Experience of Teaching Theory of Computation[⋆]

Ornela Dardha[0000−0001−9927−7875]

School of Computing Science, University of Glasgow, Glasgow, UK
`Ornela.Dardha@glasgow.ac.uk`

Abstract. Teaching formal methods is as important as it is challenging. On one hand, formal methods capture the foundations of computing science, needed by any computer scientist; on the other hand teaching formal methods is challenging, because students often lack the necessary background to assimilate the course material or worse, they lack interest for theory, or they do not see the need for it or its relevance.
In this paper, I will report on my experience of teaching the λ-calculus and the π-calculus at the Theory of Computation (ToC) course at the School of Computing Science, University of Glasgow. I will discuss what went well and what went less well. Finally, I will discuss and reflect on how much of the challenge in teaching formal methods is due to the topic taught *vs.* the approaches to teaching adopted in class.

1 Introduction

Teaching formal methods is as important as it is challenging. On one hand, formal methods capture the very foundations and mathematical concepts of computing science, needed by any computer scientist; on the other hand teaching formal methods is challenging, because often students lack the necessary background to understand and assimilate the course material or worse, they lack interest for theory, or they do not see the need for it or its relevance. Today, in the era of big data, machine learning and robotics, this is more noticeable than ever!

As a new lecturer in formal methods, I was faced with this challenge when for the first time in this academic year 2018-19 I taught the Theory of Computation (ToC) course in Computing Science, at University of Glasgow. As stated by Weller [29] "*becoming a teacher happens over time*" and by Kugel [19] "*like the learning abilities of their students, the teaching abilities of college professors seem to develop in stages.*" Kugel identifies five stages of teachers' focus: *stage 1*: self; *stage 2*: subject; *stage 3*: students as receptive; *stage 4*: students as active; *stage 5*: students as independent. As a teacher moves up the stages, the emphasis shifts from *teaching* to *learning*, which begins at *stage 3* [19].

As a lecturer in computing science, who passionately researches and teaches formal methods, this challenge brought me to the following questions:

⋆ Supported by the UK EPSRC grant EP/K034413/1, "*From Data Types to Session Types: A Basis for Concurrency and Distribution* (ABCD)", and the EU HORIZON 2020 MSCA RISE project 778233 "*BehAPI: Behavioural Application Program Interfaces*".

Q1: *As university teachers, how can we teach formal methods in an effective and interesting way to students?*

Q2: *How much of the challenge in teaching formal methods is due to the topic itself vs. the approaches to teaching and learning adopted?*

In this paper, I will try and address these questions; I will describe what went well and what went less well in the Theory of Computation course, and I will try to *critically reflect* [13] on my practice in learning and teaching in relation to the *four lenses* [5]: my experience, students feedback, colleagues, and learning and teaching literature. Finally, I will discuss future improvements to the course.

2 The Greek-letter Calculi

2.1 The λ-calculus

The λ-calculus is a formal model of computation, based on *functions*. As such, it has a formal syntax for defining functions; a formal semantics for evaluating functions; and a formal theory for equating functions. In this paper, we will discuss all three ingredients of the model.

The λ-calculus in this paper is based on the Theory of Computation (ToC) course [1], which in turn is based on Hankin [12].

Syntax We start this section by giving the formal syntax of the λ-calculus.

Definition 1 (λ-terms (Def. 2.1.2 in Hankin)). *The set Λ of λ-terms is the smallest set satisfying the following:*

1. *if x is a variable then $x \in \Lambda$ (assuming an infinite set of variables)*
2. *if $M \in \Lambda$ then $(\lambda x.M) \in \Lambda$ (called abstractions; right-associative)*
3. *if $M, N \in \Lambda$ then $(MN) \in \Lambda$ (called applications; left-associative)*

An alternative, but equivalent definition of the λ-calculus syntax is given by the following grammar:

$$M, N ::= x \quad | \quad (\lambda x.M) \quad | \quad (MN)$$

Definition 2 (Bound Variables). *The set of bound variables of a λ-term M is $BV(M)$, where the function $BV : \Lambda \to \mathcal{P}(Var)$ is defined by:*

$$\begin{aligned} BV\ x &= \emptyset \\ BV(\lambda x.M) &= (BV\ M) \cup \{x\} \\ BV(MN) &= (BV\ M) \cup (BV\ N) \end{aligned}$$

λ is a binder in $\lambda x.M$; it creates "local" variable x and binds it with scope M.

Definition 3 (Free Variables). *The set of free variables of a λ-term M is $FV(M)$, where the function $FV : \Lambda \to \mathcal{P}(Var)$ is defined by:*

$$\begin{aligned} FV\ x &= \{x\} \\ FV(\lambda x.M) &= (FV\ M) - \{x\} \\ FV(MN) &= (FV\ M) \cup (FV\ N) \end{aligned}$$

[1] https://www.gla.ac.uk/coursecatalogue/course/?code=COMPSCI4072

Semantics The main *reduction rule* in the λ-calculus is the *β-rule.*

$$(\lambda x.M)N \to_\beta M[x := N] \qquad\qquad (\beta)$$

where $M[x := N]$ denotes *substitution*: *every* occurrence of variable x in term M is replaced by term N.

