{fn}\ y : \mathsf{unit} \Rightarrow (\#p\ x) := 0)\} \\
\quad\quad \textbf{end}))
\end{array}
$$

$$
\begin{aligned}
\mathtt{CounterRep} &= \{p : \mathsf{int\ ref}\} \\
\mathtt{Counter} &= \{\mathtt{get} : \mathsf{unit} \to \mathsf{int},\ \mathtt{inc} : \mathsf{unit} \to \mathsf{unit}\} \\
\mathtt{ResCounter} &= \{\mathtt{get} : \mathsf{unit} \to \mathsf{int},\ \mathtt{inc} : \mathsf{unit} \to \mathsf{unit},\ \mathtt{reset} : \mathsf{unit} \to \mathsf{unit}\}
\end{aligned}
$$

Australian National University

IMP vs. Java

```
class ResetCounter
  extends Counter
  {  void reset () {this.p=0;}
  };
```

(Simple) Classes

$$\begin{aligned}\texttt{BuCounter} = \{&\texttt{get}:\mathsf{unit} \to \mathsf{int},\ \texttt{inc}:\mathsf{unit} \to \mathsf{unit},\\ &\texttt{reset}:\mathsf{unit} \to \mathsf{unit},\ \texttt{backup}:\mathsf{unit} \to \mathsf{unit}\}\\ \texttt{BuCounterRep} = \{&p:\mathsf{int\ ref},\ b:\mathsf{int\ ref}\}\end{aligned}$$

$$\begin{array}{l}\textbf{let val}\ \texttt{BuCounterClass} : \texttt{BuCounterRep} \to \texttt{BuCounter} =\\ \quad (\textbf{fn}\ x : \texttt{BuCounterRep} \Rightarrow\\ \qquad \textbf{let val}\ \texttt{super} : \texttt{ResCounter} = \texttt{ResCounterClass}\ x\ \textbf{in}\\ \qquad\quad \{\texttt{get} = \#\texttt{get super},\ \texttt{inc} = \#\texttt{inc super},\\ \qquad\quad\ \texttt{reset} = (\textbf{fn}\ y : \mathsf{unit} \Rightarrow (\#p\ x) := !(\#b\ x))\}\\ \qquad\quad\ \texttt{backup} = (\textbf{fn}\ y : \mathsf{unit} \Rightarrow (\#b\ x) := !(\#p\ x))\}\\ \qquad \textbf{end})\end{array}$$

Section 12

Implementing IMP

Motivation

- started with (a variant) of IMP
- added several features
  (e.g. functions, exceptions, objects, ...)
- no concurrency yet
- no verification
  (you may have seen some bits of Hoare logic)

Implementations of IMP I

- **ML**
  `https://www.cl.cam.ac.uk/teaching/2021/Semantics/L2/`
  P. Sewell
- **C**
  “any compiler”
- **Java**
  `https://www.cl.cam.ac.uk/teaching/2021/Semantics/L1/l1.java`
  M. Parkinson
- **Haskell**
  (several implementations available)

Implementations of IMP II

- **Coq**
  `https://softwarefoundations.cis.upenn.edu/lf-current/Imp.html`
  B. Pierce
- **Isabelle**
  `https://isabelle.in.tum.de` (`src/HOL/IMP`)
  G. Klein and T. Nipkow

Australian National University

Section 13

IMP in Isabelle/HOL

Motivation/Disclaimer

- generic proof assistant
- express mathematical formulas in a formal language
- tools for proving those formulas in a logical calculus
- originally developed at the University of Cambridge and Technische Universität München (now numerous contributions, including Australia)

- this is **neither a course about Isabelle nor a proper introduction to Isabelle**

Isabelle/HOL – Introduction

Isabelle/HOL = Functional Programming + Logic

Isabelle HOL has

- datatypes
- recursive functions
- logical operators
- ...

Isabelle/HOL is a programming language, too

- Higher-order means that functions are values, too

Isabelle/HOL – Terms (Expressions)

- **Functions**
  - application: $f\ E$
    call of function f with parameter E
  - abstraction: $\lambda x.\ E$
    function with parameter x (of some type) and result E ((**fn** $x : T_? \Rightarrow t$))
  - Convention (as always) $f\ E_1\ E_2\ E_3 \equiv ((f\ E_1)\ E_2)\ E_3$

- **Basic syntax** (Isabelle)

$$
\begin{array}{rlll}
t ::= & (t) & & \\
& \mid\ a & & \text{identifier (constant or variable)} \\
& \mid\ t\ t & & \text{function application} \\
& \mid\ \lambda x.\ t & & \text{function abstraction} \\
& \mid\ \ldots & & \text{syntactic sugar}
\end{array}
$$

- **Substitution** notation: $t[u/x]$

Isabelle/HOL – Types I

- **Basic syntax** (Isabelle)

$$
\begin{array}{rll}
\tau ::= & (\tau) & \\
| & \texttt{bool} \mid \texttt{int} \mid \texttt{string} \mid \ldots & \text{base types} \\
| & 'a \mid 'b \mid \ldots & \text{type variables} \\
| & \tau \Rightarrow \tau & \text{functions} \\
| & \tau \times \tau & \text{pairs} \\
| & \tau\ \texttt{list} & \text{lists} \\
| & \tau\ \texttt{set} & \text{sets} \\
| & \ldots & \text{user-defined types}
\end{array}
$$

  Convention: $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \;\equiv\; \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$

- **Terms must be well-typed**; in particular

$$
\frac{t :: \tau_1 \Rightarrow \tau_2 \qquad u :: \tau_1}{t\ u :: \tau_2}
$$

Isabelle/HOL – Types II

Type inference

- automatic
- function overloading possible
  can prevent type inference
- **type annotation** $t :: \tau$ (for example $f\ (x :: \mathtt{int})$)

Currying

- curried vs. tupled

$$f \quad \tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \qquad \text{vs} \qquad f \quad \tau_1 \times \tau_2 \Rightarrow \tau_3$$

- use curried versions if possible
- advantage: allow *partial function application*

$$f\ a_1 \qquad \text{where } a_1 :: \tau_1$$

Isabelle (Cheatsheet I)

Isabelle module = Theory (File structure)

Syntax: **theory** $MyTh$
imports $Th_1, \ldots, Th_n$
begin
(definitions, lemmas, theorems, proofs, . . .)*
end

$MyTh$: name of theory. Must live in file $MyTh$`.thy`
Th_i: names of imported theories; imports are transitive

Usually: **imports** `Main`

IMP – Syntax (recap)

Booleans	$b \in \mathbb{B} = \{\texttt{true}, \texttt{false}\}$
Integers (Values)	$n \in \mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$
Locations	$l \in \mathbb{L} = \{l, l_0, l_1, l_2, \dots\}$
Operations	$op ::= + \mid \geq$

Expressions

$$
\begin{aligned}
E ::= {} & n \mid b \mid E\ op\ E \mid \\
& l := E \mid !l \mid \\
& \textbf{if } E \textbf{ then } E \textbf{ else } E \mid \\
& \textbf{skip} \mid E \mathrel{;} E \mid \\
& \textbf{while} E \textbf{ do } E
\end{aligned}
$$

IMP – Syntax (aexp and bexp)

Booleans	$b \in \mathbb{B}$
Integers (Values)	$n \in \mathbb{Z}$
Locations	$l \in \mathbb{L} \; = \{l, l_0, l_1, l_2, \dots\}$
Operations	$aop ::= +$

Expressions

$$\texttt{aexp} ::= n \mid !l \mid \texttt{aexp}\; aop\; \texttt{aexp}$$

$$\texttt{bexp} ::= b \mid \texttt{bexp} \wedge \texttt{bexp} \mid \texttt{aexp} \geq \texttt{aexp}$$

$$\begin{aligned}\texttt{com} ::= {} & \cancel{n} \mid \cancel{b} \mid \cancel{E\ op\ E} \mid \\ & l := \texttt{aexp} \mid \cancel{!l} \mid \\ & \texttt{IF bexp THEN com ELSE com} \mid \\ & \texttt{SKIP} \mid \texttt{com ;; com} \mid \\ & \texttt{WHILE bexp DO com}\end{aligned}$$

IMP – Syntax (Isabelle)

Booleans	`bool`
Integers (Values)	`int`
Locations	`string`

Expressions

datatype `aexp` ::= `N int` | `L loc` | `Plus aexp aexp`

datatype `bexp` ::= `B bool` | `Geq aexp aexp`

datatype `com` ::= `Assign loc aexp` |
`If bexp com com` |
`SKIP` | `Seq com com` |
`WHile bexp com`

IMP – Syntax (Isabelle)

LINK: /src/HOL/IMP

Isabelle (Cheatsheet II)

type_synonym	specify synonym for a type
datatype	define recursive (polymorphic) types
fun	define (simple, recursive) function
	(tries to prove exhaustiveness, non-overlappedness, and termination)
value	evaluate a term

Small-step semantics

- a configuration $\langle E\,,\,s\rangle$ can perform a step if there is a derivation tree
- vice versa the set of all transitions can be defined inductively
- it is an infinite set

Australian National University

IMP Semantics

(deref) $\cancel{\langle !l \, , \, s \rangle \longrightarrow \langle n \, , \, s \rangle \qquad \text{if } l \in \text{dom}(s) \text{ and } s(l) = n}$

(assign1) $\langle l := n \, , \, s \rangle \longrightarrow \langle \textbf{skip} \, , \, s + \{l \mapsto n\} \rangle \qquad \text{if } l \in \text{dom}(s)$

(assign2) $\dfrac{\cancel{\langle E \, , \, s \rangle \longrightarrow \langle E' \, , \, s' \rangle}}{\cancel{\langle l := E \, , \, s \rangle \longrightarrow \langle l := E' \, , \, s' \rangle}}$

(seq1) $\langle \textbf{skip}; E_2 \, , \, s \rangle \longrightarrow \langle E_2 \, , \, s \rangle$

(seq2) $\dfrac{\langle E_1 \, , \, s \rangle \longrightarrow \langle E_1' \, , \, s' \rangle}{\langle E_1; E_2 \, , \, s \rangle \longrightarrow \langle E_1; E_2 \, , \, s \rangle}$

(if1) $\langle \textbf{if } \texttt{true} \textbf{ then } E_2 \textbf{ else } E_3 \, , \, s \rangle \longrightarrow \langle E_2 \, , \, s \rangle$

(if2) $\langle \textbf{if } \texttt{false} \textbf{ then } E_2 \textbf{ else } E_3 \, , \, s \rangle \longrightarrow \langle E_3 \, , \, s \rangle$

(if3) $\dfrac{\cancel{\langle E_1 \, , \, s \rangle \longrightarrow \langle E_1' \, , \, s' \rangle}}{\cancel{\langle \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 \, , \, s \rangle \longrightarrow \langle \textbf{if } E_1' \textbf{ then } E_2 \textbf{ else } E_3 \, , \, s' \rangle}}$

(while) $\langle \textbf{while} E_1 \textbf{ do } E_2 \, , \, s \rangle \longrightarrow \langle \textbf{if } E_1 \textbf{ then } (E_2; \textbf{while} E_1 \textbf{ do } E_2) \textbf{ then skip} \, , \, s \rangle$

IMP Semantics

(assign1)	$\langle l := n \,,\, s\rangle \longrightarrow \langle \textbf{skip} \,,\, s + \{l \mapsto n\}\rangle \qquad \text{if } l \in \mathrm{dom}(s)$
(seq1)	$\langle \textbf{skip}; E_2 \,,\, s\rangle \longrightarrow \langle E_2 \,,\, s\rangle$
(seq2)	$\dfrac{\langle E_1 \,,\, s\rangle \longrightarrow \langle E_1' \,,\, s'\rangle}{\langle E_1; E_2 \,,\, s\rangle \longrightarrow \langle E_1; E_2 \,,\, s\rangle}$
(if1)	$\langle \textbf{if } \texttt{true} \textbf{ then } E_2 \textbf{ else } E_3 \,,\, s\rangle \longrightarrow \langle E_2 \,,\, s\rangle$
(if2)	$\langle \textbf{if } \texttt{false} \textbf{ then } E_2 \textbf{ else } E_3 \,,\, s\rangle \longrightarrow \langle E_3 \,,\, s\rangle$
(while)	$\langle \textbf{while} E_1 \textbf{ do } E_2 \,,\, s\rangle \longrightarrow \langle \textbf{if } E_1 \textbf{ then } (E_2; \textbf{while} E_1 \textbf{ do } E_2) \textbf{ then skip} \,,\, s\rangle$

IMP Semantics (Isabelle)

LINK: /src/HOL/IMP/Small_Step

IMP – Examples

- If $E = (l_2 := 0; \textbf{while} \, !l_1 \geq 1 \textbf{ do } (l_2 := !l_2 + !l_1; l_1 := !l_1 + -1))$
  $s = \{l_1 \mapsto 3, l_2 \mapsto 0\}$
  then $\langle E \, , \, s\rangle \longrightarrow^* \; ?$
- determinacy
- progress

Isabelle (Cheatsheet III)

inductive	defines (smallest) inductive set
print_theorems	shows generated theorems
find_theorems	searches available theorems by name and/or pattern
apply `(<rule/tactic>)`	applies rule to proof goal (simp, auto, blast, rule <name>)

Big-step semantics
(in Isabelle/HOL)

Another View: Big-step Semantics

- we have seen a **small-step semantics**

$$\langle E \,,\, s\rangle \longrightarrow \langle E' \,,\, s'\rangle$$

- alternatively, we could have looked at a **big-step semantics**

$$\langle E \,,\, s\rangle \Downarrow \langle E' \,,\, s'\rangle$$

For example

$$\frac{}{\langle n \,,\, s\rangle \Downarrow \langle n \,,\, s\rangle} \qquad \frac{\langle E_1 \,,\, s\rangle \Downarrow \langle n_1 \,,\, s'\rangle \quad \langle E_2 \,,\, s'\rangle \Downarrow \langle n_2 \,,\, s''\rangle}{\langle E_1 + E_2 \,,\, s\rangle \Downarrow \langle n \,,\, s''\rangle}\ (n = n_1{+}n_2)$$

- no major difference for sequential programs
- small-step much better for modelling concurrency

Australian National University

Final State

- Isabelle's version of IMP has only one value: `SKIP`
- big-step semantics can be seen as relation

$$\langle E \, , \, s\rangle \Longrightarrow s'$$

Semantics

(Skip)

$$\langle \texttt{SKIP} , s\rangle \Longrightarrow s$$

(Assign)

$$\langle l := a , s\rangle \Longrightarrow s + \{l \mapsto \textsf{aval}\ a\ s)$$

(Seq)

$$\frac{\langle E_1 , s\rangle \Longrightarrow s' \qquad \langle E_2 , s'\rangle \Longrightarrow s''}{\langle E_1 ; E_2 , s\rangle \Longrightarrow s''}$$

(IfT)

$$\frac{\textsf{bval}\ b\ s = \texttt{true} \qquad \langle E_1 , s\rangle \Longrightarrow s'}{\langle \textbf{if}\ b\ \textbf{then}\ E_1\ \textbf{else}\ E_2 , s\rangle \Longrightarrow s'}$$

(IfF)

$$\frac{\textsf{bval}\ b\ s = \texttt{false} \qquad \langle E_2 , s\rangle \Longrightarrow s'}{\langle \textbf{if}\ b\ \textbf{then}\ E_1\ \textbf{else}\ E_2 , s\rangle \Longrightarrow s'}$$

(WhileF)

$$\frac{\textsf{bval}\ b\ s = \texttt{false}}{\langle \textbf{while}\ b\ \textbf{do}\ E , s\rangle \Longrightarrow s}$$

(WhileT)

$$\frac{\textsf{bval}\ b\ s = \texttt{true} \qquad \langle E , s\rangle \Longrightarrow s' \qquad \langle \textbf{while}\ b\ \textbf{do}\ E , s'\rangle \Longrightarrow s''}{\langle \textbf{while}\ b\ \textbf{do}\ E , s\rangle \Longrightarrow s''}$$

IMP Semantics (Isabelle)

LINK: /src/HOL/IMP/Big_Step

- inversion rules
- induction set up
- see Nipkow/Klein for more details and explanation

Are big and small-step semantics equivalent?

Isabelle (Cheatsheet IV)

Proof Styles/Proof 'Tactics'

apply-style	apply rules (backwards)
ISAR	human readable proofs
slegdehammer	the 'secret' weapon
	incorporating automated theorem provers

From Big to Small

Theorem

If $cs \Rightarrow s'$ *then* $cs \longrightarrow^* \langle \mathit{SKIP}, s' \rangle$.

Proof by rule induction (on $cs \Rightarrow s'$).

In two cases a lemma is needed.

Lemma

If $\langle E , s \rangle \longrightarrow^* \langle E' , s' \rangle$ *then* $\langle E ; E_2 , s \rangle \longrightarrow^* \langle E' ; E_2 , s' \rangle$.

Proof by rule induction.
(generalisation of (seq2))

From Small to Big

Theorem

If $cs \longrightarrow^* \langle \mathit{SKIP}\,,\ s' \rangle$ *then* $cs \Rightarrow s'$*.*

Proof by rule induction (on $cs \longrightarrow^* \langle \texttt{SKIP}\,,\ s' \rangle$).

The induction step needs the following (interesting) lemma.

Lemma

If $cs \longrightarrow cs'$ *and* $cs' \Rightarrow s$ *then* $cs \Rightarrow s$*.*

Proof by rule induction on $cs \longrightarrow cs'$.

Equivalence

Corollary

$cs \longrightarrow^* \langle \mathit{SKIP}, s' \rangle$ *if and only if* $cs \Rightarrow s'$.

LINK: /src/HOL/Small_Step

But are they really equivalent?

- What about premature termination?
- What about (non) termination?

Lemma

1. *final* $\langle E\,,\, s\rangle$ *if and only if* $E = \mathit{SKIP}$.
2. $\exists s.\ cs \Rightarrow s$ *if and only if* $\exists cs'.\ cs \longrightarrow^{*} cs' \wedge$ *final* cs'.

where final $cs \equiv (\neg\exists cs'.\ cs \rightarrow cs')$

Proof.