The β-rule captures function application evaluation: a function $\lambda x.M$ when applied to a term N reduces $\to_\beta$ to the body of the function M, where the "local" variable x is substituted by the term N.

Example 1. To illustrate the β-rule, let us consider the *addition* function, applied to two arguments (we are assuming reduction rules for arithmetical operations):

$$((\lambda x.\lambda y.(x+y))2)3 \to_\beta (\lambda y.(2+y))3 \to_\beta 2+3 \to_\beta 5$$

Variable substitution is tricky and one can define it in different ways (see [12] for details). Why is it tricky? Consider the most intuitive substitution after the β-rule is applied to $(\lambda x.\lambda y.yx)\mathbf{y}$ (example taken from ToC):

$$(\lambda x.\lambda y.yx)\mathbf{y} \to_\beta \lambda y.y\mathbf{y}$$

The problem here is that $\mathbf{y}$ is *captured* by λy after reduction. In order to avoid the unwanted capture of the free variable $\mathbf{y}$, we can simply rename the bound variable y into a fresh z—called α-renaming [12], and then substitution becomes:

$$(\lambda x.\lambda z.zx)\mathbf{y} \to_\beta \lambda z.z\mathbf{y}$$

In this paper we will assume that *all bound variables have already been α-renamed and they are different from each other and from any free variables*—called the Barendregt variable convention, which avoids unwanted variables capture.

Definition 4 (Substitution (Def. 2.3.3 in Hankin)). *Assume the Barendregt variable convention, achieved by α-renaming. We define substitution as:*

1. $x[x := N] \equiv N$
2. $y[x := N] \equiv y$ *if y is not the same as x*
3. $(\lambda y.M)[x := N] \equiv \lambda y.(M[x := N])$
4. $(M_1M_2)[x := N] \equiv (M_1[x := N])(M_2[x := N])$

We are now ready to define the rest of the *one-step β-reduction* rules, which allow reductions in bigger λ terms.

$$(\lambda x.M)N \to_\beta M[x := N] \qquad \frac{M \to_\beta N}{MZ \to_\beta NZ}$$

$$\frac{M \to_\beta N}{ZM \to_\beta ZN} \qquad \frac{M \to_\beta N}{\lambda x.M \to_\beta \lambda x.N}$$

Example 2 (one-step β-reduction (taken from ToC)).

$$\begin{aligned}(\lambda xyz.xzy)(\lambda xz.x) &\equiv_\alpha (\lambda xyz.xzy)(\lambda uv.u) \text{ (Barendregt variable convention)}\\ &\to_\beta \lambda yz.(\lambda uv.u)zy\\ &\to_\beta \lambda yz.z\end{aligned}$$

Finally, a *β-reduction* $\twoheadrightarrow_\beta$, is a sequence of zero or more one-step β-reductions, meaning it is the *reflexive and transitive closure of one-step β-reduction.*

$$\frac{M \to_\beta N}{M \twoheadrightarrow_\beta N} \qquad M \twoheadrightarrow_\beta M \qquad \frac{M \twoheadrightarrow_\beta N \quad N \twoheadrightarrow_\beta L}{M \twoheadrightarrow_\beta L}$$

We will now introduce *compatibility*, which states that the definitions of $\to_\beta$ and $\twoheadrightarrow_\beta$ are well-founded and well-formed. Before that we need the notion of *context.*

Definition 5 (Context). *A context $C[]$ is defined by the following grammar:*

$$C[] ::= x \quad | \quad [] \quad | \quad (\lambda x.C[]) \quad | \quad (C[]C[])$$

Proposition 1 (Compatibility (Prop. 3.2.1 in Hankin)). *Let $C[]$ be a context with one hole.*

1. *If $M \to_\beta N$ then $C[M] \to_\beta C[N]$.*
2. *If $M \twoheadrightarrow_\beta N$ then $C[M] \twoheadrightarrow_\beta C[N]$.*

Theory of equality We will now talk about theory λ, which captures *equal* $=$ λ-terms. Applying $\to_\beta$ (and $\twoheadrightarrow_\beta$) produces equal terms, for e.g., $(\lambda x.\ x+1)1 = 2$. But we also want to equate for e.g., $(\lambda x.\ x+1)2 = (\lambda x.\ x+2)1$. The theory λ, $\lambda \vdash M = N$, should thus satisfy the following requirements:

– an application term is *equal to the result* of applying the function part of the term to the argument
– equality is an *equivalence relation*, i.e., reflexive, symmetric and transitive
– equal terms are *equal in any context.*

These requirements are captured by the following axioms and inference rules.

$$(\lambda x.M)N = M[x := N]$$

$$M = M \qquad \frac{M = N}{N = M} \qquad \frac{M = N \quad N = L}{M = L}$$

$$\frac{M = N}{MZ = NZ} \qquad \frac{M = N}{ZM = ZN} \qquad \frac{M = N}{\lambda x.M = \lambda x.N}$$

The first line is the β-axiom, namely the β-rule without direction. The second line states that equality is an equivalence relation. The last line states respectively, applying equal functions to the same argument Z produces equal results;

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\,\mathsf{TVar} \qquad \frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T.M : T \to U}\,\mathsf{TAbs} \qquad \frac{\Gamma \vdash M : T \to U \quad \Gamma \vdash N : T}{\Gamma \vdash MN : U}\,\mathsf{TApp}$$

Fig. 1. Typing rules for the simply-typed λ-calculus

applying a function to equal arguments $M = N$, produces equal results; abstractions with equal bodies, are equal.

Finally, we show the *referential transparency* in Thm. 1, stating that if two terms are equal, then they are equal in every context. This result implies that only the value of an expression is important, not the computation.

Theorem 1. *For every context $C[]$ and all terms $M, N \in \Lambda$, if $\lambda \vdash M = N$ then $\lambda \vdash C[M] = C[N]$.*

Types and Type Systems We will now present the *simply-typed λ-calculus*, being the most basic typed version of λ-calculus.

Definition 6 (Types, Context and Judgement). *The syntax of* types *is defined by the grammar:*

$$T ::= A \quad | \quad T \to T$$

where A is an atomic *type and by convention, $\to$ is right-associative.*
A typing context *Γ is a set of variables associated to types, defined by:*

$$\Gamma ::= \emptyset \quad | \quad \Gamma, x : T$$

Let Γ be a typing context and M a λ-term. A typing judgement *is*

$$\Gamma \vdash M : T$$

reading "term M has type T under the typing context Γ".

To accommodate types, we adopt *Church*-style abstractions: $\lambda x : T.M$ i.e., x has type T. The type system for the simply-typed λ-calculus is defined by the typing rules in Fig. 1. Note that for a more general-purpose λ-calculus, we could add *ground* types, like `Bool` and `Int`, together with their values and typing rules.

Example 3 (Typing derivation (taken from ToC)). We will now construct the typing derivation for the term $\lambda x : A.\lambda y : A.\lambda z : A \to A \to A.zxy$, and let $\Gamma = x : A, y : A, z : A \to A \to A$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash z : A \to A \to A \quad \Gamma \vdash x : A}{\Gamma \vdash zx : A \to A}\,\mathsf{TApp} \qquad \Gamma \vdash y : A}{\Gamma \vdash zxy : A}\,\mathsf{TApp}}{x : A, y : A \vdash \lambda z : A \to A \to A.zxy : (A \to A \to A) \to A}\,\mathsf{TAbs}}{x : A \vdash \lambda y : A.\lambda z : A \to A \to A.zxy : A \to (A \to A \to A) \to A}\,\mathsf{TAbs}}{\vdash \lambda x : A.\lambda y : A.\lambda z : A \to A \to A.zxy : A \to A \to (A \to A \to A) \to A}\,\mathsf{TAbs}$$

As it happens in mainstream programming languages, we introduce a type system to statically (compile-time) guarantee absences of runtime type errors: in the case of λ-calculus we want to eliminate mismatches between an abstraction type and its argument, when the abstraction is applied. Type safety is a corollary of two main results: Type Preservation (Subject Reduction) and Progress [24].

Theorem 2 (Type Preservation/Subject Reduction). *If $\Gamma \vdash M : T$ and $M \to_\beta N$, then $\Gamma \vdash N : T$.*

We call *value* an expression which cannot be reduced further. In λ-calculus the only values are abstractions (and the possibly added *ground* values).

Theorem 3 (Progress). *If $\Gamma \vdash M : T$, then either M is a value or there exists M' such that $M \to_\beta M'$.*

2.2 The π-calculus

The π-calculus is a formal model of computation, based on *processes.* The core features of π-calculus are *communication* and *concurrency* and the key difference with respect to its predecessor CCS [21] is *mobility*: a communication *channel* is used to send and receive *messages*, and itself it can be sent and received over other channels by other processes.

The π-calculus in this paper is based on the Theory of Computation (ToC) course, which in turn is based on Milner [22], Sangiorgi and Walker [27].

Syntax We let x, y range over *channel names* or *variables*, and P, Q range over *processes.* The syntax of π-calculus processes is defined by the grammar:

$$\begin{array}{rll}
P, Q ::= & \mathbf{0} & \text{(inaction)} \\
\mid & x(y).P & \text{(input)} \\
\mid & \overline{x}\langle y \rangle.P & \text{(output)} \\
\mid & \tau.P & \text{(silent action)} \\
\mid & P + Q & \text{(choice)} \\
\mid & P \mid Q & \text{(parallel composition)} \\
\mid & (\nu x)P & \text{(restriction)} \\
\mid & !P & \text{(replication)}
\end{array}$$

$\mathbf{0}$ is the inaction process. The input $x(y).P$ (resp. output $\overline{x}\langle y \rangle.P$) process receives (resp. sends) on channel x a message y and proceeds as P. Process $\tau.P$ is a silent action. $P + Q$ is a choice and $P \mid Q$ is a parallel composition between P and Q. Process $(\nu x)P$ is the channel restriction process; it creates a channel with name x and binds it with scope P. Process $!P$ is a replication of process P.

As for the λ-calculus, we adopt the Barendregt variable convention. We use $BV(P)$ to denote the set of bound variables and $FV(P)$ the set of free variables of P. Note that the only processes creating bound variables are the input $x(y).P$ and restriction $(\nu x)P$, where y (resp. x) is bound with scope P.