1. induction and rule inversion
2. $$\begin{aligned} (\exists s.\ cs \Rightarrow s) &\Leftrightarrow \exists s.\ cs \longrightarrow^{*} \langle \mathtt{SKIP}\,,\, s\rangle && \text{(by big = small)} \\ &\Leftrightarrow \exists cs'.\ cs \longrightarrow^{*} cs' \wedge \textit{final}\ cs' && \text{(by final = SKIP)} \end{aligned}$$

$\square$

Typing

(almost straight-forward)

LINK: /src/HOL/Types

```
inductive btyping :: "tyenv ⇒ bexp ⇒ bool"(infix "⊢"50)
where
B_ty: "Γ ⊢ Bc v" |
Not_ty: "Γ ⊢ b ⟹ Γ ⊢ Not b" |
And_ty: "Γ ⊢ b1 ⟹ Γ ⊢ b2 ⟹ Γ ⊢ And b1 b2" |
Less_ty: "Γ ⊢ a1 : τ ⟹ Γ ⊢ a2 : τ ⟹ Γ ⊢ Less a1 a2"

inductive ctyping ::"tyenv ⇒ com ⇒ bool"(infix " ⊢ "50)
where
Skip_ty: "Γ ⊢ SKIP" |
Assign_ty: "Γ ⊢ a : Γ(x) ⟹ Γ ⊢ x ::= a" |
Seq_ty: "Γ ⊢ c1 ⟹ Γ ⊢ c2 ⟹ Γ ⊢ c1 ;; c2" |
If_ty: "Γ ⊢ b ⟹ Γ ⊢ c1 ⟹ Γ ⊢ c2 ⟹ Γ ⊢ IF b THEN c1 ELSE c2" |
While_ty: "Γ ⊢ b ⟹ Γ ⊢ c ⟹ Γ ⊢ WHILE b DO c"
```

Section 14

Semantic Equivalence

Operational Semantics (Reminder)

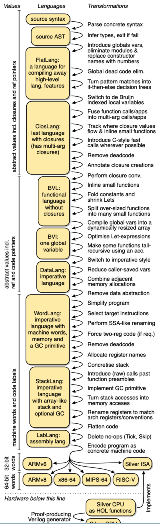
- describe how to evaluate programs
- a valid program is interpreted as sequences of steps
- small-step semantics
  - individual steps of a computation
  - more rules (compared to big-step)
  - allows to reason about non-terminating programs, concurrency, . . .
- big-step semantics
  - overall results of the executions
    ‘divide-and-conquer manner’
  - can be seen as relations
  - fewer rules, simpler proofs
  - no non-terminating behaviour
- allow non-determinism

Motivation

When are two programs considered the ‘same’

- compiler construction
- program optimisation
- refinement
- …

CakeML



Equivalence: Intuition I

$$l := !l + 2 \quad \overset{?}{\simeq} \quad l := !l + (1 + 1) \quad \overset{?}{\simeq} \quad l := !l + 1 \,;\, l := !l + 1$$

- are these expressions the same
- in what sense
  - different abstract syntax trees
  - different reduction sequences
- in any (sequential) program one could replace one by the other without affecting the result

Note: mathematicians often take these equivalences for granted

Equivalence: Intuition II

$l := 0 \,;\, 4 \quad \overset{?}{\simeq} \quad l := 1 \,;\, 3 + !l$

- produce same result (for all stores)
- cannot be replaced in an arbitrary context C

For example, let $C[\_] = \_ + !l$

$$C[l := 0 \,;\, 4] = (l := 0 \,;\, 4) + !l$$

$$\not\simeq$$

$$C[l := 1 \,;\, 3 + !l] = (l := 1 \,;\, 3 + !l) + !l$$

On the other hand $(l := !l + 2) \simeq (l := !l + 1 \,;\, l := !l + 1)$

Equivalence: Intuition III

From particular expressions to general laws

- $E_1 \mathbin{;} (E_2 \mathbin{;} E_3) \overset{?}{\simeq} (E_1 \mathbin{;} E_2) \mathbin{;} E_3$
- $(\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3) \mathbin{;} E \overset{?}{\simeq} \textbf{if } E_1 \textbf{ then } E_2 \mathbin{;} E \textbf{ else } E_3 \mathbin{;} E$
- $E \mathbin{;} (\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3) \overset{?}{\simeq} \textbf{if } E_1 \textbf{ then } E \mathbin{;} E_2 \textbf{ else } E \mathbin{;} E_3$
- $E \mathbin{;} (\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3) \overset{?}{\simeq} \textbf{if } E \mathbin{;} E_1 \textbf{ then } E_2 \textbf{ else } E_3$

Exercise

$$\textbf{let val } x : \text{int ref} = \text{ref } 0 \textbf{ in } (\textbf{fn } y : \text{int} \Rightarrow (x := !x + y)\ ; !x) \textbf{ end}$$

$$\overset{?}{\simeq}$$

$$\textbf{let val } x : \text{int ref} = \text{ref } 0 \textbf{ in } (\textbf{fn } y : \text{int} \Rightarrow (x := !x - y)\ ; (0 - !x)) \textbf{ end}$$

Exercise II

Extend our language with location equality

$$op := \ \dots \mid =$$

(op =) $$\frac{\Gamma \vdash E_1 : T \text{ ref} \qquad \Gamma \vdash E_2 : T \text{ ref}}{\Gamma \vdash E_1 = E_2 : \text{bool}}$$

(op=1) $\langle l = l' \, , \, s \rangle \longrightarrow \langle b \, , \, s \rangle$ if $b = (l = l')$

(op=2) $\dots$

Exercise II

$$f \overset{?}{\simeq} g$$

for

$$\begin{aligned} f = {} & \textbf{let val } x : \text{int ref} = \text{ref } 0 \textbf{ in} \\ & \textbf{let val } y : \text{int ref} = \text{ref } 0 \textbf{ in} \\ & (\textbf{fn } z : \text{int ref} \Rightarrow \textbf{if } z = x \textbf{ then } y \textbf{ else } x) \\ & \ \textbf{end end} \end{aligned}$$

and

$$\begin{aligned} g = {} & \textbf{let val } x : \text{int ref} = \text{ref } 0 \textbf{ in} \\ & \textbf{let val } y : \text{int ref} = \text{ref } 0 \textbf{ in} \\ & (\textbf{fn } z : \text{int ref} \Rightarrow \textbf{if } z = y \textbf{ then } y \textbf{ else } x) \\ & \ \textbf{end end} \end{aligned}$$

Exercise II (cont'd)

$$f \overset{?}{\simeq} g \qquad \textbf{NO}$$

Consider $C[\_] = t\_$ with

$$t = (\textbf{fn}\ h : (\text{int ref} \rightarrow \text{int ref}) \Rightarrow$$
$$\textbf{let val}\ z : \text{int ref} = \text{ref}\ 0\ \textbf{in}\ h\ (h\ z) = h\ z\ \textbf{end})$$

$$\langle t\ f\ ,\ s\rangle \longrightarrow^* \ ?$$
$$\langle t\ g\ ,\ s\rangle \longrightarrow^* \ ?$$

A 'good' notion of semantic equivalence

We might

- understand what a program *is*
- prove that some particular expressions to be equivalent
  (e.g. efficient algorithm vs. clear specification)
- prove the soundness of general laws for equational reasoning about programs
- prove some compiler optimisations are sound (see CakeML or CertiCos)
- understand the differences between languages

What does ‘good’ mean?

1. programs that result in observably-different values (for some store) must not be equivalent

$$
\begin{aligned}
&(\exists s, s_1, s_2, v_1, v_2. \\
&\quad \langle E_1 \,,\, s \rangle \longrightarrow^* \langle v_1 \,,\, s_1 \rangle \wedge \\
&\quad \langle E_2 \,,\, s \rangle \longrightarrow^* \langle v_2 \,,\, s_2 \rangle \wedge \\
&\quad v_1 \neq v_2) \\
&\Rightarrow E_1 \not\simeq E_2
\end{aligned}
$$

2. programs that terminate must not be equivalent to programs that do not terminate

What does ‘good’ mean?

3. $\simeq$ must be an equivalence relation, i.e.

reflexivity	$E \simeq E$
symmetry	$E_1 \simeq E_2 \Rightarrow E_2 \simeq E_1$
transitivity	$E_1 \simeq E_2 \wedge E_2 \simeq E_3 \Rightarrow E_1 \simeq E_3$

4. $\simeq$ must be a congruence, i.e,

   if $E_1 \simeq E_2$ then for any context C we must have $C[E_1] \simeq C[E_2]$

   (for example, $(E_1 \simeq E_2) \Rightarrow (E_1 \ ; E \simeq E_2 \ ; E)$)

5. $\simeq$ should relate as many programs as possible

– an equivalence relation that is a congruence is sometimes called *congruence relation*
– this semantic equivalence, is called observable operational or contextual equivalence
– congruence proofs are often tedious, and incredible hard when it comes to recursion

Semantic Equivalence for (simple) Typed IMP

Definition

$E_1 \simeq_\Gamma^T E_2$ iff for all stores s with $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$ we have

$$\Gamma \vdash E_1 : T \quad \text{and} \quad \Gamma \vdash E_2 : T \,,$$

and either

(i) $\langle E_1 \,,\, s\rangle \longrightarrow^\omega$ and $\langle E_2 \,,\, s\rangle \longrightarrow^\omega$, or

(ii) for some v, s' we have $\langle E_1 \,,\, s\rangle \longrightarrow^* \langle v \,,\, s'\rangle$ and $\langle E_2 \,,\, s\rangle \longrightarrow^* \langle v \,,\, s'\rangle$.

$\longrightarrow^\omega$: infinite sequence

$\longrightarrow^*$: finite sequence (reflexive transitive closure)

Justification

Part (ii) requires same value v and same store s'. If a program generates different stores, we can distinguish them using contexts:

- If $T =$ unit then $C[\_] = \_\ ; !l$
- If $T =$ bool then $C[\_] =$ **if** $\_$ **then** $!l$ **else** $!l$
- If $T =$ int then $C[\_] = (l_1 := \_\ ; !l)$

Equivalence Relation

Theorem

The relation $\simeq_{\Gamma}^{T}$ is an equivalence relation.

Proof.

trivial $\square$

Congruence for (simple) Typed IMP

contexts are:

$$
\begin{aligned}
C[\_] \quad ::= \; & \_ \; op \; E_2 \mid E_1 \; op \; \_ \mid \\
& \textbf{if } \_ \textbf{ then } E_2 \textbf{ else } E_3 \mid \\
& \textbf{if } E_1 \textbf{ then } \_ \textbf{ else } E_3 \mid \\
& \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } \_ \mid \\
& l := \_ \mid \\
& \_ \; ; E_2 \mid E_1 \; ; \; \_ \\
& \textbf{while } \_ \textbf{ do } E_2 \mid \textbf{while } E_1 \textbf{ do } \_
\end{aligned}
$$

Definition

The relation $\simeq_\Gamma^T$ has the *congruence property* if, for all E_1 and E_2, whenever $E_1 \simeq_\Gamma^T E_2$ we have for all C and T', if $\Gamma \vdash C[E_1] : T'$ and $\Gamma \vdash C[E_2] : T'$ then

$$C[E_1] \simeq_\Gamma^{T'} C[E_2]$$

Congruence Property

Theorem (Congruence for (simple) typed IMP)

The relation $\simeq^T_\Gamma$ has the congruence property.

Proof.

By case distinction, considering all contexts C. $\square$

For each context C (and arbitrary expression E and store s) consider the possible reduction sequence

$$\langle C[E]\,,\,s\rangle \longrightarrow \langle E_1\,,\,s_1\rangle \longrightarrow \langle E_2\,,\,s_2\rangle \longrightarrow \ldots$$

and deduce the behaviour of E:

$$\langle E\,,\,s\rangle \longrightarrow \langle \hat{E}_1\,,\,s_1\rangle \longrightarrow \ldots$$

Use $E \simeq^T_\Gamma E'$ find a similar reduction sequence of E' and use the reduction rules to construct a sequence of $C[E']$.

Proof of Congruence Property

Case $C = (l := \_)$

Suppose $E \simeq_\Gamma^T E'$, $\Gamma \vdash l := E : T'$ and $\Gamma \vdash l := E' : T'$.
By examination of the typing rule, we have $T = \mathrm{int}$ and $T' = \mathrm{unit}$.
To show $(l := E) \simeq_\Gamma^{T'} (l := E')$ we have to show that for all stores s if dom$(\Gamma) \subseteq$ dom(s) then

- $\Gamma \vdash l := E : T'$, (obvious)
- $\Gamma \vdash l := E' : T'$,(obvious)
- and either
  (i) $\langle l := E \,,\, s \rangle \longrightarrow^\omega$ and $\langle l := E' \,,\, s \rangle \longrightarrow^\omega$
  (ii) for some v, s' we have $\langle l := E \,,\, s \rangle \longrightarrow^* \langle v \,,\, s' \rangle$ and $\langle l := E' \,,\, s \rangle \longrightarrow^* \langle v \,,\, s' \rangle$.

Proof of Congruence Property

Subcase $\langle l := E\ ,\ s \rangle \longrightarrow^{\omega}$

That is

$$\langle l := E\ ,\ s \rangle \longrightarrow \langle E_1\ ,\ s_1 \rangle \longrightarrow \langle E_2\ ,\ s_2 \rangle \longrightarrow \ldots$$

All these must be instances of Rule (assign2), with

$$\langle E\ ,\ s \rangle \longrightarrow \langle \hat{E}_1\ ,\ s_1 \rangle \longrightarrow \langle \hat{E}_2\ ,\ s_2 \rangle \longrightarrow \ldots$$

and $E_1 = (l := \hat{E}_1)$, $E_2 = (l := \hat{E}_2)$, ...
By $E \simeq^T_\Gamma E'$ there is an infinite reduction sequence of $\langle E'\ ,\ s \rangle$.
Using Rule (assign2) there is an infinite reduction sequence of $\langle l := E'\ ,\ s \rangle$.

We made the proof simple by staying in a deterministic language with unique derivation trees.

Proof of Congruence Property

Subcase $\neg(\langle l := E\,,\, s\rangle \longrightarrow^{\omega})$

That is

$$\langle l := E\,,\, s\rangle \longrightarrow \langle E_1\,,\, s_1\rangle \longrightarrow \langle E_2\,,\, s_2\rangle \longrightarrow \ldots \longrightarrow \langle E_k\,,\, s_k\rangle \not\longrightarrow$$

All these must be instances of Rule (assign2), except the last step which is an instance of (assign1)

$$\langle E\,,\, s\rangle \longrightarrow \langle \hat{E}_1\,,\, s_1\rangle \longrightarrow \langle \hat{E}_2\,,\, s_2\rangle \longrightarrow \ldots \longrightarrow \langle \hat{E}_{k-1}\,,\, s_{k-1}\rangle$$

and $E_1 = (l := \hat{E}_1)$, $E_2 = (l := \hat{E}_2)$, . . . , $E_{k-1} = (l := \hat{E}_{k-1})$ and $\hat{E}_{k-1} = n$, $E_k =$ **skip** and $s_k = s_{k-1} + \{l \mapsto n\}$, for some n.

Proof of Congruence Property

Subcase $\neg(\langle l := E\,,\, s\rangle \longrightarrow^{\omega})$ **(cont'd)**

Hence there is some n and s_{k-1} such that

$$\langle E\,,\, s\rangle \longrightarrow^{*} \langle n\,,\, s_{k-1}\rangle \quad \text{and} \quad \langle l := E\,,\, s\rangle \longrightarrow \langle \textbf{skip}\,,\, s_{k-1} + \{l \mapsto n\}\rangle\,.$$

By $E \simeq^{T}_{\Gamma} E'$ we have $\langle E'\,,\, s\rangle \longrightarrow^{*} \langle n\,,\, s_{k-1}\rangle$.

Using Rules (assign2) and (assign1)

$$\langle l := E'\,,\, s\rangle \longrightarrow^{*} \langle l := n\,,\, s_{k-1}\rangle \to \langle \textbf{skip}\,,\, s_{k-1} + \{l \mapsto n\}\rangle\,.$$

Congruence Proofs

Congruence proofs are

- tedious
- long
- mostly boring (up to the point where they brake)
- error prone
- recursion is often the killer case

There are dozens of different semantic equivalences
(and each requires a proof)

Back to Examples

- $1 + 1 \simeq_\Gamma^{\text{int}} 2$ for any Γ
- $(l := 0 \,;\, 4) \not\simeq_\Gamma^{\text{int}} (l := 1 \,;\, 3 + !l)$ for any Γ
- $(l :=!l + 1) \,;\, (l :=!l + 1) \simeq_\Gamma^{\text{unit}} (l :=!l + 2)$ for any Γ including l : intref

General Laws

Conjecture

$E_1 \; ; (E_2 \; ; E_3) \simeq_\Gamma^T (E_1 \; ; E_2) \; ; E_3$
for any Γ, T , E_1, E_2 *and* E_3 *such that* $\Gamma \vdash E_1$: *unit,* $\Gamma \vdash E_2$: *unit and* $\Gamma \vdash E_3 : T$.

Conjecture

$((\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3) \; ; E) \simeq_\Gamma^T (\textbf{if } E_1 \textbf{ then } E_2 \; ; E \textbf{ else } E_3 \; ; E)$
for any Γ, T, E, E_1, E_2 *and* E_3 *such that* $\Gamma \vdash E_1$: *bool,* $\Gamma \vdash E_2$: *unit,* $\Gamma \vdash E_3$: *unit, and* $\Gamma \vdash E : T$.

Conjecture

$(E \; ; (\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3)) \not\simeq_\Gamma^T (\textbf{if } E_1 \textbf{ then } E \; ; E_2 \textbf{ else } E \; ; E_3)$

General Laws

Suppose $\Gamma \vdash E_1 : \mathsf{unit}$ and $\Gamma \vdash E_2 : \mathsf{unit}$.
When is $E_1 \ ; E_2 \simeq_{\Gamma}^{\mathsf{unit}} E_2 \ ; E_1$?

A Philosophical Question

What is a typed expression $\Gamma \vdash E : T$?

for example $l : \text{intref} \vdash$ **if** $!l \geq 0$ **then skip else** $($**skip** $; l := 0) : \text{unit}$.