Semantics The main reduction rule for π-calculus is the *communication rule.*

$$a(x).P \mid \overline{a}\langle y\rangle.Q \to P[x := y] \mid Q \qquad\qquad (com)$$

As with λ-calculus, we want to be able to apply *com* in bigger terms, i.e.,

$$a(x).P \mid \mathbf{R} \mid \overline{a}\langle y\rangle.Q \to P[x := y] \mid \mathbf{R} \mid Q$$

In order to virtually bring the parallel components of a processes closer together, we use *structural congruence* $\equiv$ on processes, which allows us to ignore the order of parallel composition, and more thus allows reduction in bigger *contexts.*

Definition 7 (Structural congruence). *Structural congruence is the* smallest congruence relation *on processes that includes α-renaming and the axioms:*

$$\begin{array}{cl}
P \mid Q \equiv Q \mid P & \text{(parallel is commutative)}\\
P \mid (Q \mid R) \equiv (P \mid Q) \mid R & \text{(parallel is associative)}\\
P \mid \mathbf{0} \equiv P & \text{(garbage collection)}\\
P + Q \equiv Q + P & \text{(choice is commutative)}\\
P + (Q + R) \equiv (P + Q) + R & \text{(choice is associative)}\\
P + \mathbf{0} \equiv P & \text{(garbage collection)}\\
(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P & \text{(reordering } \nu)\\
(\nu x)\mathbf{0} \equiv \mathbf{0} & \text{(garbage collection)}\\
!P \equiv P \mid !P & \text{(behaviour of replication)}\\
P \mid (\nu x)Q \equiv (\nu x)(P \mid Q)\ x \notin FV(P) & \text{(scope expansion)}
\end{array}$$

We can now complete the *reduction relation* for the π-calculus ([22, Def. 4.13]):

$$(a(x).P + P') \mid (\overline{a}\langle y\rangle.Q + Q') \to P[x := y] \mid Q$$

$$\tau.P \ + \ Q \to P$$

$$\frac{P \to Q}{(\nu x)P \to (\nu x)Q} \qquad \frac{P \to Q}{P \mid R \to Q \mid R} \qquad \frac{P' \equiv P \quad P \to Q \quad Q \equiv Q'}{P' \to Q'}$$

Example 4 (Reduction (taken from ToC)). Consider the following process:

$$a(x).\overline{a}\langle -x\rangle.\mathbf{0} \mid \overline{a}\langle 2\rangle.a(z).P(z)$$

channel a is sending and receiving in parallel, triggering *reduction* steps:

$$a(x).\overline{a}\langle -x\rangle.\mathbf{0} \mid \overline{a}\langle 2\rangle.a(z).P(z) \ \to\ \overline{a}\langle -2\rangle.\mathbf{0} \mid a(z).P(z) \ \to\ \mathbf{0} \mid P(-2)$$

Mobile phones (taken from ToC and [22, §8.2]) This example illustrates the π-calculus and its key concept: *mobility.* The system is composed by a car, two transmitters, and a controller. At any time, the car communicates, i.e., *talk*, *switch*, with only one of the transmitters. The controller tells a transmitter to *lose* or *gain* connection to the car. Without loss of generality, we will work with polyadic π-calculus, where sends and receives accept multiple parameters, and parametrised recursive process definitions, rather than replication[2].

Trans is parametrised by the channels it shares with *Control*, i.e., *lose*, *gain* and *Car*, i.e., *talk*, *switch*. It can either *talk* or *lose* the connection. *IdTrans*, the idle transmitter, is parametrised by the channels it shares with *Control*, i.e., *lose*, *gain*. It can *gain* a connection.

$$\begin{aligned} \mathit{Trans}(\mathit{talk}, \mathit{switch}, \mathit{gain}, \mathit{lose}) &= \mathit{talk}().\mathit{Trans}(\mathit{talk}, \mathit{switch}, \mathit{gain}, \mathit{lose}) \\ &\quad + \mathit{lose}(t, s).\overline{\mathit{switch}}\langle t, s\rangle.\mathit{IdTrans}(\mathit{gain}, \mathit{lose}) \\ \mathit{IdTrans}(\mathit{gain}, \mathit{lose}) &= \mathit{gain}(t, s).\mathit{Trans}(t, s, \mathit{gain}, \mathit{lose}) \end{aligned}$$

Control will either be $\mathit{Control}_1$ or $\mathit{Control}_2$. *Control* can tell one transmitter to lose a connection and tell the other transmitter to gain a connection.

$$\begin{aligned} \mathit{Control}_1 &= \overline{\mathit{lose}_1}\langle \mathit{talk}_2, \mathit{switch}_2\rangle.\overline{\mathit{gain}_2}\langle \mathit{talk}_2, \mathit{switch}_2\rangle.\mathit{Control}_2 \\ \mathit{Control}_2 &= \overline{\mathit{lose}_2}\langle \mathit{talk}_1, \mathit{switch}_1\rangle.\overline{\mathit{gain}_1}\langle \mathit{talk}_1, \mathit{switch}_1\rangle.\mathit{Control}_1 \end{aligned}$$

The talk_i, switch_i, gain_i, lose_i channels are treated as *global* variables, meaning they are in scope of the whole system (see System_1 below). The *Car* can either *talk*, or *switch* to a new pair of channels.