1. a list of tokens (after parsing) `[IF, DEREF, LOC "l", GTEQ, ...]`
2. an abstract syntax tree
3. the function taking store s to the reduction sequence

$$\langle E\,,\,s\rangle \longrightarrow \langle E_1\,,\,s_1\rangle \longrightarrow \langle E_2\,,\,s_2\rangle \longrightarrow \ldots$$

4. the equivalence class $\{E' \mid E \simeq^T_\Gamma E'\}$
5. the partial function $[\![E]\!]_\Gamma$ that takes any store s with $\text{dom}(s) = \text{dom}(\Gamma)$ and either is undefined if $\langle E\,,\,s\rangle \longrightarrow^\omega$, or is $\langle v\,,\,s'\rangle$, if $\langle E\,,\,s\rangle \longrightarrow^* \langle v\,,\,s'\rangle$

Section 15

Denotational Semantics

Operational Semantics (Reminder)

- describe how to evaluate programs
- a valid program is interpreted as sequences of steps
- small-step semantics
  - individual steps of a computation
  - more rules (compared to big-step)
  - allows to reason about non-terminating programs, concurrency, . . .
- big-step semantics
  - overall results of the executions
    ‘divide-and-conquer manner’
  - can be seen as relations
  - fewer rules, simpler proofs
  - no non-terminating behaviour
- allow non-determinism

Operational vs Denotational

An *operational semantics* is like an interpreter

$$\langle E \,,\, s\rangle \longrightarrow \langle E' \,,\, s'\rangle \qquad \text{and} \qquad \langle E \,,\, s\rangle \Downarrow \langle v \,,\, s'\rangle$$

A denotational semantics is like a compiler.
A *denotational semantics* defines what a program means as a (partial) function:

$$\mathcal{C}[\![\texttt{com}]\!] \in \text{Store} \rightharpoonup \text{Store}$$

Allows the use of ‘standard’ mathematics

Big Picture

$$
\begin{tikzcd}
 & E \arrow[dl, "\text{op. sem}"'] \arrow[d, "{[\![\text{-}]\!]_{\simeq^T_\Gamma}}"] \arrow[dr, "\text{denot. sem}"] & \\
\text{NForm} \arrow[r, leftrightarrow] & E/\simeq^T_\Gamma \arrow[r] & \text{Semantics}
\end{tikzcd}
$$

IMP – Syntax (aexp and bexp)

Booleans	$b \in \mathbb{B}$
Integers (Values)	$n \in \mathbb{Z}$
Locations	$l \in \mathbb{L} \ = \{l, l_0, l_1, l_2, \dots\}$
Operations	$aop ::= +$

Expressions

$$\begin{aligned}
\text{aexp} &::= n \mid !l \mid \text{aexp}\ aop\ \text{aexp} \\
\text{bexp} &::= b \mid \text{bexp} \wedge \text{bexp} \mid \text{aexp} \geq \text{aexp} \\
\text{com} &::= l := \text{aexp} \mid \\
&\quad\ \ \textbf{if}\ \text{bexp}\ \textbf{then}\ \text{com}\ \textbf{else}\ \text{com} \mid \\
&\quad\ \ \textbf{skip} \mid \text{com}\ ;\ \text{com} \mid \\
&\quad\ \ \textbf{while}\ \text{bexp}\ \textbf{do}\ \text{com}
\end{aligned}$$

Semantic Domains

$\mathcal{C}[\![c]\!] \in \text{Store} \rightharpoonup \text{Store}$ $\qquad$ $\mathcal{C}[\![\_]\!]\_ : \text{com} \to \text{Store} \rightharpoonup \text{Store}$

$\mathcal{A}[\![a]\!] \in \text{Store} \rightharpoonup \text{int}$ $\qquad$ $\mathcal{A}[\![\_]\!]\_ : \text{aexp} \to \text{Store} \rightharpoonup \text{int}$

$\mathcal{B}[\![b]\!] \in \text{Store} \rightharpoonup \text{bool}$ $\qquad$ $\mathcal{B}[\![\_]\!]\_ : \text{bexp} \to \text{Store} \rightharpoonup \text{bool}$

Convention: (Partial) Functions are defined point-wise.
$\mathcal{C}[\![\_]\!]$ is the denotation function.

Partial Functions

Remember that partial functions can be represented as sets.

- $\mathcal{C}[\![c]\!]$ can be described as a set
- the equation $\mathcal{C}[\![c]\!] = S$,
  for a set S gives the definition for command c
- $\mathcal{C}[\![c]\!](s)$ is a store

Denotational Semantics for IMP

Arithmetic Expressions

$$\mathcal{A}[\![\underline{n}]\!] = \{(s, n)\}$$

$$\mathcal{A}[\![!l]\!] = \{(s, s(l)) \mid l \in \mathsf{dom}(s)\}$$

$$\mathcal{A}[\![a_1 \,\underline{+}\, a_2]\!] = \{(s, n) \mid (s, n_1) \in \mathcal{A}[\![a_1]\!] \wedge (s, n_2) \in \mathcal{A}[\![a_2]\!] \wedge n = n_1 + n_2\}$$

$\underline{n}$ is syntactical, n semantical value.

Australian National University

Denotational Semantics for IMP

Boolean Expressions

$$\mathcal{B}[\![\underline{\texttt{true}}]\!] = \{(s, \texttt{true})\}$$

$$\mathcal{B}[\![\underline{\texttt{false}}]\!] = \{(s, \texttt{false})\}$$

$$\mathcal{B}[\![b_1 \mathbin{\underline{\land}} b_2]\!] = \{(s, b) \mid (s, b') \in \mathcal{B}[\![b_1]\!] \land (s, b'') \in \mathcal{B}[\![b_2]\!] \land (b = b' \land b'')\}$$

$$\begin{aligned}\mathcal{B}[\![a_1 \mathbin{\underline{\geq}} a_2]\!] = {} & \{(s, \texttt{true}) \mid (s, n_1) \in \mathcal{A}[\![a_1]\!] \land (s, n_2) \in \mathcal{A}[\![a_2]\!] \land n_1 \geq n_2\} \cup \\ & \{(s, \texttt{false}) \mid (s, n_1) \in \mathcal{A}[\![a_1]\!] \land (s, n_2) \in \mathcal{A}[\![a_2]\!] \land n_1 < n_2\}\end{aligned}$$

Denotational Semantics for IMP

Arithmetic and Boolean Expressions in Function-Style

$$\mathcal{A}[\![\underline{n}]\!](s) = n$$

$$\mathcal{A}[\![!l]\!](s) = s(l) \quad \text{if } l \in \mathsf{dom}(s)$$

$$\mathcal{A}[\![a_1 \underline{+} a_2]\!](s) = \mathcal{A}[\![a_1]\!](s) + \mathcal{A}[\![a_2]\!](s)$$

$$\mathcal{B}[\![\underline{\texttt{true}}]\!](s) = \texttt{true}$$

$$\mathcal{B}[\![\underline{\texttt{false}}]\!](s) = \texttt{false}$$

$$\mathcal{B}[\![a_1 \underline{\land} a_2]\!](s) = \mathcal{B}[\![b_1]\!](s) \land \mathcal{B}[\![b_2]\!](s)$$

$$\mathcal{B}[\![b_1 \underline{\geq} a_2]\!](s) = \begin{cases} \texttt{true} & \text{if } \mathcal{A}[\![a_1]\!](s) \geq \mathcal{A}[\![a_2]\!](s) \\ \texttt{false} & \text{otherwise} \end{cases}$$

Denotational Semantics for IMP

Commands

$$\begin{aligned}
\mathcal{C}[\![\textbf{skip}]\!] &= \{(s,s)\} \\
\mathcal{C}[\![l := a]\!] &= \{(s, s + \{l \mapsto n\}) \mid (s,n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![c_1 \ ; \ c_2]\!] &= \{(s,s'') \mid \exists s'.\ (s,s') \in \mathcal{C}[\![c_1]\!] \wedge (s',s'') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2]\!] &= \{(s,s') \mid (s,\texttt{true}) \in \mathcal{B}[\![b]\!] \wedge (s,s') \in \mathcal{C}[\![c_1]\!]\} \cup \\
&\quad\ \{(s,s') \mid (s,\texttt{false}) \in \mathcal{B}[\![b]\!] \wedge (s,s') \in \mathcal{C}[\![c_2]\!]\}
\end{aligned}$$

Denotational Semantics for IMP
Commands in Function-Style

$$\mathcal{C}[\![\textbf{skip}]\!](s) = s$$

$$\mathcal{C}[\![l := a]\!](s) = s + \{l \mapsto (\mathcal{A}[\![a]\!](s))\}$$

$$\mathcal{C}[\![c_1 \ ; \ c_2]\!] = \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!]$$
$$(\text{or } \mathcal{C}[\![c_1 \ ; \ c_2]\!](s) = \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!](s)) \ )$$

$$\mathcal{C}[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!](s) = \begin{cases} \mathcal{C}[\![c_1]\!](s) & \text{if } \mathcal{B}[\![b]\!](s) = \texttt{true} \\ \mathcal{C}[\![c_2]\!](s) & \text{if } \mathcal{B}[\![b]\!](s) = \texttt{false} \end{cases}$$

denotational semantics is often *compositional*

Denotational Semantics for IMP

Commands

(cont'd)

$$\begin{aligned}\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = {} & \{(s,s) \mid (s, \texttt{false}) \in \mathcal{B}[\![b]\!]\} \cup \\ & \{(s,s') \mid (s, \texttt{true}) \in \mathcal{B}[\![b]\!] \wedge \\ & \quad \exists s''.\ (s,s'') \in \mathcal{C}[\![c]\!] \wedge (s'',s') \in \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]\}\end{aligned}$$

$$\begin{aligned}\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!](s) &= \mathcal{C}[\![\textbf{if } b \textbf{ then } c\ ;\ (\textbf{while } b \textbf{ do } c) \textbf{ else skip}]\!](s) \\ &= \begin{cases}\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!](\mathcal{C}[\![c]\!](s)) & \text{if } \mathcal{B}[\![b]\!](s) = \texttt{true} \\ \mathcal{C}[\![\textbf{skip}]\!](s) & \text{if } \mathcal{B}[\![b]\!](s) = \texttt{false}\end{cases}\end{aligned}$$

Problem: this is not a function definition;
it is a recursive equation, we require its solution

Recursive Equations – Example

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ f(x-1) + 2x - 1 & \text{otherwise} \end{cases}$$

Question: What function(s) satisfy this equation?
Answer: $f(x) = x^2$

Recursive Equations – Example II

$$g(x) = g(x) + 1$$

Question: What function(s) satisfy this equation?
Answer: none

Australian National University

Recursive Equations – Example III

$$h(x) = 4 \cdot h\left(\frac{x}{2}\right)$$

Question: What function(s) satisfy this equation?
Answer: multiple

Solving Recursive Equations

Build a solution by approximation (interpret functions as sets)

$$f_0 = \emptyset$$

$$f_1 = \begin{cases} 0 & \text{if } x = 0 \\ f_0(x-1) + 2x - 1 & \text{otherwise} \end{cases}$$
$$= \{(0,0)\}$$

$$f_2 = \begin{cases} 0 & \text{if } x = 0 \\ f_1(x-1) + 2x - 1 & \text{otherwise} \end{cases}$$
$$= \{(0,0), (1,1)\}$$

$$f_3 = \begin{cases} 0 & \text{if } x = 0 \\ f_2(x-1) + 2x - 1 & \text{otherwise} \end{cases}$$
$$= \{(0,0), (1,1), (2,4)\}$$

Solving Recursive Equations

Model this process as higher-order function F that takes the approximation f_k as input and returns the next approximation.

$$F : (\mathbb{N} \rightharpoonup \mathbb{N}) \to (\mathbb{N} \rightharpoonup \mathbb{N})$$

where

$$(F(f))(x) = \begin{cases} 0 & \text{if } x = 0 \\ f(x-1) + 2x - 1 & \text{otherwise} \end{cases}$$

Iterate till a fixed point is reached ($f = F(f)$)

Fixed Point

Definition

Given a function $F : A \to A$, $a \in A$ is a *fixed point* of F if $F(a) = a$.

Notation: Write $a = \mathsf{fix}\,(F)$ to indicate that a is a fixed point of F.

Idea: Compute fixed points iteratively, starting from the completely undefined function. The fixed point is the limit of this process:

$$\begin{aligned} f &= \mathsf{fix}\,(F) \\ &= f_0 \cup f_1 \cup f_2 \cup \ldots \\ &= \emptyset \cup F(\emptyset) \cup F(F(\emptyset)) \cup \ldots \\ &= \bigcup_{i \geq 0}^{\infty} F^i(\emptyset) \end{aligned}$$

Denotational Semantics for **while**

$$\mathcal{C}[\![\textbf{while}\ b\ \textbf{do}\ c]\!] = \mathsf{fix}\,(F)$$

where

$$\begin{aligned} F(f) = &\{(s, s) \mid (s, \texttt{false}) \in \mathcal{B}[\![b]\!]\} \cup \\ &\{(s, s') \mid (s, \texttt{true}) \in \mathcal{B}[\![b]\!] \wedge \\ &\qquad \exists s''.\ (s, s'') \in \mathcal{C}[\![c]\!] \wedge (s'', s') \in f\} \end{aligned}$$

Denotational Semantics – Example

$$\mathcal{C}[\![\textbf{while } !l \geq 0 \textbf{ do } m := !l + !m \; ; \; l := !l + (-1)]\!]$$

$$f_0 = \emptyset$$

$$f_1 = \begin{cases} s & \text{if } !l < 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$f_2 = \begin{cases} s & \text{if } !l < 0 \\ s + \{l \mapsto -1, m \mapsto s(m)\} & \text{if } !l = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$f_3 = \begin{cases} s & \text{if } !l < 0 \\ s + \{l \mapsto -1\} & \text{if } !l = 0 \\ s + \{l \mapsto -1, m \mapsto 1 + s(m)\} & \text{if } !l = 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$f_4 = \begin{cases} s & \text{if } !l < 0 \\ s + \{l \mapsto -1\} & \text{if } !l = 0 \\ s + \{l \mapsto -1, m \mapsto 1 + s(m)\} & \text{if } !l = 1 \\ s + \{l \mapsto -1, m \mapsto 3 + s(m)\} & \text{if } !l = 2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Fixed Points

- Why does (fix F) have a solution?
- What if there are several solutions?
  (which should we take)

Fixed Point Theory

Definition (sub preserving)

A function F *preserves suprema* if for every chain $X_1 \subseteq X_2 \subseteq \ldots$

$$F(\bigcup_i X_i) = \bigcup_i F(X_i) \ .$$

Lemma

Every suprema-preserving function F is monotone increasing.

$$X \subseteq Y \Longrightarrow F(X) \subseteq F(Y)$$

(works for arbitrary partially ordered sets)

Kleene's fixed point theorem

Theorem

Let F be a suprema-preserving function. The least fixed point of F exists and is equal to

$$\bigcup_{i \geq 0} F^i(\emptyset)$$

Australian National University

$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$

$$
\begin{aligned}
& \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!](s) \\
= {} & \mathsf{fix}\,(F) \\
= {} & \begin{cases} \mathcal{C}[\![c]\!]^k(s) & \text{if } k \geq 0 \text{ such that } \mathcal{B}[\![b]\!](\mathcal{C}[\![c]\!]^k(s)) = \texttt{false} \\ & \text{and } \mathcal{B}[\![b]\!](\mathcal{C}[\![c]\!]^i(s)) = \texttt{true} \text{ for all } 0 \leq i < k \\ \text{undefined} & \text{if } \mathcal{B}[\![b]\!](\mathcal{C}[\![c]\!]^i(s)) = \texttt{true} \text{ for all } i \geq 0 \end{cases}
\end{aligned}
$$

This may be what you would have expected, but now it is grounded on well-known mathematics

Exercises

- Show that **skip** $;\, c$ and $c\,;$ **skip** are equivalent.
- What does equivalent mean in the context of denotational semantics?
- Show that $(c_1\ ;\ c_2)\ ;\ c_3$ is equivalent to $c_1\ ;\ (c_2\ ;\ c_3)$.

Section 16

Partial and Total Correctness

Styles of semantics

Operational
Meanings for program phrases defined in terms of the steps of computation they can take during program execution.

Denotational
Meanings for program phrases defined abstractly as elements of some suitable mathematical structure.

Axiomatic
Meanings for program phrases defined indirectly via the axioms and rules of some logic of program properties.

Styles of semantics

Operational
– *how* to evaluate programs (interpreter)
– close connection to implementations

Denotational
Meanings for program phrases defined abstractly as elements of some suitable mathematical structure.

Axiomatic
Meanings for program phrases defined indirectly via the axioms and rules of some logic of program properties.

Styles of semantics

Operational
– *how* to evaluate programs (interpreter)
– close connection to implementations

Denotational
– *what* programs calculate (compiler)
– simplifies equational reasoning (semantic equivalence)

Axiomatic
Meanings for program phrases defined indirectly via the axioms and rules of some logic of program properties.

Styles of semantics

Operational
– *how* to evaluate programs (interpreter)
– close connection to implementations

Denotational
– *what* programs calculate (compiler)
– simplifies equational reasoning (semantic equivalence)

Axiomatic
– *describes properties* of programs
– allows reasoning about the correctness of programs

Assertions

Axiomatic semantics *describe properties* of programs. Hence it requires

- a language for expressing properties
- proof rules to establish the validity of properties w.r.t. programs

Examples

- value of l is greater than 0
- value of l is even
- value of l is prime
- eventually the value of l will 0
- $\ldots$

Applications

- proving correctness
- documentation
- test generation
- symbolic execution
- bug finding
- malware detection
- …

Assertion Languages

- (English)
- first-order logic ($\forall, \exists, \wedge, \neg, =, R(x), \ldots$)
- temporal and modal logic ($\Box, \Diamond, \circledcirc,$ **Until**$, \ldots$)
- special-purpose logics (Alloy, Z3, . . .)

Assertions as Comments

assertions are (should) be used in code regularly

```
/* Precondition: 0 <= i < A.length */
/* Postcodition: returns A[i] */
public int get (int i) {
        return A[i];
}
```

- useful as documentation or run-time checks
- no guarantee that they are correct
- sometimes not useful (e.g. /*increment i*/)

aim: make this rigorous by defining the semantics of a language using pre- and post-conditions

Partial Correctness

$$\{P\}\ c\ \{Q\}$$

Meaning: if P holds before c, and c executes *and terminates* then Q holds afterwards

Partial Correctness – Examples

- $\{l = 21\}\ l := !l + !l\ \{l = 42\}$
- $\{l = 0 \land m = i\}$
  $k := 0\ ;$
  while $!l \neq\ !m$
  do
  $\quad k := !k - 2\ ;$
  $\quad l := !l + 1$
  $\{k = -i - i\}$

Note: i is a ghost variable
we do not use dereferencing in conditions

Partial Correctness – Examples

The second example is a valid partial correctness statement.

Lemma

$$\forall s, s'.\quad k, l, m \in \textit{dom}(s) \wedge s(l) = 0 \wedge$$
$$\mathcal{C}[\![k := 0 \,;\, \textbf{while}\ !l \neq !m\ \textbf{do}\ (k := !k - 2 \,;\, l := !l + 1)]\!](s) = s'$$
$$\Longrightarrow s'(k) = -s(m) - s(m)$$

Partial Correctness – Examples

Is the following partial correctness statement valid?