$$\mathit{Car}(\mathit{talk}, \mathit{switch}) = \overline{\mathit{talk}}\langle\rangle.\mathit{Car}(\mathit{talk}, \mathit{switch}) + \mathit{switch}(t, s).\mathit{Car}(t, s)$$

We now define the whole communicating system, and assume that *Car* is connected to Trans_1.

$$\begin{aligned} \mathit{System}_1 = (\nu \mathit{talk}_1, \mathit{switch}_1, \mathit{gain}_1, \mathit{lose}_1, \mathit{talk}_2, \mathit{switch}_2, \mathit{gain}_2, \mathit{lose}_2) \\ (\mathit{Car}(\mathit{talk}_1, \mathit{switch}_1) \mid \mathit{Trans}_1 \mid \mathit{IdTrans}_2 \mid \mathit{Control}_1) \end{aligned}$$

where, for $i \in \{1, 2\}$,

$$\begin{aligned} \mathit{Trans}_i &= \mathit{Trans}(\mathit{talk}_i, \mathit{switch}_i, \mathit{gain}_i, \mathit{lose}_i) \\ \mathit{IdTrans}_i &= \mathit{IdTrans}(\mathit{gain}_i, \mathit{lose}_i) \end{aligned}$$

Applying the reduction rules for π-calculus, previously introduced, we can easily verify that $\mathit{System}_1 \to^* \mathit{System}_2$, where System_2 is just System_1 with indices 1 and 2 exchanged.

$$\begin{aligned} \mathit{System}_2 = (\nu \mathit{talk}_1, \mathit{switch}_1, \mathit{gain}_1, \mathit{lose}_1, \mathit{talk}_2, \mathit{switch}_2, \mathit{gain}_2, \mathit{lose}_2) \\ (\mathit{Car}(\mathit{talk}_2, \mathit{switch}_2) \mid \mathit{IdTrans}_1 \mid \mathit{Trans}_2 \mid \mathit{Control}_2) \end{aligned}$$

Types and Type Systems We will now present the *simply-typed π-calculus* [27]. As for the simply-typed λ-calculus, we want to eliminate runtime type errors, in particular mismatches in: (i) the *number* of messages, for e.g., $\overline{a}\langle u, v\rangle.\mathbf{0} \mid a(x).P$; (ii) the *type* of messages, for e.g., $\overline{a}\langle \mathsf{true}\rangle.a(y).\mathbf{0} \mid a(x).\overline{a}\langle -x\rangle.\mathbf{0}$.

As for λ-calculus, types trigger modifications in the syntax of processes. We will annotate bound variables with types and consider the *polyadic* π-calculus: $a(x_1 : T_1, \ldots, x_n : T_n)$ and $(\nu x : T)$.

[2] Following [22] it is easy to translate recursive definitions into replication

$$\frac{}{\Gamma, x : T \vdash x : T}\ \mathsf{TVar} \qquad \frac{}{\Gamma \vdash \mathbf{0}}\ \mathsf{TNil}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : \sharp[T_1, \ldots, T_n] \\ \Gamma, y_1 : T_1, \ldots, y_n : T_n \vdash P\end{array}}{\Gamma \vdash x(y_1 : T_1, \ldots, y_n : T_n).P}\ \mathsf{TIn} \qquad \frac{\begin{array}{c}\Gamma \vdash x : \sharp[T_1, \ldots, T_n] \\ \Gamma \vdash y_i : T_i \ \forall i \in [n] \quad \Gamma \vdash P\end{array}}{\Gamma \vdash \overline{x}\langle y_1, \ldots, y_n\rangle.P}\ \mathsf{TOut}$$

$$\frac{\Gamma, x : \sharp[T_1, \ldots, T_n] \vdash P}{\Gamma \vdash (\nu\ x : \sharp[T_1, \ldots, T_n])P}\ \mathsf{TNew} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q}\ \mathsf{TPar}$$

Fig. 2. Typing rules for the simply-typed π-calculus

Definition 8 (Types, Context and Judgement). *The syntax of* types *is defined by the following grammar:*

$$T ::= \sharp[T, \ldots, T] \quad \mid \quad \mathtt{Bool}$$

A typing context Γ *is a set of variables associated to types, defined by:*

$$\Gamma ::= \emptyset \quad \mid \quad \Gamma, x : T$$

Let Γ be a typing context and P a process. A typing judgement for processes *is*

$$\Gamma \vdash P$$

reading "process P is well typed under typing context Γ".
A typing judgement for channel names *is $\Gamma \vdash x : T$ reading "channel name x has type T under typing context Γ".*

The type system for the simply-typed π-calculus is defined by the typing rules in Fig. 2 and it guarantees that all channels are used according to their types. This type safety property is a corollary of the following Thm. 4 and Thm. 5.

Theorem 4 (Type Preservation). *If $\Gamma \vdash P$ and $P \to Q$ then $\Gamma \vdash Q$.*

Before stating the next theorem, note that by using scope expansion Def. 7, any process can be written in the form $(\nu x_1 \ldots x_k)(P_1 \mid \ldots \mid P_m)$ where immediate communications among processes P_i for all $i \in [m]$ are at the top level, and not inside other ν constructors.

Theorem 5 (Absence of Immediate Communication Errors). *If*

$$\Gamma \vdash (\nu x_1 : T_1 \ldots x_k : T_k)(x(y_1 : U_1, \ldots, y_r : U_r).P \mid \overline{x}\langle v_1, \ldots, v_n\rangle.Q \mid R)$$

then (i) $n = r$ and (ii) for all $i \in [n]$, v_i has type U_i.

The combined Thm. 4 and Thm. 5 state that if a process is well typed—there are no immediate communication errors—and it reduces, then the reduced process is also well typed, thus guaranteeing items (i) and (ii) in Thm. 5 for all reduced processes, meaning for all communications of a process. Thus, we have captured the two items discussed at the beginning of this section.

3 Theory of Computation: Course Overview

Theory of Computation (ToC) is an honours course at the School of Computing Science, University of Glasgow, aimed at level four undergraduate students. However, it is also available to master students, visiting and Erasmus students.

The course description on the course catalogue states: "*This course covers the theory of sequential, concurrent and quantum computation. The main topics are: formal language theory and the connection to automata; lambda calculus as a foundation for functional computation; pi calculus as a foundation for concurrent computation; the theory of operational semantics and type systems; principles of quantum computation.*" The aim of the course is to "*show how several models of computation can be formally defined in order to give a rigorous foundation for sequential, concurrent and quantum programming paradigms.*"

ToC is structured in four *sections*. The following are the reference textbooks:

– V. J. Rayward-Smith. *A First Course in Formal Language Theory* [25], covering §1 on Automata and Formal Language Theory.
– C. Hankin. *An Introduction to Lambda Calculi for Computer Scientists* [12], covering §2 on the λ-calculus.
– R. Milner. *Communicating and Mobile Systems: The Pi-Calculus* [22], covering a short introduction to §3 on the π-calculus.
– D. Sangiorgi and D. Walker. *The pi-calculus: A Theory of Mobile Processes* [27], full cover of §3 on the π-calculus.
– J. Gruska. *Quantum Computing* [11], covering §4 on Quantum Computing.

ToC was originally designed by Prof. Simon J. Gay at the School of Computing Science, University of Glasgow and it has been running since the academic year 2017–18. I will comment on my experience of teaching this course in the last two academic years, since I started my lecturing position at University of Glasgow and became the responsible lecturer for this course.

The timetable for ToC is *three hours per week*: two hours face-to-face lectures, and one hour tutorial or practical work in the lab. ToC runs for ten weeks during the (first) semester. Student attendance is low compared to the other honours courses, it ranges in 15–25 students per academic year.

Lectures There are two hours of face-to face lectures per week. The lectures are delivered via a slide presentation prepared in Beamer/LaTeX. The above textbooks are for reference and can be found at the University's library.

During lectures I use the whiteboard to solve exercises in class, as well as invite students to come and solve exercises, in turn. These exercises include for e.g., proofs by (structural) induction, showing reductions in λ-calculus or π-calculus, or drawing type derivations in λ-calculus or π-calculus.

Labs There is one hour lab work per week, where the teacher is present in the lab to help students. The lab work for each section is as follows:

- §1 Implement (pushdown) automata in Python[3].
- §2 Explore programming with λ-calculus in Python: representation of booleans, natural numbers, conditionals and recursive programming with fixed points.
- §3 Explore programming with π-calculus in Go[4]: representation of booleans, natural numbers, conditionals. Finally, students implemented in Go the mobile phones example given in § 2.2.
- §4 Use a simulation environment—developed by a final year student project—to explore key concepts of quantum computing, such as quantum gates, entanglement and teleportation. The web application can be found following the link http://quantumplayground.azurewebsites.net.

The reason for using Python for automata theory and λ-calculus is that students in Computing Science at University of Glasgow study Python during their first year and they are familiar with the language.

The reason for using Go for π-calculus is twofold: first, it is a language based on CCS/π-calculus [21,22,27]; second, it is a new language that students have not used in other courses, so it adds to their knowledge and skills set.

Assessed Coursework and Exam The examination for the Theory of Computation course is split between 80% for the end of year/summer exam, usually in April/May and 20% for the assessed coursework, which happens roughly halfway during the semester [5].

The exam paper covers all the four sections taught in ToC. For the assessed coursework, the assignment covers the main parts of the course which are the λ-calculus and the π-calculus. For e.g., for the academic year 2018–19, the assignment on λ-calculus was to define a λ-calculus representation of binary trees in Python, and the assignment on π-calculus was to define a π-calculus representation of lists of natural numbers in Go.

The coursework is submitted online and results are sent by email, which facilitates the whole marking process [26]. The final exam is done in classroom, and the completed exam paper is returned for marking to the lecturer.

3.1 Other topics covered but not discussed in § 2

On λ-calculus We covered the *fixed point* and fixed point theorem and detailed its proof, showing how to construct a fixed point for every λ-term [12]. Students used fixed points to explore recursive programming with λ-calculus in Python during labs, and fixed points were also assessed in the final exam.

We covered normal forms, strong and weak normalisation, and the Church-Rosser theorem; we covered the notions of consistency and completeness of λ theory and the **S**, **K** and **I** combinators [12].

After simply-typed λ-calculus, I spoke about the λ-cube and the Curry-Howard isomorphism [9], as advanced topics, not assessed in the final exam.