- $\{l = 0 \wedge m = i\}$
  $k := 0\ ;$
  while $!l \neq !m$
  do
  $\quad k := !k + !l\ ;$
  $\quad l := !l + 1$
  $\{k = i\}$

Total Correctness

- partial correctness specifications do not ensure termination
- sometimes termination is needed

$$[P]\ c\ [Q]$$

Meaning: if P holds, then c will terminate and Q holds afterwards

Total Correctness – Example

- $[l = 0 \wedge m = i \wedge \mathbf{i} \geq \mathbf{0}]$
  $k := 0\ ;$
  while $!l \neq\ !m$
  do
  $\quad k :=\ !k - 2\ ;$
  $\quad l :=\ !l + 1$
  $[k = -i - i]$

Assertions

What properties do we want to state in pre-conditions and post-conditions; so far

- locations (program variables)
- equality
- logical/ghost variables (e.g. i)
- comparison
- we have not used ‘pointers’

choice of assertion language influences the sort of properties we can specify

Assertions – Syntax

Booleans	$b \in \mathbb{B}$
Integers (Values)	$n \in \mathbb{Z}$
Locations	$l \in \mathbb{L} \quad = \{l, l_0, l_1, l_2, \ldots\}$
Logical variables	$i \in \mathbf{LVar} = \{i, i_0, i_1, i_2, \ldots\}$
Operations	$aop ::= +$

Expressions

$$
\begin{aligned}
\mathsf{aexp}_i &::= n \mid l \mid i \mid \mathsf{aexp}_i \; aop \; \mathsf{aexp}_i \\
\mathsf{assn} &::= b \mid \mathsf{aexp}_i \geq \mathsf{aexp}_i \mid \\
&\quad\;\; \mathsf{assn} \wedge \mathsf{assn} \mid \mathsf{assn} \vee \mathsf{assn} \mid \\
&\quad\;\; \mathsf{assn} \Rightarrow \mathsf{assn} \mid \neg\mathsf{assn} \mid \\
&\quad\;\; \forall i.\ \mathsf{assn} \mid \exists i.\ \mathsf{assn}
\end{aligned}
$$

Note: bexp included in assn; assn not minimal

Assertions – Satisfaction

when does a store s satisfy an assertion

- need interpretation for logical variables

$$I : \mathbf{LVar} \to \mathbb{Z}$$

- denotation function $\mathcal{A}_I[\![\_]\!]$ (similar to $\mathcal{A}[\![\_]\!]$

$$\begin{aligned}
\mathcal{A}_I[\![n]\!](s,I) &= n \\
\mathcal{A}_I[\![l]\!](s,I) &= s(l), \qquad l \in \mathsf{dom}(s) \\
\mathcal{A}_I[\![i]\!](s,I) &= I(i), \qquad i \in \mathsf{dom}(I) \\
\mathcal{A}_I[\![a_1 + a_2]\!](s,I) &= \mathcal{A}_I[\![a_1]\!](s,I) + \mathcal{A}[\![a_2]\!](s,I)
\end{aligned}$$

Assertion Satisfaction

define satisfaction relation for assertions on a given state s

$s \models_I \texttt{true}$	
$s \models_I a_1 \geq a_2$	if $\mathcal{A}_I[\![a_1]\!](s, I) \geq \mathcal{A}_I[\![a_2]\!](s, I)$
$s \models_I P_1 \wedge P_2$	if $s \models_I P_1$ and $s \models_I P_2$
$s \models_I P_1 \vee P_2$	if $s \models_I P_1$ or $s \models_I P_2$
$s \models_I P_1 \Rightarrow P_2$	if $s \not\models_I P_1$ or $s \models_I P_2$
$s \models_I \neg P$	if $s \not\models_I P$
$s \models_I \forall i.\ P$	if $\forall n \in \mathbb{Z}.\ s \models_{I+\{i \mapsto n\}} P$
$s \models_I \exists i.\ P$	if $\exists n \in \mathbb{Z}.\ s \models_{I+\{i \mapsto n\}} P$

an assertion is *valid* ($\models P$) if it is valid in any store, under any interpretation

$$\forall s, I.\ s \models_I P$$

Partial Correctness – Satisfiability

A partial correctness statement $\{P\}\ c\ \{Q\}$ is *satisfied* in store s and under interpretation I ($s \models_I \{P\}\ c\ \{Q\}$) if

$$\forall s'.\ \text{if } s \models_I P \text{ and } \mathcal{C}[\![c]\!](s) = s' \text{ then } s' \models_I Q\ .$$

Partial Correctness – Validity

Assertion validity

An assertion P is *valid* (*holds*) ($\models P$) if it is *valid* in any store under interpretation.

$$\models P \;:\Longleftrightarrow\; \forall s, I.\ s \models_I P$$

Partial correctness validity

A partial correctness statement $\{P\}\ c\ \{Q\}$ is *valid* ($\models \{P\}\ c\ \{Q\}$) if it is valid in any store under interpretation.

$$\models \{P\}\ c\ \{Q\} \;:\Longleftrightarrow\; \forall s, I.\ s \models_I \{P\}\ c\ \{Q\}$$

Proving Specifications

how to proof the (partial) correctness of $\{P\}\ c\ \{Q\}$

- show $\forall s, I. s \models_I \{P\}\ c\ \{Q\}$
- $s \models_I \{P\}\ c\ \{Q\}$ requires denotational semantics $\mathcal{C}$

- we can do this manually, but ...
- we can derive inference rules and axioms (axiomatic semantics)
- allows derivation of correctness statements without reasoning about stores and interpretations

Section 17

Axiomatic Semantics

Floyd-Hoare Logic

Idea: develop proof system as an inductively-defined set; every member will be a valid partial correctness statement

Judgement

$$\vdash \{P\}\; c\; \{Q\}$$

Floyd-Hoare Logic – Skip

(skip) $\vdash \{P\}$ **skip** $\{P\}$

Floyd-Hoare Logic – Assignment

(assign) $\vdash \{P[a/l]\}\ l := a\ \{P\}$

Notation: $P[a/l]$ denotes substitution of a for l in P;
in operational semantics we wrote $\{a/l\}\, P$

Example

$$\{7 = 7\}\ l := 7\ \{l = 7\}$$

Floyd-Hoare Logic – Incorrect Assignment

(wrong1) $\vdash \{P\}\ l := a\ \{P[a/l]\}$

Example

$$\{l = 0\}\ l := 7\ \{7 = 0\}$$

(wrong2) $\vdash \{P\}\ l := a\ \{P[l/a]\}$

Example

$$\{l = 0\}\ l := 7\ \{l = 0\}$$

Floyd-Hoare Logic – Sequence, If, While

(seq) $$\frac{\vdash \{P\}\ c_1\ \{R\} \qquad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1\ ;\ c_2\ \{Q\}}$$

(if) $$\frac{\vdash \{P \wedge b\}\ c_1\ \{Q\} \qquad \vdash \{P \wedge \neg b\}\ c_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q\}}$$

(while) $$\frac{\vdash \{P \wedge b\}\ c\ \{P\}}{\vdash \{P\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{P \wedge \neg b\}}$$

P acts as *loop invariant*

Floyd-Hoare Logic – Consequence

We cannot combine arbitrary triple yet

$$\dfrac{\dfrac{}{\vdash \{3 = 3\}\ l := 3\ \{l = 3\}}\text{(assign)} \qquad \dfrac{\cdots}{\vdash \{l \geq 2\}\ l :=!l - 2\ \{l \geq 0\}}}{\vdash \{3 = 3\}\ l := 3\ ;\ l :=!l - 2\ \{l \geq 0\}}$$

Floyd-Hoare Logic – Consequence

strengthen pre-conditions and weaken post-conditions

(cons) $$\frac{\models P \Rightarrow P' \qquad \vdash \{P'\}\ c\ \{Q'\} \qquad \models Q' \Rightarrow Q}{\vdash \{P\}\ c\ \{Q\}}$$

Recall: $\models P \Rightarrow P'$ denotes assertion validity

Floyd-Hoare Logic – Summary

(skip) $\vdash \{P\}\ \textbf{skip}\ \{P\}$

(assign) $\vdash \{P[a/l]\}\ l := a\ \{P\}$

(seq) $$\frac{\vdash \{P\}\ c_1\ \{R\} \qquad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1\ ;\ c_2\ \{Q\}}$$

(if) $$\frac{\vdash \{P \wedge b\}\ c_1\ \{Q\} \qquad \vdash \{P \wedge \neg b\}\ c_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q\}}$$

(while) $$\frac{\vdash \{P \wedge b\}\ c\ \{P\}}{\vdash \{P\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{P \wedge \neg b\}}$$

(cons) $$\frac{\models P \Rightarrow P' \qquad \vdash \{P'\}\ c\ \{Q'\} \qquad \models Q' \Rightarrow Q}{\vdash \{P\}\ c\ \{Q\}}$$

Floyd-Hoare Logic – Exercise

$$
\begin{array}{l}
\{l_0 = n \wedge n > 0\} \\
l_1 := 1 \; ; \\
\textbf{while } !l_0 > 0 \textbf{ do} \\
\qquad l_1 := !l_1 \cdot !l_0 \; ; \\
\qquad l_0 := !l_0 - 1 \\
\{l_1 = n!\}
\end{array}
$$

Soundness and Completeness

how do $\vdash$ (judgement) and $\models$ (validity) relate?

Soundness:
if a partial correctness statement can be derived ($\vdash$) then is is valid ($\models$)

Completeness:
if the statement is valid ($\models$) then a derivation exists ($\vdash$)

Soundness and Completeness

Theorem (Soundness)

If $\vdash \{P\}\ c\ \{Q\}$ *then* $\models \{P\}\ c\ \{Q\}$.

Proof.

Induction on the derivation of $\vdash \{P\}\ c\ \{Q\}$. $\square$

Soundness and Completeness

Conjecture (Completeness)

If $\models \{P\}\ c\ \{Q\}$ *then* $\vdash \{P\}\ c\ \{Q\}$.

Rule (cons) spoils completeness

$$\text{(cons)} \quad \frac{\models P \Rightarrow P' \qquad \vdash \{P'\}\ c\ \{Q'\} \qquad \models Q' \Rightarrow Q}{\vdash \{P\}\ c\ \{Q\}}$$

Can we derive $\models P \Rightarrow P'$?
No, according to Gödel's incompleteness theorem (1931)

Soundness and Completeness

Theorem (Relative Completeness)

$P, Q \in$ *assn*, $c \in$ *com*. $\models \{P\}\ c\ \{Q\}$ *implies* $\vdash \{P\}\ c\ \{Q\}$.

Floyd-Hoare logic is no more incomplete than our language of assertions

Proof depends on the notion of *weakest liberal preconditions*.

Decorated Programs

Observation: once loop invariants and uses of consequence are identified, the structure of a derivation in Floyd-Hoare logic is determined
Write "proofs" by decorating programs with:

- a precondition ($\{P\}$)
- a postcondition ($\{Q\}$)
- invariants ($\{I\}$**while** b **do** c)
- uses of consequence ($\{R\} \Rightarrow \{S\}$)
- assertions between sequences ($c_1 \; ; \{T\}c_2$)

decorated programs describe a valid Hoare logic proof if the rest of the proof tree's structure is implied
(caveats: Invariants are constrained, etc.)

(Informal) Rules for Decoration

Idea: check whether a decorated program represents a valid proof using local consistency checks

skip

pre and post-condition should be the same

$\{P\}$
skip
$\{P\}$

(skip) $\vdash \{P\}$ **skip** $\{P\}$

(Informal) Rules for Decoration

assignment

use the substitution from the rule

$$\{P[a/l]\}$$
$$l := a$$
$$\{P\}$$

(assign) $\vdash \{P[a/l]\}\ l := a\ \{P\}$

sequencing

$\{P\}\ c_1\ \{R\}$ and $\{R\}\ c_2\ \{Q\}$ should be (recursively) locally consistent

$$\{P\}$$
$$c_1\ ;$$
$$\{R\}$$
$$c_2$$
$$\{Q\}$$

(seq) $\dfrac{\vdash \{P\}\ c_1\ \{R\} \qquad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1\ ;\ c_2\ \{Q\}}$

(Informal) Rules for Decoration

if then

both branches are locally consistent; add condition to both

$\{P\}$
if b **then**
  $\{P \wedge b\}$
  c_1
  $\{Q\}$
else
  $\{P \wedge \neg b\}$
  c_2
  $\{Q\}$
$\{Q\}$

$$\text{(if)}\ \frac{\vdash \{P \wedge b\}\ c_1\ \{Q\} \qquad \vdash \{P \wedge \neg b\}\ c_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q\}}$$

(Informal) Rules for Decoration

while

add/create loop invariant

$\{P\}$
while b **do**
  $\{P \wedge b\}$
  c
  $\{P\}$
$\{P \wedge \neg b\}$

$$\text{(while)}\ \frac{\vdash \{P \wedge b\}\ c\ \{P\}}{\vdash \{P\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{P \wedge \neg b\}}$$

(Informal) Rules for Decoration

consequence

always write a (valid) implication

$$\{P\} \Rightarrow$$
$$\{P'\}$$

(cons) $$\frac{\models P \Rightarrow P' \qquad \vdash \{P'\}\, c\, \{Q'\} \qquad \models Q' \Rightarrow Q}{\vdash \{P\}\, c\, \{Q\}}$$

Floyd-Hoare Logic – Exercise

$$\{l_0 = n \wedge n > 0\}$$
$$l_1 := 1 \;;$$
$$\textbf{while } !l_0 > 0 \textbf{ do}$$
$$\quad l_1 := !l_1 \cdot l_0 \;;$$
$$\quad l_0 := !l_0 - 1$$
$$\{l_1 = n!\}$$

Floyd-Hoare Logic – Exercise

$\{l_0 = n \wedge n > 0\} \Rightarrow$

$\{1 = 1 \wedge l_0 = n \wedge n > 0\}$

$l_1 := 1\ ;$

$\{l_1 = 1 \wedge l_0 = n \wedge n > 0\} \Rightarrow$

$\{l_1 \cdot l_0! = n! \wedge l_0 \geq 0\}$

while $!l_0 > 0$ **do**

$\{l_1 \cdot l_0! = n! \wedge l_0 > 0 \wedge l_0 \geq 0\} \Rightarrow$

$\{l_1 \cdot l_0 \cdot (l_0 - 1)! = n! \wedge (l_0 - 1) \geq 0\}$

$l_1 := !l_1 \cdot l_0\ ;$

$\{l_1 \cdot (l_0 - 1)! = n! \wedge (l_0 - 1) \geq 0\}$

$l_0 := !l_0 - 1$

$\{l_1 \cdot l_0! = n! \wedge l_0 \geq 0\}$

$\{l_1 \cdot l_0! = n! \wedge (l_0 \geq 0) \wedge \neg(l_0 > 0)\} \Rightarrow$

$\{l_1 = n!\}$

Section 18

Weakest Preconditions

Generating Preconditions

$$\{\ ?\ \}\ c\ \{Q\}$$

- many possible preconditions
- some are more useful than others

Australian National University

Weakest Liberal Preconditions

Intuition: the weakest liberal precondition for c and Q is the *weakest* assertion P such that $\{P\}\ c\ \{Q\}$ is valid

Definition (Weakest Liberal Precondition)

P is a *weakest liberal precondition* of c and Q (wlp(c, Q)) if

$$\forall s, I.\ s \models_I P \iff \mathcal{C}[\![c]\!](s) \text{ is undefined } \vee\ \mathcal{C}[\![c]\!](s) \models_I Q$$

Weakest Preconditions

$$
\begin{aligned}
\textsf{wlp}(\textbf{skip}, Q) &= Q \\
\textsf{wlp}(l := a, Q) &= Q[a/l] \\
\textsf{wlp}((c_1 \ ; \ c_2), Q) &= \textsf{wlp}(c_1, \textsf{wlp}(c_2, Q)) \\
\textsf{wlp}(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, Q) &= (b \implies \textsf{wlp}(c_1, Q)) \land \\
&\quad\ (\neg b \implies \textsf{wlp}(c_2, Q)) \\
\textsf{wlp}(\textbf{while } b \textbf{ do } c, Q) &= (b \implies \textsf{wlp}(c, \textsf{wlp}(\textbf{while } b \textbf{ do } c, Q))) \ \land \\
&\quad\ (\neg b \implies Q) \\
&= \bigwedge_i F_i(Q)
\end{aligned}
$$

where

$$
\begin{aligned}
F_0(Q) &= \mathtt{true} \\
F_{i+1}(Q) &= (\neg b \implies Q) \land (b \implies \textsf{wlp}(c, F_i(Q)))
\end{aligned}
$$

(Greatest fixed point)

Properties of Weakest Preconditions

Lemma (Correctness of wlp)

$\forall c \in \textit{com}, Q \in \textit{assn}.$
$\quad \models \{\textit{wlp}(c, Q)\}\ c\ \{Q\}$ *and*
$\quad \forall R \in \textit{assn}.\ \models \{R\}\ c\ \{Q\}$ *implies* $(R \Longrightarrow \textit{wlp}(c, Q))$

Lemma (Provability of wlp)

$\forall c \in \textit{com}, Q \in \textit{assn}.\ \vdash \{\textit{wlp}(c, Q)\}\ c\ \{Q\}$

Soundness and Completeness

Theorem (Relative Completeness)

$P, Q \in$ *assn*$, c \in$ *com*. $\models \{P\}\ c\ \{Q\}$ *implies* $\vdash \{P\}\ c\ \{Q\}$.

Proof Sketch.

- let $\{P\}\ c\ \{Q\}$ be a valid partial correctness specification
- by the first lemma we have $\models P \Longrightarrow \text{wlp}(c, Q)$
- by the second lemma we have $\vdash \{\text{wlp}(c, Q)\}\ c\ \{Q\}$
- hence $\vdash \{P\}\ c\ \{Q\}$, using the Rule (cons)

$\square$

Total Correctness

Definition (Weakest Precondition)

P is a *weakest precondition* of c and Q (wp(c, Q)) if

$$\forall s, I.\ s \models_I P \iff \mathcal{C}[\![c]\!](s) \models_I Q$$

all rules are the same, except the one for while. This requires a fresh ghost variable that guarantees termination

Lemma (Correctness of wp)

$\forall c \in$ *com*, $Q \in$ *assn*.
$\models [$*wp*$(c, Q)]\ c\ [Q]$ *and*
$\forall R \in$ *assn*. $\models [R]\ c\ [Q]$ *implies* $(R \Longrightarrow$ *wp*$(c, Q))$

(for appropriate definition of $\models$)

Strongest Postcondition

$$\{P\}\, c\, \{\ ?\ \}$$

- wlp motivates backwards reasoning
- this seems unintuitive and unnatural
- however, often it is known what a program is supposed to do
- sometimes forward reasoning is useful, e.g. reverse engineering

Strongest Postcondition

$$
\begin{aligned}
\mathsf{sp}(\textbf{skip}, P) &= P \\
\mathsf{sp}(l := a, P) &= \exists v.\ (l = a[v/l] \land P[v/l]) \\
\mathsf{sp}((c_1\ ;\ c_2), P) &= \mathsf{sp}(c_2, \mathsf{sp}(c_1, P)) \\
\mathsf{sp}(\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2, P) &= (\mathsf{sp}(c_1, b \land P)) \lor (\mathsf{sp}(c_2, \neg b \land P)) \\
\mathsf{sp}(\textbf{while}\ b\ \textbf{do}\ c, P) &= \mathsf{sp}(\textbf{while}\ b\ \textbf{do}\ c, \mathsf{sp}(c, P \land b)) \lor (\neg b \land P)
\end{aligned}
$$

where

$$
\begin{aligned}
F_0(P) &= \texttt{false} \\
F_{i+1}(P) &= (\neg b \land P) \lor (\mathsf{sp}(c, F_i(P \land b)))
\end{aligned}
$$

(Least fixed point)

Section 19

Concurrency

Concurrency and Distribution

so far we concentrated on semantics for sequential computation
but the world is not sequential...

- hardware is intrinsically parallel
- multi-processor architectures
- multi-threading (perhaps on a single processor)
- networked machines

Problems

aim: languages that can be used to model computations that execute in parallel and on distributed architectures

problems

- state-space explosion
  with n threads, each of which can be in 2 states, the system has 2^n states
- state-spaces become complex
- computation becomes nondeterministic
- competing for access to resources may deadlock or suffer starvation
- partial failure (of some processes, of some machines in a network, of some persistent storage devices)
- communication between different environments
- partial version change
- communication between administrative regions with partial trust (or, indeed, no trust)
- protection against malicious attack
- ...

Problems

this course can only scratch the surface

concurrency theory is a broad and active field for research

Process Calculi

- Observation (1970s): computers with shared-nothing architectures communicating by sending messages to each other would be important
  [Edsger W. Dijkstra, Tony Hoare, Robin Milner, and others]
- Hoare's Communicating Sequential Processes (CSP) is an early and highly-influential language that capture a message passing form of concurrency
- many languages have built on CSP including Milner's CCS and π-calculus, Petri nets, and others

IMP – Parallel Commands

we extend our while-language that is based on aexp, bexp and com

Syntax

$$\mathsf{com} ::= \ldots \mid \mathsf{com} \parallel \mathsf{com}$$

Semantics

(par1) $$\frac{\langle c_0\ ,\ s\rangle \longrightarrow \langle c_0'\ ,\ s'\rangle}{\langle c_0 \parallel c_1\ ,\ s\rangle \longrightarrow \langle c_0' \parallel c_1\ ,\ s'\rangle}$$

(par2) $$\frac{\langle c_1\ ,\ s\rangle \longrightarrow \langle c_1'\ ,\ s'\rangle}{\langle c_0 \parallel c_1\ ,\ s\rangle \longrightarrow \langle c_0 \parallel c_1'\ ,\ s'\rangle}$$

IMP – Parallel Commands

Typing

(thread) $$\frac{\Gamma \vdash c : \mathsf{unit}}{\Gamma \vdash c : \mathsf{proc}}$$

(par) $$\frac{\Gamma \vdash c_0 : \mathsf{proc} \qquad \Gamma \vdash c_1 : \mathsf{proc}}{\Gamma \vdash c_0 \parallel c_1 : \mathsf{proc}}$$

Parallel Composition: Design Choices

- threads do not return a value
- threads do not have an identity
- termination of a thread cannot be observed within the language
- threads are not partitioned into ‘processes’ or machines
- threads cannot be killed externally

Asynchronous Execution

- semantics allow interleavings

$$\langle l := 1 \parallel l := 2\,,\ \{l \mapsto 0\}\rangle \longrightarrow \langle \textbf{skip} \parallel l := 2\,,\ \{l \mapsto 1\}\rangle \longrightarrow \langle \textbf{skip} \parallel \textbf{skip}\,,\ \{l \mapsto 2\}\rangle$$

$$\langle l := 1 \parallel l := 2\,,\ \{l \mapsto 0\}\rangle \longrightarrow \langle l := 1 \parallel \textbf{skip}\,,\ \{l \mapsto 2\}\rangle \longrightarrow \langle \textbf{skip} \parallel \textbf{skip}\,,\ \{l \mapsto 1\}\rangle$$

- assignments and dereferencing are *atomic*

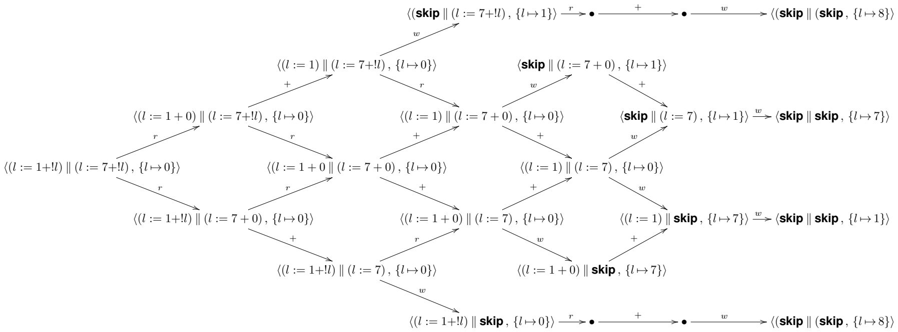
$$\langle l := N \parallel l := 2\,,\ \{l \mapsto 0\}\rangle \longrightarrow \langle \textbf{skip} \parallel l := 2\,,\ \{l \mapsto N\}\rangle \longrightarrow \langle \textbf{skip} \parallel \textbf{skip}\,,\ \{l \mapsto 2\}\rangle$$

$$\langle l := N \parallel l := 2\,,\ \{l \mapsto 0\}\rangle \longrightarrow \langle l := N \parallel \textbf{skip}\,,\ \{l \mapsto 2\}\rangle \longrightarrow \langle \textbf{skip} \parallel \textbf{skip}\,,\ \{l \mapsto N\}\rangle$$

for $N = 3498734590879238429384$.
(not something as the first word of one and the second word of the other)

Asynchronous Execution

- interleavings in $\langle (l := 1{+}!l) \parallel (l := 7{+}!l) \,,\, \{l \mapsto 0\}\rangle$



Morals

- combinatorial explosion
- drawing state-space diagrams only works for really tiny examples
- almost certainly the programmer does not want all those 3 outcomes to be possible
- complicated/impossible to analyse without formal methods

Parallel Commands – Nondeterminism

Semantics

(par1) $$\frac{\langle c_0\ ,\ s\rangle \longrightarrow \langle c_0'\ ,\ s'\rangle}{\langle c_0 \parallel c_1\ ,\ s\rangle \longrightarrow \langle c_0' \parallel c_1\ ,\ s'\rangle}$$

(par2) $$\frac{\langle c_1\ ,\ s\rangle \longrightarrow \langle c_1'\ ,\ s'\rangle}{\langle c_0 \parallel c_1\ ,\ s\rangle \longrightarrow \langle c_0 \parallel c_1'\ ,\ s'\rangle}$$

(+maybe rules for termination)

- study of nondeterminism
- $\parallel$ is not a partial function from state to state; big-step semantics needs adaptation
- can we achieve parallelism by nondeterministic interleavings
- communication via shared variable

Study of Parallelism (or Concurrency)

includes

Study of Nondeterminism

Dijkstra’s Guarded Command Language (GCL)

- defined by Edsger Dijkstra for predicate transformer semantics
- combines programming concepts in a compact/abstract way
- simplicity allows correctness proofs
- closely related to Hoare logic

GCL – Syntax

- arithmetic expressions: aexp (as before)
- Boolean expressions: bexp (as before)
- Commands:

$$\begin{aligned}\mathsf{com} ::= {} & \textbf{skip} \mid \textbf{abort} \mid l := \mathsf{aexp} \mid \mathsf{com}\ ;\ \mathsf{com} \mid \\ & \textbf{if}\ \mathsf{gc}\ \textbf{fi} \mid \textbf{do}\ \mathsf{gc}\ \textbf{od}\end{aligned}$$

- Guarded Commands:

$$\begin{aligned}\mathsf{gc} ::= {} & \mathsf{bexp} \to \mathsf{com} \mid \\ & \mathsf{gc}\ [\!]\ \mathsf{gc}\end{aligned}$$

GCL – Semantics

- assume we have semantic rules for bexp and aexp (standard) we skip the deref-operator from now on
- assume a new configuration `fail`

Guarded Commands

(pos) $$\frac{\langle b\,,\,s\rangle \longrightarrow \langle \texttt{true}\,,\,s\rangle}{\langle b \rightarrow c\,,\,s\rangle \longrightarrow \langle c\,,\,s\rangle}$$

(neg) $$\frac{\langle b\,,\,s\rangle \longrightarrow \langle \texttt{false}\,,\,s\rangle}{\langle b \rightarrow c\,,\,s\rangle \longrightarrow \texttt{fail}}$$

(par1) $$\frac{\langle gc_0\,,\,s\rangle \longrightarrow \langle c\,,\,s'\rangle}{\langle gc_0 \,[\!]\, gc_1\,,\,s\rangle \longrightarrow \langle c\,,\,s'\rangle}$$

(par2) $$\frac{\langle gc_1\,,\,s\rangle \longrightarrow \langle c\,,\,s'\rangle}{\langle gc_0 \,[\!]\, gc_1\,,\,s\rangle \longrightarrow \langle c\,,\,s'\rangle}$$

(par3) $$\frac{\langle gc_0\,,\,s\rangle \longrightarrow \texttt{fail} \qquad \langle gc_1\,,\,s\rangle \longrightarrow \texttt{fail}}{\langle gc_0 \,[\!]\, gc_1\,,\,s\rangle \longrightarrow \texttt{fail}}$$

GCL – Semantics
Commands

- **skip** and sequencing ; as before (can drop determinacy)
- **abort** has no rules

(cond)
$$\frac{\langle gc\,,\ s\rangle \longrightarrow \langle c\,,\ s'\rangle}{\langle \textbf{if}\ gc\ \textbf{fi}\,,\ s\rangle \longrightarrow \langle c\,,\ s'\rangle}$$

(loop1)
$$\frac{\langle gc\,,\ s\rangle \longrightarrow \texttt{fail}}{\langle \textbf{do}\ gc\ \textbf{od}\,,\ s\rangle \longrightarrow \langle\!\langle s\rangle\!\rangle \quad {}^\dagger}$$

(loop2)
$$\frac{\langle gc\,,\ s\rangle \longrightarrow \langle c\,,\ s'\rangle}{\langle \textbf{do}\ gc\ \textbf{od}\,,\ s\rangle \longrightarrow \langle c\ ;\ \textbf{do}\ gc\ \textbf{od}\,,\ s'\rangle}$$

† new notation: behaves like **skip**

Processes

$$\textbf{do } b_1 \to c_1 \;[\!]\; \cdots \;[\!]\; b_n \to c_n \textbf{ od}$$

- form of (nondeterministically interleaved) parallel composition
- each c_i occurs atomically (uninterruptedly),
  provided b_i holds each time it starts

Some languages support/are based on GCL

- UNITY (Misra and Chandy)
- Hardware languages (Staunstrup)

GCL – Examples

- compute the maximum of x and y

if

$$x \geq y \rightarrow \max := x$$

$$[\!]$$

$$y \geq x \rightarrow \max := y$$

fi

- Euclid's algorithm

do

$$x > y \rightarrow x := x - y$$

$$[\!]$$

$$y > x \rightarrow y := y - x$$

od

GCL and Floyd-Hoare logic

guarded commands support a neat Hoare logic and decorated programs

Hoare triple for Euclid

$$\{x = m \wedge y = n \wedge m > 0 \wedge n > 0\}$$

Euclid

$$\{x = y = \mathrm{gcd}(m, n)\}$$

Proving Euclid's Algorithm Correct

- recall $\gcd(m,n)|m$, $\gcd(m,n)|n$ and

$$\ell|m,n \Rightarrow \ell|\gcd(m,n)$$

- invariant: $\gcd(m,n) = \gcd(x,y)$
- key properties:

$$\begin{aligned} \gcd(m,n) &= \gcd(m-n,n) && \text{if } m > n \\ \gcd(m,n) &= \gcd(m,n-m) && \text{if } n > m \\ \gcd(m,m) &= m \end{aligned}$$

Synchronised Communication

- communication by “handshake”
- possible exchange of value
  (localised to process-process (CSP) or to a channel (CCS))
- abstracts from the protocol underlying coordination
- invented by Hoare (CSP) and Milner (CCS)

Extending GCL

- allow processes to send and receive values on channels
  $\alpha!a$ evaluate expression a and send value on channel α
  $\alpha?x$ receive value on channel α and store it in x
- all interactions between parallel processes is by sending / receiving values on channels
- communication is synchronised (no broadcast yet)
- allow send and receive in commands c and in guards g:

$$\textbf{do } y < 100 \wedge \alpha?x \ \rightarrow \ \alpha!(x \cdot x) \parallel y := y + 1 \textbf{ od}$$

Extending GCL – Semantics

transitions may carry labels when possibility of interaction

$$\frac{}{\langle \alpha?x\,,\,s\rangle \xrightarrow{\alpha?n} \langle\!\langle s + \{x \mapsto n\}\rangle\!\rangle} \qquad \frac{\langle a\,,\,s\rangle \longrightarrow \langle n\,,\,s\rangle}{\langle \alpha!a\,,\,s\rangle \xrightarrow{\alpha!n} \langle\!\langle s\rangle\!\rangle}$$

$$\frac{\langle c_0\,,\,s\rangle \xrightarrow{\lambda} \langle c_0'\,,\,s'\rangle}{\langle c_0 \parallel c_1\,,\,s\rangle \xrightarrow{\lambda} \langle c_0' \parallel c_1\,,\,s'\rangle}$$ (+ symmetric)

$$\frac{\langle c_0\,,\,s\rangle \xrightarrow{\alpha?n} \langle c_0'\,,\,s'\rangle \qquad \langle c_1\,,\,s\rangle \xrightarrow{\alpha!n} \langle c_1'\,,\,s\rangle}{\langle c_0 \parallel c_1\,,\,s\rangle \longrightarrow \langle c_0' \parallel c_1'\,,\,s'\rangle}$$ (+ symmetric)

$$\frac{\langle c\,,\,s\rangle \xrightarrow{\lambda} \langle c'\,,\,s'\rangle}{\langle c\backslash\alpha\,,\,s\rangle \xrightarrow{\lambda} \langle c'\backslash\alpha\,,\,s'\rangle}\ \lambda \notin \{\alpha?n, \alpha!n\}$$

λ may be the empty label

Examples

- forwarder:

$$\textbf{do } \alpha?x \rightarrow \beta!x \textbf{ od}$$

- buffer of capacity 2:

$$\begin{array}{ll} (& \textbf{do } \alpha?x \rightarrow \beta!x \textbf{ od} \\ \| & \textbf{do } \beta?x \rightarrow \gamma!x \textbf{ od })\backslash\beta \end{array}$$

External vs Internal Choice

the following two processes are *not* equivalent w.r.t. deadlock capabilities

$$\textbf{if}\ (\texttt{true} \wedge \alpha?x \rightarrow c_0)\ [\!]\ (\texttt{true} \wedge \beta?x \rightarrow c_1)\ \textbf{fi}$$

$$\textbf{if}\ (\texttt{true} \rightarrow \alpha?x\ ;\ c_0)\ [\!]\ (\texttt{true} \rightarrow \beta?x\ ;\ c_1)\ \textbf{fi}$$

Section 20

The Process Algebra CCS

Towards an Abstract Mechanism for Concurrency

The Calculus of Communicating Systems (CCS)

- introduced by Robin Milner in 1980
- first process calculus developed with its operational semantics
- supports algebraic reasoning about equivalence
- simplifies Dijkstra's GCL by removing the store

Actions and Communications

- processes communicate values (numbers) on channels
- communication is synchronous and between two processes
- a is an arithmetic expression; evaluation is written $a \to n$
- input: $\alpha?x$
- output $\alpha!a$
- *silent* actions τ (internal to a process)
- λ will range over all the kinds of actions, including τ

(Decorated) CCS – Syntax

Expressions:

arithmetic a and Boolean b

Processes:

p ::= **nil**	nil process
$\mid\ (\tau \to p)$	silent/internal action
$\mid\ (\alpha!a \to p)$	output
$\mid\ (\alpha?x \to p)$	input
$\mid\ (b \to p)$	Boolean guard
$\mid\ p + p$	nondeterministic choice
$\mid\ p \parallel p$	parallel composition
$\mid\ p \backslash L$	restriction (L a set of channel identifiers)
$\mid\ p[f]$	relabelling (f a function on channel identifiers)
$\mid\ P(a_1, \ldots, a_k)$	process identifier

(Decorated) CCS – Syntax

Process Definitions:

$$P(x_1, \ldots, x_k) \overset{\text{def}}{=} p$$

(free variables of $p \subseteq \{x_1, \ldots, x_k\}$)

Restriction and Relabelling – Examples

- $p\backslash L$: disallow *external* interaction on channels in L
- $p[f]$: rename *external* interface to channels by f

Operational semantics of CCS

Guarded processes

silent action

$$(\tau \to p) \xrightarrow{\tau} p$$

output

$$\frac{a \longrightarrow n}{(\alpha!a \to p) \xrightarrow{\alpha!n} p}$$

input

$$(\alpha?x \to p) \xrightarrow{\alpha?n} p[n/x]$$

Boolean

$$\frac{b \to \mathtt{true} \qquad p \xrightarrow{\lambda} p'}{(b \to p) \xrightarrow{\lambda} p'}$$

Operational semantics of CCS

Sum

$$\frac{p_0 \xrightarrow{\lambda} p_0'}{p_0 + p_1 \xrightarrow{\lambda} p_0'} \qquad \frac{p_1 \xrightarrow{\lambda} p_1'}{p_0 + p_1 \xrightarrow{\lambda} p_1'}$$

Parallel composition

$$\frac{p_0 \xrightarrow{\lambda} p_0'}{p_0 \parallel p_1 \xrightarrow{\lambda} p_0' \parallel p_1} \qquad \frac{p_0 \xrightarrow{\alpha?n} p_0' \quad p_1 \xrightarrow{\alpha!n} p_1'}{p_0 \parallel p_1 \xrightarrow{\tau} p_0' \parallel p_1'}$$

$$\frac{p_1 \xrightarrow{\lambda} p_1'}{p_0 \parallel p_1 \xrightarrow{\lambda} p_0 \parallel p_1'} \qquad \frac{p_0 \xrightarrow{\alpha!n} p_0' \quad p_1 \xrightarrow{\alpha?n} p_1'}{p_0 \parallel p_1 \xrightarrow{\tau} p_0' \parallel p_1'}$$

Operational semantics of CCS

Restriction

$$\frac{p \xrightarrow{\lambda} p'}{p\backslash L \xrightarrow{\lambda} p'\backslash L} \quad \text{if } \lambda \in \{\alpha?n, \alpha!n\} \text{ then } \alpha \notin L$$

Relabelling

$$\frac{p \xrightarrow{\lambda} p'}{p[f] \xrightarrow{f(\lambda)} p'[f]}$$

where f is extended to labels as $f(\tau) = \tau$ and $f(\alpha?n) = f(\alpha)?n$ and $f(\alpha!n) = f(\alpha)!n$

Identifiers

$$\frac{p[a_1/x_1, \ldots, a_n/x_n] \xrightarrow{\lambda} p'}{P(a_1, \ldots, a_n) \xrightarrow{\lambda} p'} \; P(x_1, \ldots, x_n) \stackrel{\text{def}}{=} p$$

Nil process

no rules

A Derivation

$$(((\alpha!3 \to \textbf{nil} + P) \parallel \tau \to \textbf{nil}) \parallel \alpha?x \to \textbf{nil})\backslash\{\alpha\} \xrightarrow{\tau} ((\textbf{nil} \parallel \tau \to \textbf{nil}) \parallel \textbf{nil})\backslash\{\alpha\}$$