[3] https://www.python.org
[4] https://golang.org
[5] https://www.gla.ac.uk/coursecatalogue/course/?code=COMPSCI4072

On π-calculus Since types are a crucial part of the π-calculus, we explored a plethora of types and type systems, classifying channels in more precise ways, like input/output, linear, variant etc., surveyed in [18,27].

We thoroughly discussed *session types* [14,28,15]—a type formalism used to model communication-based programming of concurrent and distributed systems. We discussed the expressiveness of session types and their encoding into linear π-calculus types [18,7,6,8].

Finally we covered equivalence of processes and the notions of trace equivalence and bisimulation and labelled transition systems (LTS) [27].

4 Discussion

In this section, I will discuss what went well and what went less well in teaching Theory of Computation, and I will try and *critically reflect* [13] on my practice in learning and teaching in relation to the *four lenses* [5]: my experience, students feedback, colleagues, and learning and teaching literature.

4.1 What went well

Teaching a small class has been shown to be beneficial for students as learners [3]. ToC is a small class, usually 15–25 students, hence interaction with students is easy, natural and more relaxed, both on the teacher and students sides.

In my practice, I interleave lecturing with *breaks*, where I ask questions and solve exercises on the whiteboard, for e.g., proofs by induction, reductions of λ- or π-terms, or typing derivations, and I invite students to volunteer to come to the whiteboard. This has been very successful, as due to the size of the class students feel comfortable to ask and reply to questions and enjoy to come to the whiteboard to solve exercises together. By adopting this approach of *interactive windows* and aligning with *active learning* [16,23], students are more focused, engaged and motivated, which is beneficial for their learning.

Another proved beneficial technique, which I adopted in class, was allowing students to work in pairs/small groups or individually [10], for about 10 minutes. This technique in particular triggered discussions and engagement in class and a better connection among students, which promoted an even more relaxed and comfortable classroom environment.

To further facilitate interaction with students by using technology [1], there are two tools which I adopted/would like to adopt in class: Padlet[6] and YACRS[7]. Padlet provides a virtual post-it board and allows discussions on specific topics. For e.g., in another course, Programming Languages (PL), I used Padlet to discuss with students the differences between compilers and interpreters, and their pros and cons. This enabled the shyer students to contribute, without being put under the spotlight. Later on, I posted an exported .pdf file of the Padlet responses on Moodle[8], used at University of Glasgow for our courses. I

[6] https://en-gb.padlet.com
[7] http://classresponse.gla.ac.uk
[8] https://moodle.gla.ac.uk

found that using Padlet in this way breaks the monotony of teaching, engages students and promotes deeper learning, as also evidenced by the literature [4,20]. The effectiveness of the tool was reflected later on in that in the PL final exam almost every student got full marks on a question on compilers *vs.* interpreters.

YACRS (Yet Another Classroom Response System) allows teachers to ask questions and students to (anonymously) answer by following a link and using their phones/tablets/laptops. When I teach I often ask "*are you following?*" to check that students are understanding. However, as discussed with colleagues at University of Glasgow, it is possible for students not to be honest when they nod/say yes, because they might be shy, or they do not want to show in class that they did not understand—a phenomenon which is common not just among students. Hence, YACRS can improve the effectiveness of teaching and check students understanding and learning, as well as regularly assess them.

From the anonymous and official students feedback on the Theory of Computation course, the combination of lectures and labs worked well as "*labs provide a good practical understanding and are a good preparation for the coursework*", which is assessed and part of the students final mark.

4.2 What went less well

Representation of natural numbers, conditionals, booleans and logical operators in λ-calculus in Python and π-calculus in Go, did not capture students interest in the same way as for e.g., implementing the mobile phones § 2.2 in π-calculus in Go. Students felt that building such primitive constructs was just a theoretical exercise, whether the mobile phones example illustrated a real-world, albeit simple, communicating and concurrent system.

Moreover, implementation of π-calculus in Go was challenging. It required more setting up than λ-calculus in Python for two reasons: *i*) as previously stated, students are taught Python from the first year, whether for most, if not all, it was the first time they used Go, and *ii*) programming with λ-calculus in Python is more natural than programming with π-calculus in Go, as Go hides the pure concurrency features of π-calculus. For e.g., representing two processes in parallel in Go is done in either of the following ways (as described in detail by Gay in the lab worksheet): *i*) one process as the main function and the other as goroutine, or *ii*) both goroutines, or *iii*) both as goroutines with anonymous functions. All three ways are hacky for the π-calculus.

Even though learning a new language was a good experience, according to the students feedback, it was difficult to focus on the π-calculus itself when using Go. Programming with λ-calculus in Python was easier than Go, but it had its drawbacks and it did not allow for a full understanding of the theory, which was again hidden under the programming language features.

5 Conclusions

As university teachers, our goal is to teach in such a way for students to become independent learners and critical thinkers—moving to *stage 5* of Kugel [19], as discussed in § 1—and in particular in computing science, to become independent *computational* thinkers. A student-centred approach to teaching [23,2] is shown to be beneficial to this aim. As such, I used *interactive windows* [16], where teaching is interleaved with whiteboard work, use of technology such as Padlet, or pair, small group or individual work. For small classes, like often is the case for formal methods, this approach to teaching is easy to achieve, and finally we can use the class size to our advantage.