More Examples

- Mixed choice

$$
\begin{tikzcd}[column sep=small]
 & \alpha!2 \rightarrow \textbf{nil} + \tau \rightarrow \beta!3 \rightarrow \textbf{nil} \arrow[dl, "\alpha!2"'] \arrow[dr, "\tau"] & \\
\textbf{nil} & & \beta!3 \rightarrow \textbf{nil} \arrow[d, "\beta!3"] \\
 & & \textbf{nil}
\end{tikzcd}
$$

Linking Process

(some syntactic sugar)

Let

$$P \stackrel{\text{def}}{=} in?x \rightarrow out!x \rightarrow P$$

$$Q \stackrel{\text{def}}{=} in?y \rightarrow out!y \rightarrow Q$$

Connect P's output port to Q's input port

$$P \frown Q = (P[c/out] \parallel Q[c/in]) \backslash \{c\}$$

where c is a *fresh* channel name

Euclid's algorithm in CSS

$$
\begin{aligned}
E(x,y) \stackrel{\textsf{def}}{=} \quad & \quad x = y \to \mathrm{gcd}!x \to \textbf{nil} \\
& + x < y \to E(x, y-x) \\
& + y < x \to E(x-y, x)
\end{aligned}
$$

$$
\mathit{Euclid} \stackrel{\textsf{def}}{=} \mathit{in}?x \to \mathit{in}?y \to E(x,y)
$$

Section 21

Pure CCS

Towards a more basic language

aim: removal of variables to reveal symmetry of input and output

- transitions for value-passing carry labels τ, $a?n$, $a!n$

$$\begin{tikzcd} \alpha?x \to p \arrow[r, "\alpha?0"] \arrow[dr, "\alpha?n"'] & p[0/x] \arrow[d, dotted, no head] \\ & p[n/x] \arrow[d, dotted, no head] \\ & {} \end{tikzcd}$$

- this suggests introducing *prefix* $\alpha?n.p$ (as well as $\alpha!n.p$) and view $\alpha?x \to p$ as a *(infinite) sum* $\sum_n \alpha?n.p[n/x]$
- view $\alpha?n$ and $\alpha!n$ as *complementary* actions
- synchronisation can only occur on complementary actions

Pure CCS

- Actions: a, b, c, …
- Complementary actions: $\bar{a}$, $\bar{b}$, $\bar{c}$,…
- Internal action: τ
- Notational convention: $\bar{\bar{a}} = a$
- Processes:

p ::= $\lambda.p$	prefix	λ ranges over $\tau, a, \bar{a}$ for any action
$\mid\ \sum_{i \in I} p_i$	sum	I is an index set
$\mid\ p_0 \parallel p_1$	parallel	
$\mid\ p \backslash L$	restriction	L a set of actions
$\mid\ p[f]$	relabelling	f a relabelling function on actions
$\mid\ P$		process identifier

- Process definitions:

$$P \overset{\text{def}}{=} p$$

Pure CCS – Semantics

Guarded processes (prefixing)

$$\lambda.p \xrightarrow{\lambda} p$$

Sum

$$\frac{p_j \xrightarrow{\lambda} p'}{\sum_{i\in I} p_i \xrightarrow{\lambda} p'} \; j \in I$$

Parallel composition

$$\frac{p_0 \xrightarrow{\lambda} p_0'}{p_0 \parallel p_1 \xrightarrow{\lambda} p_0' \parallel p_1} \qquad \frac{p_1 \xrightarrow{\lambda} p_1'}{p_0 \parallel p_1 \xrightarrow{\lambda} p_0 \parallel p_1'}$$

$$\frac{p_0 \xrightarrow{a} p_0' \qquad p_1 \xrightarrow{\bar{a}} p_1'}{p_0 \parallel p_1 \xrightarrow{\tau} p_0' \parallel p_1'}$$

Pure CCS – Semantics

Restriction

$$\frac{p \xrightarrow{\lambda} p'}{p \backslash L \xrightarrow{\lambda} p' \backslash L} \; \lambda \notin L \cup \overline{L}$$

where $\overline{L} = \{\bar{a} \mid a \in L\}$

Relabelling

$$\frac{p \xrightarrow{\lambda} p'}{p[f] \xrightarrow{\lambda} p'[f]}$$

where f is a function such that $f(\tau) = \tau$ and $f(\bar{a}) = \overline{f(a)}$

Identifiers

$$\frac{p \xrightarrow{\lambda} p'}{P \xrightarrow{\lambda} p'} \; P \overset{\text{def}}{=} p$$

From Value-passing to Pure CCS

translation from a value-passing CCS *closed* term p to a pure CCS term $\widehat{p}$

p	$\widehat{p}$	
nil	**nil**	
$(\tau \rightarrow p)$	$\tau.\widehat{p}$	
$(\alpha!a \rightarrow p)$	$\overline{\alpha m}.\widehat{p}$	where a evaluates to m
$(\alpha?x \rightarrow p)$	$\sum_{m\in\text{int}} \alpha m.\widehat{p[m/x]}$	
$(b \rightarrow p)$	$\widehat{p}$	if b evaluates to true
	nil	if b evaluates to false
$p_0 + p_1$	$\widehat{p}_0 + \widehat{p}_1$	
$p_0 \parallel p_1$	$\widehat{p}_0 \parallel \widehat{p}_1$	
$p\backslash L$	$\widehat{p}\backslash\{\alpha m \mid \alpha \in L \wedge m \in \text{int}\}$	
$P(a_1, \ldots, a_k)$	$P_{m_1,\ldots,m_k}$	where a_i evaluates to m_i

For every definition $P(x_1, \ldots, x_k)$ we have a collection of definitions $P_{m_1,\ldots,m_k}$ indexed by $m_1,\ldots,m_k \in$ int

Correspondence

Theorem

$$p \xrightarrow{\lambda} p' \quad \textit{iff} \quad \widehat{p} \xrightarrow{\widehat{\lambda}} \widehat{p'}$$

Section 22

Semantic Equivalences

Labelled Transition Systems

CCS naturally implies a graphical model of computation.

a **labelled transition system** (LTS) is a pair $(S, \Rightarrow)$ with

- S a set (of *states* or processes), and
- $\Rightarrow \subseteq S \times Act \times S$, the *transition relation*.

here $Act = A \uplus \{\tau\}$ is a set of *actions*, containing visible actions $a, b, c, ... \in A$, and the *invisible action* τ.

a finite *path* is a sequence $p_0 \xrightarrow{\lambda_1} p_1 \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_n} p_n$ with $p_i \in S$ for $i = 0, ..., n$ and $(p_{i-1}, \lambda_i, p_i) \in \Rightarrow$ for all $i = 1, ..., n$.

Trace equivalence

- if such a path exists, then the sequence $\lambda_1\lambda_2 \ldots \lambda_n$ is a (partial) *trace* of the process p_0
- two processes p and q are (partial) *trace equivalent* if they have the same (partial) traces.

Australian National University

Four Kinds of Trace Equivalence

Let $T^*(p)$ be the set of (partial) traces of process $p \in S$.
Let $T^\infty(p)$ be the set of infinite traces of p.
Let $CT^*(p)$ be the set of completed traces of p.
Let $CT^\infty(p) := CT^*(p) \uplus T^\infty(p)$.

A finite trace is *complete* if it last state has no outgoing transition.

Write $p =_T^* q$ if $T^*(p) = T^*(q)$ — (partial) trace equivalence.
Write $p =_{CT}^* q$ if $CT^*(p) = CT^*(q)$ and $T^*(p) = T^*(q)$ — completed trace equivalence
Write $p =_T^\infty q$ if $T^\infty(p) = T^\infty(q)$ and $T^*(p) = T^*(q)$ — infinitary trace equivalence
Write $p =_{CT}^\infty$ if $CT^\infty(p) = CT^\infty(q)$ — infinitary completed tr. eq.

A Lattice of Semantic Equivalence Relations

A relation $\sim \subseteq S \times S$ on processes is an *equivalence relation* if it is

- *reflexive*: $p \sim p$,
- *symmetric*: if $p \sim q$ then $q \sim p$,
- and *transitive*: if $p \sim q$ and $q \sim r$ then $p \sim r$.

Let $[p]_\sim$ be the *equivalence class* of p: the set of all processes that are $\sim$-equivalent to p.

$$[p]_\sim := \{q \in S \mid q \sim p\}.$$

Equivalence relation $\sim$ is *finer than* equivalence relation $\approx$ iff

$$p \sim q \Rightarrow p \approx q.$$

Thus if $\sim \subseteq \approx$. In that case each equivalence class of $\sim$ is included in an equivalence class of $\approx$.

Four Additional Trace Equivalence

A *weak* trace is obtained from a strong one by deleting all τs.
Let $WT^*(p) := \{$*detau*$(\sigma) \mid \sigma \in T^*(p)\}$.

This leads to *weak trace equivalences* $=^*_{WT}$, $=^\infty_{WT}$, $=^*_{WCT}$, $=^\infty_{WCT}$.

Safety and Liveness Properties

A **safety** property says that something bad will never happen.
A **liveness** property says that something good will happen eventually.

If we deem two processes p and q semantically equivalent we often want them to have the same safety and/or liveness properties.

$$ab \overset{?}{\sim} ab + a$$

Weak partial trace equivalence respects safety properties.

$$ag \overset{?}{\sim} ag + a$$

We need at least completed traces to deal with liveness properties

Compositionality

If $p \sim q$ then $C[p] \sim C[q]$.
Here $C[\ ]$ is a context, made from operators of some language.

For instance $(\_|\bar{b}.\bar{a}.\mathbf{nil})\backslash\{a,b\}$ is a CCS-context.
If $p \sim q$ then $(p|\bar{b}.\bar{a}.\mathbf{nil})\backslash\{a,b\} \sim (q|\bar{b}.\bar{a}.\mathbf{nil})\backslash\{a,b\}$.

Then $\sim$ is a *congruence* for the language,
or the language if *compositional* for $\sim$.

$$p \sim p' \quad \Rightarrow \quad (p|p|...|p)\backslash L \sim (p'|p'|...|p')\backslash L.$$

$a.b + a.c =^*_{CT} a.(b+c)$ but
$((a.b + a.c)|\bar{a}.\bar{b})\backslash\{a,b\} \neq^*_{CT} (a.(b+c)|\bar{a}.\bar{b})\backslash\{a,b\}$.

Thus $=^*_{CT}$ is a not a congruence for CCS.

Congruence closure

Theorem: Given any equivalence $\approx$ that need not be a congruence for some language $\mathcal{L}$, there exists a coarsest congruence $\sim$ for $\mathcal{L}$ that is finer than $\sim$.

In fact, $\sim$ can be defined by

$$p \sim q \quad :\Leftrightarrow \quad C[p] \approx C[q] \text{ for any } \mathcal{L}\text{-context } C[\ \ ].$$

Bisimulation equivalence

A relation $\mathcal{R} \subseteq S \times S$ is a *bisimulation* if it satisfies:

- if $p\mathcal{R}q$ and $p \xrightarrow{\lambda} p'$ then $\exists q'$ s.t. $q \xrightarrow{\lambda} q'$ and $p'\mathcal{R}q'$, and
- if $p\mathcal{R}q$ and $q \xrightarrow{\lambda} q'$ then $\exists p'$ s.t. $p \xrightarrow{\lambda} p'$ and $p'\mathcal{R}q'$.

Two processes $p, q \in S$ are *bisimulation equivalent* or *bisimilar* —notation $p =_B q$—if $p\mathcal{R}q$ for some bisimulation $\mathcal{R}$.

Examples: $a.b + a.c \neq_B a.(b+c)$ $\qquad$ $a.b + a.b =_B a.b$

Weak bisimulation equivalence

A relation $\mathcal{R} \subseteq S \times S$ is a *weak bisimulation* if it satisfies:

- if $p\mathcal{R}q$ and $p \xrightarrow{\lambda} p'$ then $\exists q'$ s.t. $q \Longrightarrow\xrightarrow{(\lambda)}\Longrightarrow q'$ and $p'\mathcal{R}q'$, and
- if $p\mathcal{R}q$ and $q \xrightarrow{\lambda} q'$ then $\exists p'$ s.t. $p \Longrightarrow\xrightarrow{(\lambda)}\Longrightarrow p'$ and $p'\mathcal{R}q'$.

Here $\Longrightarrow$ denotes a finite sequence of τ-steps,
and (λ) means λ, except that it is optional in case $\lambda = \tau$.
(That is, $p \xrightarrow{(\lambda)} q$ iff $p \xrightarrow{\lambda} q \vee (\lambda = \tau \wedge q = p)$.)
Two processes $p, q \in S$ are *weakly bisimilar*
—notation $p =_{WB} q$—if $p\mathcal{R}q$ for some bisimulation $\mathcal{R}$.

Examples: $\tau.b + c \neq_{WB} b + c$ $\qquad$ $\tau.b + b =_{WB} b$

Semantic Equivalences – Summary

- relate to systems (via LTSs)
- can be extended to states carrying stores
- sos-rules give raise to LTSs in a straightforward way
- reduce complicated (big) systems to simpler ones
- smaller systems may be easier to verify
- understand which properties are preserved

Section 23

The Owicki-Gries Method

Motivation

- nondeterminism and concurrency required
- handle interleaving
- Floyd-Hoare logic only for sequential programs

- Owicki-Gries Logic/Method
  - a.k.a. interference freedom
  - Susan Owicki and PhD supervisor David Gries
  - add a construct to the programming language for threads
  - study the impact for Hoare triples

Floyd-Hoare Logic and Decorated Programs

Notation: *processes*: individual program
system: overall (concurrent) program will be

Floyd-Hoare logic

- each of the individual processes has an assertion
  - before its first statement (precondition)
  - between every pair of its statements (pre-/postcondition), and
  - after its last statement (postcondition)
- Hoare-triples can be checked (local correctness)
- Floyd-Hoare logic is compositional

Motivation

add pre- and postcondition for system, and a rule

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \qquad \{P_2\}\ c_2\ \{Q_2\}}{\{P_1 \wedge P_2\}\ c_1 \parallel c_2\ \{Q_1 \wedge Q_2\}}$$

but this rule is incorrect

Note: we are considering an interleaving semantics

Simple Example

$$\{x == 0\}$$

$$\begin{array}{ccc} \{x == 0 \lor x == 2\} & & \{x == 0 \lor x == 1\} \\ x := x + 1 & \Big\| & x := x + 2 \\ \{x == 1 \lor x == 3\} & & \{x == 2 \lor x == 3\} \end{array}$$

$$\{x == 3\}$$

What would we have to show?

The Rule of Owicki Gries

all rules of Floyd-Hoare logic remain valid

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \ldots \{P_n\}\ c_n\ \{Q_n\} \qquad \textit{interference freedom}}{\{P_1 \wedge \cdots \wedge P_n\}\ c_1 \parallel \cdots \parallel c_n\ \{Q_1 \wedge \cdots \wedge Q_n\}}\ \text{(par)}$$

Interference Freedom

Interference freedom is a property of proofs of the $\{P_i\}\ c_i\ \{Q_i\}$

- suppose we have a proof for $\{P_i\}\ c_i\ \{Q_i\}$
- prove that the execution of any other statement c_j does not validate the reasoning for $\{P_i\}\ c_i\ \{Q_i\}$

it is a bit tricky

- interference freedom is a property of *proofs*, not Hoare triples
- identifying which parts of a proof need to be considered requires some effort

Australian National University

Formalising Interference Freedom

In a decorated program D and command c of the program, let

- pre(D, c) be the precondition (assumption/predicate) immediately before c, and
- post(D, c) the postcondition immediately after c
- remember $\{P\}\ c\ \{Q\}$ valid if there is a decorated program D with pre$(D, c) = P$ and post$(D, c) = Q$

Formalising Interference Freedom

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \ldots \{P_n\}\ c_n\ \{Q_n\} \qquad \textit{interference freedom}}{\{P_1 \wedge \cdots \wedge P_n\}\ c_1 \parallel \cdots \parallel c_n\ \{Q_1 \wedge \cdots \wedge Q_n\}}\ \text{(par)}$$

Suppose every c_i has a decorated program D_{c_i}.

Definition

D_{c_i} is *interference-free* with respect to D_{c_j} $(i \neq j)$ if for each statement c_i' in c_i and c_j' in c_j

- $\{\text{pre}(D_{c_i}, c_i') \wedge \text{pre}(D_{c_j}, c_j')\}\ c_j'\ \{\text{pre}(D_{c_i}, c_i')\}$
- $\{\text{post}(D_{c_i}, c_i') \wedge \text{pre}(D_{c_j}, c_j')\}\ c_j'\ \{\text{post}(D_{c_i}, c_i'))\}$

The $D_{c_1}, D_{c_1}, \ldots D_{c_n}$ are interference-free if they are pairwise interference-free with respect to one other.

Interference Freedom – Remark

- applying the Rule (par) requires the development of interference-free decorated programs for the c_i
- proving interference-freedom of D_{c_i} with respect to D_{c_j} focusses on
  - preconditions of each statement in c_i and postcondition of D_{c_i}

Simple Example

Why is interference freedom violated?

$$\{x == 0\}$$

$$\begin{array}{ccc} \{x == 0\} & & \{x == 0\} \\ x := x + 1 & \Big\| & x := x + 2 \\ \{x == 1\} & & \{x == 1\} \end{array}$$

$$\{x == 1\}$$

Soundness

Theorem

If $\{P\}\ c\ \{Q\}$ *is derivable using the proof rules seen so far then* c *is valid*

Completeness

Can every correct Hoare triple be derived?

- completeness does not hold
- neither does relative completeness

Incompleteness

Lemma

The following valid Hoare triple cannot be derived using the rules so far.

$$\{\texttt{true}\} \quad x := x + 2 \parallel x := 0 \quad \{x == 0 \vee x == 2\}$$

Proof.

By contradiction. Suppose there were such a proof. Then there would be Q, R such that

$$\begin{aligned} \{\texttt{true}\}\ x := x + 2\ \{Q\} \\ \{\texttt{true}\}\ x := 0\ \{R\} \\ Q \wedge R \implies x == 0 \vee x == 2 \end{aligned}$$

By (assign) $\big(\{P[a/l]\}\ l := a\ \{P\}\big)$, $\texttt{true} \implies Q[x + 2/x]$ holds. Similarly, $R[0/x]$ holds. By (par), $\{R \wedge \texttt{true}\}\ x := x + 2\ \{R\}$ holds, meaning $R \Rightarrow R[x + 2/x]$ is valid. But then by induction, $\forall x.\ (x \geq 0 \wedge \text{even}(x)) \implies R$ is true. Since $Q \wedge R \implies x = 0 \vee x = 2$, it follows that

$$\forall x.\ (x \geq 0 \wedge \text{even}(x)) \implies (x == 0 \vee x == 2)\ ,$$

which is a contradiction. □

Fixing the Problem

We showed

- R must hold for all even, positive x
- R must hold after execution of $x := 0$
- R must also hold both before and after execution of $x := x + 2$

we need the capability in R to say that
until $x := x + 2$ is executed, $x = 0$ holds.

Auxiliary Variables

variables that are put into a program just to reason about progress in other processes

$$
\begin{array}{l}
\text{done} := 0\ ; \\
(\\
\qquad x, \text{done} := x + 2, 1 \\
\| \\
\qquad x := 0 \\
)
\end{array}
$$

- requires synchronous/atomic assignment
- proof is now possible

Decorated Programs with Auxiliary Variables

$\{\texttt{true}\}$
done $:= 0$;
$\{\text{done} == 0\}$
(
    $\{\text{done} == 0\}$
    x, done $:= x + 2, 1$
    $\{\texttt{true}\}$
$\|$
    $\{\texttt{true}\}$
    $x := 0$
    $\{(x == 0 \vee x == 2) \wedge (\text{done} == 0 \Rightarrow x == 0)\}$
)
$\{c == 0 \vee x == 2\}$

Note: some implications skipped in the decorated program

Relative Completeness

- adding auxiliary variables enables proofs
- we do not want these variables to be in our code

$$\frac{\{P\}\ c\ \{Q\} \qquad x \text{ not free in } Q \qquad x \text{ auxiliary in } c}{\{P\}\ c'\ \{Q\}}\ (aux)$$

where c' is c with all references to x removed.

Theorem (Relative Completeness)

Adding Rules (par) and (aux) to the other rules of Floyd-Hoare logic yields a relatively complete proof system.

Problem

The Owicki-Griess Methods is *not compositional*.

Peterson's Algorithm for Mutual exclusion

the following 4 lines of (symmetric) code took 15 years to discover (mid 60's to early 80s)

let a, b be Booleans and $t : \{A, B\}$

$$\{\neg a \wedge \neg b\}$$

other code of A	other code of B
$a := \texttt{true}$	$b := \texttt{true}$
$t := A$	$t := B$
await $(\neg b \vee t == B)$	**await** $(\neg a \vee t == A)$
critical section A	critical section B
$a := \texttt{false}$	$b := \texttt{false}$

Notes on Peterson's Algorithm

- protects critical sections from mutual destructive interference
- guarantees fair treatment of A and B

- how do we show that A (or B) is never perpetually ignored in favour of B (A)?
  - requires *liveness* in this case
  - a topic for another course/research project
  - in fact there is one line that could potentially violate liveness (requires knowledge about hardware)

- 4 correct lines of code in 15 years is a coding rate of roughly

  1 LoC every 4 years

Yet Another Example

FindFirstPositive

$$i := 0 \ ; \ j := 1 \ ; \ x := |A| \ ; \ y := |A| \ ;$$

$$\left(\begin{array}{l} \textbf{while } i < \min(x,y) \textbf{ do} \\ \quad \textbf{if } A[i] > 0 \textbf{ then} \\ \quad\quad x := i \\ \quad \textbf{else} \\ \quad\quad i := i + 2 \end{array}\right) \ \Big|\Big| \ \left(\begin{array}{l} \textbf{while } j < \min(x,y) \textbf{ do} \\ \quad \textbf{if } A[j] > 0 \textbf{ then} \\ \quad\quad y := j \\ \quad \textbf{else} \\ \quad\quad j := j + 2 \end{array}\right)$$

$$r := \min(x,y)$$

Australian National University

$$i := 0 \; ; \; j := 1 \; ; \; x := |A| \; ; \; y := |A| \; ;$$
$$\{P_1 \wedge P_2\}$$

$\{P_1\}$
while $i < \min(x, y)$ **do**
  $\{P_1 \wedge i < x \wedge i < |A|\}$
  if $A[i] > 0$ **then**
    $\{P_1 \wedge i < x \wedge i < |A| \wedge A[i] > 0\}$
    $x := i$
    $\{P_1\}$
  else
    $\{P_1 \wedge i < x \wedge i < |A| \wedge A[i] \leq 0\}$
    $i := i + 2$
    $\{P_1\}$
  $\{P_1\}$
$\{P_1 \wedge i \geq \min(x, y)\}$

$||$

$\{P_2\}$
while $j < \min(x, y)$ **do**
  $\{P_2 \wedge j < y \wedge j < |A|\}$
  if $A[j] > 0$ **then**
    $\{P_2 \wedge j < y \wedge j < |A| \wedge A[j] > 0\}$
    $y := j$
    $\{P_2\}$
  else
    $\{P_2 \wedge j < y \wedge j < |A| \wedge A[j] \leq 0\}$
    $j := j + 2$
    $\{P_2\}$
  $\{P_2\}$
$\{P_2 \wedge j \geq \min(x, y)\}$

$$\{P_1 \wedge P_2 \wedge i \geq \min(x, y) \wedge j \geq \min(x, y)\}$$
$$r := \min(x, y)$$
$$\{r \leq |A| \wedge (\forall k.\ 0 \leq k < r \Rightarrow A[k] \leq 0) \wedge (r < |A| \Rightarrow A[r] > 0)\}$$

$P_1 = x \leq |A| \wedge (\forall k.\ 0 \leq k < i \wedge k \text{ even} \Rightarrow A[k] \leq 0) \wedge i \text{ even} \wedge (x < |A| \Rightarrow A[x] > 0)$

$P_2 = y \leq |A| \wedge (\forall k.\ 0 \leq k < j \wedge k \text{ odd} \Rightarrow A[k] \leq 0) \wedge j \text{ odd} \wedge (y < |A| \Rightarrow A[y] > 0)$

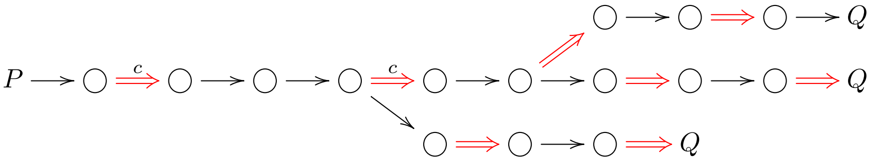
Section 24

Rely-Guarantee

Motivation

- Owicki-Gries is *not* compositional
- generalise it to make it compositional

$$\{P\}\ c \parallel E\ \{Q\}$$



Motivation

$$P \xrightarrow{*} \bigcirc \overset{c}{\Longrightarrow} \bigcirc \xrightarrow{*} \bigcirc \overset{c}{\Longrightarrow} \bigcirc \xrightarrow{*} Q$$

$\xrightarrow{*}$: any state transition that can be done by *any* other thread, repeated zero or more times

Rely-Guarantee

$$\{P, R\}\ c\ \{G, Q\}$$

If

- the initial state satisfies P, and
- every state change by another thread satisfies the *rely condition* R, and

then c is executed and terminates,

then

- every final state satisfies Q, and
- every state change in c satisfies the *guarantee condition* G.

Rely-Guarantee – Parallel Rule

$$\frac{\{P_1, R \vee G_2\}\ c_1\ \{G_1, Q_1\} \qquad \{P_2, R \vee G_1\}\ c_2\ \{G_2, Q_2\}}{\{P_1 \wedge P_2, R\}\ c_1 \parallel c_2\ \{G_1 \vee G_2, Q_1 \wedge Q_2\}}$$

Rely-Guarantee – Consequence Rule

$$\frac{R \Rightarrow R' \qquad \{P, R'\}\ c\ \{G', Q\} \qquad G' \Rightarrow G}{\{P, R\}\ c\ \{G, Q\}}$$

Note: both rules can be packed in a single rule.

From Floyd-Hoare to Rely-Guarantee

$$\frac{\{P\}\ c\ \{Q\} \qquad\qquad ???}{\{P, R\}\ c\ \{G, Q\}}$$

$$P \overbrace{\longrightarrow}^{R} P \overbrace{\longrightarrow}^{R} P \underbrace{\color{red}{\Longrightarrow}}_{G} Q \overbrace{\longrightarrow}^{R} Q \overbrace{\longrightarrow}^{R} Q \overbrace{\longrightarrow}^{R} Q$$

Back to Stores

$$\frac{\{P\}\ c\ \{Q\} \quad P \text{ stable under } R \quad Q \text{ stable under } R \quad c \text{ is contained in } G}{\{P, R\}\ c\ \{G, Q\}}$$

P stable under R: $\forall s, s'.\ P(s) \wedge R(s, s') \implies P(s')$

c contained in G: $\forall s, s'.\ P(s) \wedge (s, s') \in \mathcal{C}[\![c]\!] \implies G(s, s')$

Making Assertions Stable

Assume

$$
\begin{aligned}
R &= (x \mapsto n \ \rightsquigarrow \ x \mapsto n-1) \\
&= \{(s, s') \mid \exists n.\ s(x) = n \wedge s'(x) = s + \{x \mapsto n-1\}\} \\
G &= (x \mapsto n \ \rightsquigarrow \ x \mapsto n+1) \\
&= \{(s, s') \mid \exists n.\ s(x) = n \wedge s'(x) = s + \{x \mapsto n+1\}\}
\end{aligned}
$$

$$\{x == 2, R\}\ x := x + 1\ \{G, x == 3\}$$

Making Assertions Stable

Assume

$$
\begin{aligned}
R &= (x \mapsto n \ \rightsquigarrow \ x \mapsto n-1) \\
&= \{(s, s') \mid \exists n.\ s(x) = n \wedge s'(x) = s + \{x \mapsto n-1\}\} \\
G &= (x \mapsto n \ \rightsquigarrow \ x \mapsto n+1) \\
&= \{(s, s') \mid \exists n.\ s(x) = n \wedge s'(x) = s + \{x \mapsto n+1\}\}
\end{aligned}
$$

$$\{x \leq 2, R\}\ x := x + 1\ \{G, x \leq 3\}$$

FindFirstPositive

$$i := 0 \; ; \; j := 1 \; ; \; x := |A| \; ; \; y := |A| \; ;$$
$$\{P_1 \wedge P_2\}$$

$$\begin{array}{lcl}
\{P_1, \mathbf{G_2}\} & & \{P_2, \mathbf{G_1}\} \\
\textbf{while } i < \min(x,y) \textbf{ do} & & \textbf{while } j < \min(x,y) \textbf{ do} \\
\quad \{P_1 \wedge i < x \wedge i < |A|\} & \Big|\Big| & \quad \{P_2 \wedge j < y \wedge j < |A|\} \\
\quad \ldots & & \quad \ldots \\
\quad \{P_1\} & & \quad \{P_2\} \\
\{\mathbf{G_1}, P_1 \wedge i \geq \min(x,y)\} & & \{\mathbf{G_2}, P_2 \wedge j \geq \min(x,y)\}
\end{array}$$

$$\{P_1 \wedge P_2 \wedge i \geq \min(x,y) \wedge j \geq \min(x,y)\}$$
$$r := \min(x,y)$$
$$\{r \leq |A| \wedge (\forall k.\ 0 \leq k < r \Rightarrow A[k] \leq 0) \wedge (r < |A| \Rightarrow A[r] > 0)\}$$

$$P_1 = x \leq |A| \wedge (\forall k.\ 0 \leq k < i \wedge k \text{ even} \Rightarrow A[k] \leq 0) \wedge i \text{ even} \wedge (x < |A| \Rightarrow A[x] > 0)$$
$$P_2 = y \leq |A| \wedge (\forall k.\ 0 \leq k < j \wedge k \text{ odd} \Rightarrow A[k] \leq 0) \wedge j \text{ odd} \wedge (y < |A| \Rightarrow A[y] > 0)$$
$$G_1 = \{(s, s') | s'(y) = s(y) \wedge s'(j) = s(j) \wedge s'(x) \leq s(x)\}$$
$$G_2 = \{(s, s') | s'(x) = s(x) \wedge s'(i) = s(i) \wedge s'(y) \leq s(y)\}$$

Rely-Guarantee Abstraction

Forgets

- which thread performs the action
- in what order the actions are performed
- how many times the action is performed

Usually, this is fine. . .

Verify This

$$\{x == 0\}$$

$$\begin{array}{ccc} \{x == 0 \vee x == 1\} & & \{x == 0 \vee x == 1\} \\ x := x + 1 & \Big\| & x := x + 1 \\ \{x == 1 \vee x == 2\} & & \{x == 1 \vee x == 2\} \end{array}$$

$$\{x == 2\}$$

$$G_1, G_2 = (x \mapsto n \ \rightsquigarrow\ x \mapsto n + 1)$$

Verify This

$$\{x == 0\}$$

$$\{\exists n \geq 0.\ x \mapsto n, \mathbf{G_2}\} \qquad\qquad \{\exists n \geq 0.\ x \mapsto n, \mathbf{G_1}\}$$

$$x := x + 1 \qquad \Big|\Big| \qquad x := x + 1$$

$$\{\mathbf{G_1}, \exists n \geq 1.\ x \mapsto n\} \qquad\qquad \{\mathbf{G_2}, \exists n \geq 1.\ x \mapsto n\}$$

$$\{\exists n \geq 1.\ x \mapsto n\}$$

$$G_1, G_2 = (x \mapsto n \ \rightsquigarrow \ x \mapsto n + 1)$$

From Floyd-Hoare to Rely-Guarantee (recap)

$$\frac{\{P\}\ c\ \{Q\} \qquad\qquad ???}{\{P, R\}\ c\ \{G, Q\}}$$

P stable under R if and only if $\{P\}\ R^*\ \{P\}$

$$P \overbrace{\longrightarrow}^{R} P \overbrace{\longrightarrow}^{R} P \underbrace{\color{red}{\Longrightarrow}}_{G} Q \overbrace{\longrightarrow}^{R} Q \overbrace{\longrightarrow}^{R} Q \overbrace{\longrightarrow}^{R} Q$$

Section 25

Conclusion

Learning Outcome I

1. Understand the role of theoretical formalisms, such as operational and denotational semantics

   - IMP language
   - operational semantics
   - denotational semantics
   - axiomatic semantics
   - functions
     (call-by-name, call-by-value)
   - references
   - extensions
     (data structures, error handling, object-orientation,. . .)

Learning Outcome II

2. Apply these semantics in the context of programming languages

   - IMP language + extensions
   - configurations
   - derivations
   - transitions

Learning Outcome III

3. Evaluate differences (advantages/disadvantages) of these theoretical formalisms
   - small-step vs big-step
   - operational vs denotational vs axiomatic (vs algebraic)

Learning Outcome IV

4. Create operational or denotational semantics of simple imperative programs

    - IMP + extensions + types
    - derivations
    - transitions

Learning Outcome V

5. Analyse the role of types in programming languages

    - types
    - subtypes
    - progress and preservation properties
    - Curry-Howard correspondence

Learning Outcome VI

6. Formalise properties and reason about programs

   - Isabelle/HOL
   - semantic equivalences
   - decorated programs
   - Floyd-Hoare logic, wlp
   - Owicki-Gries, Rely-Guarantee

Learning Outcome VII

7. Apply basic principles for formalising concurrent programming languages

   - Guarded Command Language
   - process algebra
     (value-passing CCS and pure CCS)
   - semantic equivalences
   - Owicki-Gries, Rely-Guarantee

Learning Outcome VIII

8. Additional Outcomes

    - structural induction
    - substitution
    - ...

We covered A LOT

. . . but that's only the tip of the iceberg

The Message I

Good language design?

- precise definition of what the language is
  (so can communicate among the designers)
- technical properties
  (determinacy, decidability of type checking, etc.)
- pragmatic properties
  (usability in-the-large, implementability)

(that's also an answer to LO1)

The Message II

What can you use semantics for?

- to understand a particular language
  - what you can depend on as a programmer
  - what you must provide as a compiler writer
- as a tool for language design:
  - for clean design
  - for expressing design choices, understanding language features and how they interact
  - for proving properties of a language, eg type safety, decidability of type inference.
- as a foundation for proving properties of particular programs verified software

Trend: Verified Software

- increasingly important
- "rough consensus and running code" (trial and error) is not sufficient
- develop operational models of *real-world* languages/applications

- progress in verification makes it possible

  build end-to-end verified systems

  - formal semantics for (a large subset of C) [see M. Norrish]
  - CompCert/CakeML: verified compilers (full compiler verified in Coq/HOL4)
  - seL4: high-assurance, high-performance operating system microkernel (proofs in Isabelle/HOL)
  - formal semantics for hardware (PPC, x86, ARM)

Are We Done

- more ‘standard’ features
  - dependent types
  - continuations
  - lazy evaluation
  - side effects

- more support for separation of concerns
  - low-level features, such as memory models
  - high-level features, such as broadcast

- more applications
  - optimisations
  - code generation

More Features – Dependent Types

- having “compile-time” types that depend on “run-time” values
- can avoid out-of-bounds errors

More Features – Dependent Types

example: typing Lists with Lengths

non-dependant type for list (similar to trees)

$$
\begin{array}{ll}
\text{nil} & : \text{IList} \\
\text{cons} & : \text{int} \rightarrow \text{IList} \rightarrow \text{IList} \\
\text{hd} & : \text{IList} \rightarrow \text{int} \\
\text{tl} & : \text{IList} \rightarrow \text{IList} \\
\text{isnil} & : \text{IList} \rightarrow \text{bool}
\end{array}
$$

More Features – Dependent Types

Example: Typing Lists with Lengths

dependant type for list (carry around length)

nil : IList 0
cons : Πn:nat. int $\rightarrow$ (IList n) $\rightarrow$ (IList (succ n))
hd : Πn:nat. (IList (succ n)) $\rightarrow$ int
tl : Πn:nat. (IList (succ n)) $\rightarrow$ (IList n)
~~isnil~~ :

More Features – Dependent Types

Example: typing lists with lengths

- using and checking dependent types

$$
\begin{array}{l}
(\textbf{fn}\ n : \mathsf{nat} \Rightarrow (\textbf{fn}\ l : \mathsf{IList}(\mathsf{succ}\ (\mathsf{succ}\ n)) \Rightarrow \\
\qquad (\mathsf{hd}\ (\mathsf{succ}\ n)\ l) + \\
\qquad (\mathsf{hd}\ n\ (\mathsf{tl}\ (\mathsf{succ}\ \ n)\ l)) \\
))
\end{array}
$$

- propositions as dependent types
  (Curry–Howard lens)

$$\mathsf{get} : \Pi m : \mathsf{nat}.\ \Pi n : \mathsf{nat}.\ (\mathsf{Less}\ m\ n) \to (\mathsf{IList}\ n) \to \mathsf{int}$$

More Feature – Hardware Model

Fundamental Question

What is the behaviour of memory?

- . . . at the programmer abstraction
- . . . when observed by concurrent code

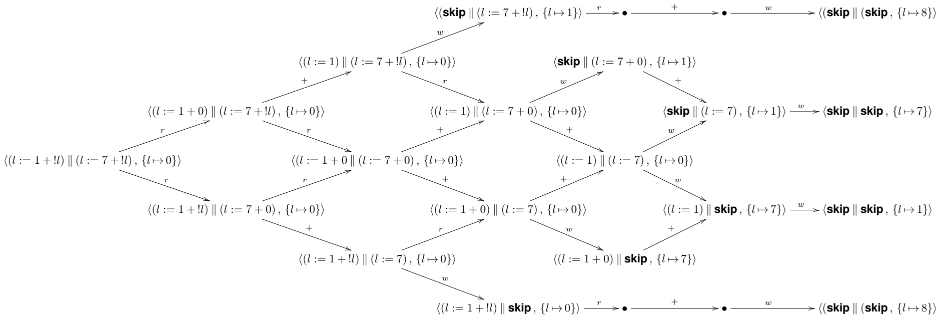
More Feature – Hardware Model

First Model: Sequential Consistency

Multiple threads acting on a sequentially consistent (SC) shared memory:

> *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, respecting the order specified by the program*
>
> *[Lamport, 1979]*

More Feature – Hardware Model



More Feature – Hardware Model

- implement naive mutual exclusion
- specify concepts such as “atomic”
  (see GCL)
- but on x86 hardware you have these behaviours
  - hardware busted?
  - program bad?
  - model is wrong?

SC is not a good model of x86 (or of Power, ARM, Sparc, Itanium. . .)

More Feature – Hardware Model

New problem?

No: IBM System 370/158MP in 1972, already non-SC

Australian National University

More Feature – Hardware Model

But still a research question

- mainstream architectures and languages are key interfaces
- . . . but it is been very unclear exactly how they behave
- **more fundamentally:**
  - it has been (and in significant ways still is) unclear how we can specify that precisely
  - *if* we can do that, we can build on top:
    explanation, testing, emulation, static/dynamic analysis, model-checking, proof-based verification,. . .

More Features – Broadcast

Motivation:
model communication

- network protocols
- communication protocols
- ...

Broadcast in CCS

$$\alpha.P \xrightarrow{\alpha} P \qquad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$

$$\frac{P \xrightarrow{\eta} P'}{P|Q \xrightarrow{\eta} P'|Q} \qquad \frac{P \xrightarrow{c} P',\ Q \xrightarrow{\bar{c}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad \frac{Q \xrightarrow{\eta} Q'}{P|Q \xrightarrow{\eta} P|Q'}$$

$$\frac{P \xrightarrow{\ell} P'}{P[f] \xrightarrow{f(\ell)} P'[f]} \qquad \frac{P \xrightarrow{\ell} P'}{P\backslash c \xrightarrow{\ell} P'\backslash c}\ (c{\neq}\ell{\neq}\bar{c}) \qquad \frac{P \xrightarrow{\ell} P'}{A \xrightarrow{\ell} P'}\ (A \stackrel{def}{=} P)$$

$$\frac{P \xrightarrow{b\sharp_1} P',\ Q \stackrel{b?}{\not\longrightarrow}}{P|Q \xrightarrow{b\sharp_1} P'|Q} \qquad \frac{P \xrightarrow{b\sharp_1} P',\ Q \xrightarrow{b\sharp_2} Q'}{P|Q \xrightarrow{b\sharp} P'|Q'} \qquad \frac{P \stackrel{b?}{\not\longrightarrow},\ Q \xrightarrow{b\sharp_2} Q'}{P|Q \xrightarrow{b\sharp_2} P|Q'}$$

$\sharp_1 \circ \sharp_2 = \sharp \neq \_$ with

$\circ$	!	?
!	_	!
?	!	?

Broadcast in CCS

- parallel composition associative, commutative?
- all operators are a congruence?

Case Study: AODV

Ad Hoc On-Demand Distance Vector Protocol

- routing protocol for wireless mesh networks
  (wireless networks without wired backbone)

- ad hoc (network is not static)
- on-Demand (routes are established when needed)
- distance (metric is hop count)

- developed 1997–2001 by Perkins, Beldig-Royer and Das
  (University of Cincinnati)

- one of the four protocols standardised by the IETF MANET working group (IEEE 802.11s)

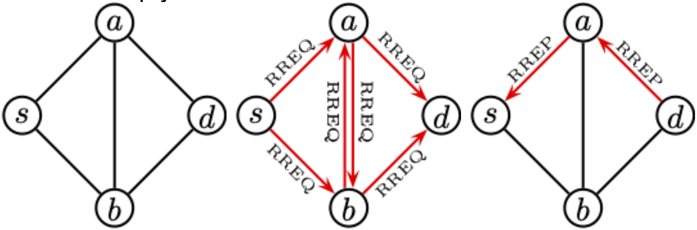
Australian National University

Case Study: AODV

Main Mechanism

- if route is needed
  BROADCAST RREQ
- if node has information about a destination
  UNICAST RREP
- if unicast fails or link break is detected
  GROUPCAST RERR

- performance improvement via intermediate route reply



Case Study: AODV

Formal Specification Language (Process Algebra)

$X(exp_1, \ldots, \mathrm{exp}_n)$	process calls
$P + Q$	nondeterministic
$[\varphi]P$	if-construct (guard)
$[\![\mathtt{var} := exp]\!]P$	assignment followed
$\mathbf{broadcast}(ms).P$	broadcast
$\mathbf{groupcast}(dests, ms).P$	groupcast
$\mathbf{unicast}(dest, ms).P \blacktriangleright Q$	unicast
$\mathbf{send}(ms).P$	send
$\mathbf{receive}(\mathtt{msg}).P$	receive
$\mathbf{deliver}(data).P$	deliver

Case Study: AODV

Specification

```
+ [ (oip, rreqid) ∉ rreqs ]     /* the RREQ is new to this node */
  [[rt := update(rt,(oip,osn,kno,val,hops + 1,sip,∅))]]     /* update the route to oip in rt */
  [[rreqs := rreqs ∪ {(oip,rreqid)}]]     /* update rreqs by adding (oip, rreqid) */
  (
    [ dip = ip ]     /* this node is the destination node */
      [[sn := max(sn,dsn)]]     /* update the sqn of ip */
      /* unicast a RREP towards oip of the RREQ */
      unicast(nhop(rt,oip),rrep(0,dip,sn,oip,ip)) . AODV(ip,sn,rt,rreqs,store)
      ▶ /* If the transmission is unsuccessful, a RERR message is generated */
        [[dests := {(rip,inc(sqn(rt,rip)))|rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,oip)}]]
        [[rt := invalidate(rt,dests)]]
        [[store := setRRF(store,dests)]]
        [[pre := ⋃{precs(rt,rip)|(rip,∗) ∈ dests}]]
        [[dests := {(rip,rsn)|(rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}]]
        groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
    + [ dip ≠ ip ]     /* this node is not the destination node */
      (
        [dip ∈ vD(rt) ∧ dsn ≤ sqn(rt,dip) ∧ sqnf(rt,dip) = kno ]     /* valid route to dip that is fresh enough */
          /* update rt by adding precursors */
          [[rt := addpreRT(rt,dip,{sip})]]
          [[rt := addpreRT(rt,oip,{nhop(rt,dip)})]]
          /* unicast a RREP towards the oip of the RREQ */
          unicast(nhop(rt,oip),rrep(dhops(rt,dip),dip,sqn(rt,dip),oip,ip)) .
```

Case Study: AODV

Full specification of AODV (IETF Standard)

Specification details

- around 5 types and 30 functions
- around 120 lines of specification
  (in contrast to 40 pages English prose)

Properties of AODV

route correctness	✓	
loop freedom	✓	(for some interpretations)
route discovery	✗	
packet delivery	✗	

Final Oral Exam

- **6–10 November, 2021**
- 30 minutes oral examination
- read the guidelines (available via course webpage)
- send through the signed statement in time

GOOD LUCK

Feedback

Please provide feedback

- types of possible feedback
  - suggestions
  - improvements
- send feedback
  - SELT
  - to me (orally, written)

The ‘Final’ Slide

- Q/A sessions
  - Thursday, November 2 (11am-12pm),
    Marie Reay room 5.02
  - topics: all questions you prepare
  - no questions, no session
- I hope you. . .
  - had some fun (I had),
    even despite the challenging times
  - learnt something useful

COMP3610/6361 done – what's next?

- COMP3630/6363 (S1 2024)
  Theory of Computation

- COMP4011/8011 (S2 2022)
  Special Topic: Software Verification using Proof Assistants

- Individual Projects/Honour's Theses/PhD projects ...
  (potentially casual jobs)

Logic Summer School
December 04 – December 15, 2021

Lectures include

- Fundamentals of Metalogic
  (John Slaney, ANU)
- Defining and Reasoning About Programming Languages
  (Fabian Muehlboeck, ANU)
- Propositions and Types, Proofs and Programs
  (Ranald Clouston, ANU)
- Gödel's Theorem Without Tears
  (Dominik Kirst, Ben-Gurion University)
- Foundations for Type-Driven Probabilistic Modelling
  (Ohad Kammar, U Edinburgh)
- . . .

Registration is A$150

`http://comp.anu.edu.au/lss`

— THE END —

Section 27

Add-On

Program Algebras: Floyd-Hoare Logic meets Regular Expressions

Motivation

- CCS and other process algebra yield algebraic expressions, e.g.

$$a.b.\textbf{nil} + c.\textbf{nil}$$

- they also give rise to algebraic (semantic) equalities, e.g.

$$a.\textbf{nil} + a.\textbf{nil} = a.\textbf{nil}$$

- but how does algebra relate to Hoare triples

Beyond Floyd-Hoare Logic

some ‘optimisations’ are not possible within Floyd-Hoare logic

$$\frac{\{P\}\ \textbf{if}\ b\ \textbf{then}\ c\ \textbf{else}\ c\ \{Q\}}{\{P\}\ c\ \{Q\}}$$

(trivially) unprovable in Floyd-Hoare logic

Australian National University

Trace Model – Intuition

a program can be interpreted as set of program runs/traces

sets of traces $s_0 c_1 s_1 c_2 \ldots s_{n-1} c_{n_1} s_n$

$$A \subseteq \Sigma \times (Act \times \Sigma)^*$$

non-deterministic choice	$A \cup B$
sequential composition	$AB = \{asb \mid xs \in A \wedge sb \in B\}$
iteration	$A^* = \bigcup_{n \geq 0} = A^0 \cup A^1 \cup A^2 \ldots$
skip	$1 = \Sigma$ (all traces of length 0)
fail/abort	$0 = \emptyset$

Guarded Commands – Intuition

a program can be interpreted as set of guarded commands

sets of guarded strings $\alpha_0 c_1 \alpha_1 c_2 \ldots \alpha_{n-1} c_{n_1} \alpha_n$
($\alpha, \beta, \ldots$ Boolean expressions)

non-deterministic choice	$A \cup B$
sequential composition	$AB = \{a\alpha b \mid x\alpha \in A \wedge \alpha b \in B\}$
iteration	$A^* = \bigcup_{n \geq 0} = A^0 \cup A^1 \cup A^2 \ldots$
skip	$1 = \{$all Boolean expressions$\}$
fail/abort	$0 = \emptyset$

Properties

- associativity: $a(bc) = (ab)c$
- neutrality: $1a = a = a1$
- distributivity: $(a + b)c = ac + bc$
  $a(b + c) = ab + ac$ (?)
- absorption: $0a = 0 = a0$
- iteration: $(ab)^*a = a(ba)^*$

Regular expressions

we know these rules from regular expressions, finite automata and formal languages

Kleene Algebra (KA)

is the algebra of *regular expressions*
(traces/guarded commands without ‘states’)

Examples

- $ab + ba$
  $\{ab, ba\}$

- $(ab)^*a = a(ba)^*$
  $\{a, aba, ababa, \ldots\}$

- $(a + b)^* = (a^*b)^*a^*$
  {all strings over a,b}

Regular Sets – Intuition

regular sets over Σ

non-deterministic choice ($+$, $\mid$)	$A \cup B$
sequential composition	$AB = \{ab \mid x \in A \wedge b \in B\}$
iteration	$A^* = \bigcup_{n \geq 0} = A^0 \cup A^1 \cup A^2 \ldots$
neutral	$1 = \{\varepsilon\}$
	(language containing the empty word)
empty language	$0 = \emptyset$

Axioms of Kleene Algebra

A *Kleene algebra* is a structure $(K, +, \cdot, 0, 1, ^*)$ such that

- K is an *idempotent semiring* under $+$, $\cdot$, 0, 1

$$
\begin{array}{ll}
(a+b)+c = a+(b+c) & (a, \cdot b)\cdot c = a\cdot(b\cdot c) \\
a+b = b+a & a\cdot 1 = 1\cdot a = a \\
a+a = a & a\cdot 0 = 0\cdot a = 0 \\
a+0 = a &
\end{array}
$$

$$
\begin{array}{l}
a\cdot(b+c) = a\cdot b + a\cdot c \\
(a+b)\cdot c = a\cdot c + b\cdot c
\end{array}
$$

- $a^*b =$ least x such that $b + ax \leq x$
- $ba^* =$ least x such that $b + xa \leq x$

$x \leq y \Leftrightarrow x + y = y$

multiplication symbol is omitted

Characterising Iteration

- complete semiring/quantales (suprema exist)

$$a^* = \Sigma_{n\geq 0}\, a^n$$

  supremum with respect to $\leq$

- Horn axiomatisation
  - $a^*b =$ least x such that $b + ax \leq x$:

$$1 + aa^* \leq a*$$
$$b + ax \leq x \Rightarrow a^*b \leq x$$

  - $ba^* =$ least x such that $b + xa \leq x$:

$$1 + a^*a \leq a*$$
$$b + ax \leq x \Rightarrow ba^* \leq x$$

Models & Properties

regular expressions, traces and guarded strings form Kleene algebras

abstract laws: $(ab)^*a \leq a(ba)^*$
(proof is a simple exercise)

applies to all models

guarded strings/commands have more structure (assertions)

Kleene Algebra with Tests (KAT)

A *Kleene algebra with tests* is a structure $(K, B, +, \cdot, ^*, \neg, 0, 1)$, such that

- $(K, +, \cdot, ^*, 0, 1)$ is a Kleene algebra
- $(B, +, \cdot, \neg, 0, 1)$ is a Boolean algebra
- $B \subseteq K$

- $a, b, c, \ldots$ range over K
- $p, q, r, \ldots$ range over B

Kleene Algebra with Tests (KAT)

$+$, $\cdot$, 0, 1 serve double duty

- applied to programs, denote choice, composition, fail, and skip, resp.
- applied to tests, denote disjunction, conjunction, falsity, and truth, resp.
- these usages do not conflict

$$pq = p \wedge q \qquad p + q = p \vee q$$

Models

- Trace models
  K: sets of traces $s_0 c_1 s_1 c_2 \ldots s_{n-1} c_{n_1} s_n$
  B: sets of traces of length 0
- Language-theoretic models K: sets of guarded strings
  $\alpha_0 c_1 \alpha_1 c_2 \ldots \alpha_{n-1} c_{n_1} \alpha_n$
  B: atoms of a finite free Boolean algebra

Modelling Programs

[Fischer & Ladner 79]

- $a\,;\,b = ab$
- **if** p **then** a **else** $c = pa + \neg pc$
- **while** p **do** $c = (pc)^*\neg p$

Floyd-Hoare Logic vs KAT

Theorem

KAT subsumes propositional Floyd-Hoare logic (PHL)
(Floyd-Hoare logic without assignment rule)

$\{p\}\ c\ \{q\}$ modeled by $pc = pcq$ (or $pc\neg q = 0$, or $pc\neg q \leq 0$)

Floyd-Hoare logic

$$\frac{\{p\}\ a\ \{q\} \quad \{q\}\ b\ \{r\}}{\{p\}\ ab\ \{r\}}$$

$$pa\neg q = 0 \wedge qb\neg r = 0 \Longrightarrow pab\neg r = 0$$

$$\frac{\{p \wedge r\}\ a\ \{q\} \quad \{p \wedge \neg r\}\ b\ \{q\}}{\{p\}\ \textbf{if}\ r\ \textbf{then}\ a\ \textbf{else}\ b\ \{q\}}$$

$$pra\neg q = 0 \wedge p\neg rb\neg q = 0 \Longrightarrow p(ra + \neg rb)\neg q = 0$$

$$\frac{\{p \wedge r\}\ a\ \{p\}}{\{p\}\ \textbf{while}\ r\ \textbf{do}\ a\ \{\neg r \wedge p\}}$$

$$pra\neg p = 0 \Longrightarrow p(ap)^*\neg(\neg rp) = 0$$

Crucial Theorems

Theorem

These are all theorems of KAT

(proof is an exercise)

Theorem (Completeness Theorem)

All valid rules of the form

$$\frac{\{p_1\}\ c_1\ \{q_1\} \quad \ldots \quad \{p_n\}\ c_n\ \{q_n\}}{\{p\}\ c\ \{q\}}$$

are derivable in KAT (not so in PDL)

Advantages of Kleene Algebra

- unifying approach
- equational reasoning + Horn clauses
  some decidability & automation
- but, missing out assignment rule of Floyd-Hoare logic

Other Applications of KA(T)

There are more applications

- automata and formal languages
  - regular expressions
- relational algebra
- program logic and verification
  - dynamic Logic
  - program analysis
  - optimisation
- design and analysis of algorithms
  - shortest paths
  - connectivity
- others
  - hybrid systems
  - ...

Rely-Guarantee Reasoning

Hoare triple

$$\{p\}\ c\ \{q\} \;\Leftrightarrow\; pc\neg q = 0$$

But what about $\{P, R\}\ c\ \{G, Q\}$?

$$\begin{aligned}\{p, a_R\}\ c\ \{b_G, q\} &\Leftrightarrow \{p\}\ a_R \parallel c\ \{q\} \wedge c \leq b_G \\ &\Leftrightarrow p(a_R \parallel c)\neg q = 0 \wedge c \leq b_G\end{aligned}$$

needs algebra featuring parallel (we have seen one)

- $R \parallel (S + T) = R \parallel S + R \parallel T$
- $R \parallel (S \cdot T) = (R \parallel S) \cdot (R \parallel T)$
- $R \parallel (S \parallel T) = (R \parallel S) \parallel (R \parallel T)$

Rely-Guarantee Reasoning

Hoare triple

$$\{p\}\ c\ \{q\} \ \Leftrightarrow\ pc\neg q = 0$$

But what about $\{P, R\}\ c\ \{G, Q\}$?

$$\begin{aligned}\{p, a_R\}\ c\ \{b_G, q\} \ &\Leftrightarrow \{p\}\ a_R \parallel c\ \{q\} \wedge c \leq b_G \\ &\Leftrightarrow p(a_R \parallel c)\neg q = 0 \wedge c \leq b_G\end{aligned}$$

needs algebra featuring parallel (we have seen one)

- $R \parallel (S + T) = R \parallel S + R \parallel T$
- $R \parallel (S \cdot T) = (R \parallel S) \cdot (R \parallel T)$
- $R \parallel (S \parallel T) = (R \parallel S) \parallel (R \parallel T)$