About the course structure, I believe that a combination of lectures and labs, as in ToC, is the best way to deliver a formal methods course, because it allows students to explore the practical side of the theory and understand its relevance, as when implementing the mobile phones example § 2.2.

Using Python and Go in ToC, on one hand showed the practicality of the Greek-letter calculi in mainstream programming languages; on the other, it obscured their key features. To overcome this, I proposed final year student projects to implement λ-calculus and π-calculus as standalone languages, together with evaluation environments and type systems. In the academic year 2018-19, an undergraduate student implemented the π-calculus with session types, and the mobile phones example; another student implemented a web interface for the encoding of session types into linear types [7,8], together with processes and typing algorithms; a master student implemented the λ-calculus, its simple types and its evaluation environment. The goal of these tools is to capture the key features of the Greek-letter calculi when programming with them; however, they are perhaps to be introduced *alongside* programming in Python, or rather in Haskell in the next year, and Go, which are mainstream languages.

To conclude, I believe success in teaching formal methods is a combination of (i) showing the relevance and practicality of theory in modelling real-world systems, for e.g., as in mobile phones, via programming and using tools; and (ii) appropriate teaching approaches, tailored to the size of the class, which have been demonstrated to be effective by learning and teaching literature.

References

1. Bates, A.W.: Teaching in a Digital Age: Guidelines for designing teaching and learning for a digital age. BCcampus (2015)
2. Ben-Ari, M.: Constructivism in computer science education. SIGCSE Bull. **30**(1), 257–261 (1998), http://doi.acm.org/10.1145/274790.274308
3. Bogaard, A., Carey, S.C., Dodd, G., Repath, I.D., Whitaker, R.: Small group teaching: Perceptions and problems1. Politics **25**(2), 116–125 (2005), https://doi.org/10.1111/j.1467-9256.2005.00236.x
4. Bonwell, C.C., Eison, J.A.: Active learning: creating excitement in the classroom. AASHE-ERIC Higher Education Report, Washington DC: School of Education and Human Development, George Washington University (1991)
5. Brookfield, S.: Becoming a Critically Reflective Teacher. Higher and Adult Education Series, Wiley (1995)
6. Dardha, O.: Type Systems for Distributed Programs: Components and Sessions, Atlantis Studies in Computing, vol. 7. Springer / Atlantis Press (2016), https://doi.org/10.2991/978-94-6239-204-5

7. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. In: PPDP. pp. 139–150. ACM (2012), http://doi.acm.org/10.1145/2370776.2370794
8. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. Inf. Comput. **256**, 253–286 (2017), https://doi.org/10.1016/j.ic.2017.06.002
9. Girard, J., Lafont, Y., Taylor, P.: Proofs and Types. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press (1989)
10. Griffiths, S.: Teaching and learning in small groups. A Handbook for Teaching and Learning in Higher Education. 3rd Ed. London, RoutledgeFalmer (2009)
11. Gruska, J.: Quantum Computing. Advanced topics in computer science series, McGraw-Hill (1999)
12. Hankin, C.: An Introduction to Lambda Calculi for Computer Scientists. Texts in computing, Kings College (2004)
13. Hatton, N., Smith, D.: Reflection in teacher education: Towards definition and implementation. Teaching and Teacher Education **11**(1), 33 – 49 (1995), https://doi.org/10.1016/0742-051X(94)00012-U
14. Honda, K.: Types for dyadic interaction. In: CONCUR. LNCS, vol. 715, pp. 509–523. Springer (1993), https://doi.org/10.1007/3-540-57208-2_35
15. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP. LNCS, vol. 1381, pp. 122–138. Springer (1998), https://doi.org/10.1007/BFb0053567
16. Huxham, M.: Learning in lectures: Do 'interactive windows' help? Active Learning in Higher Education **6**, 17–31 (2005), https://doi.org/10.1177/1469787405049943
17. Kobayashi, N.: Type systems for concurrent programs. In: Formal Methods at the Crossroads: From Panacea to Foundational Suppor. LNCS, vol. 2757, pp. 439–453. Springer (2002)
18. Kobayashi, N.: Type systems for concurrent programs (2007), extended version of [17], Tohoku University. www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf
19. Kugel, P.: How professors develop as teachers. Studies in Higher Education - STUD HIGH EDUC **18**, 315–328 (1993), https://doi.org/10.1080/03075079312331382241
20. Mayer, R.E.: Multimedia Learning. Cambridge University Press, 2 edn. (2009), https://doi.org/10.1017/CBO9780511811678
21. Milner, R.: Communication and Concurrency. Prentice Hall, New York (1989)
22. Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press (may 1999)
23. O'Neill, G., Mcmahon, T.: Student-centred learning: What does it mean for students and lecturers? Emerging Issues in the Practice of University Learning and Teaching **1** (2005)
24. Pierce, B.C.: Types and programming languages. MIT Press, MA, USA (2002)
25. Rayward-Smith, V.: A First Course in Formal Language Theory. Computer science texts, McGraw-Hill Book Company (1995)
26. Reed, P.: Staff experience and attitudes towards technology enhanced learning initiatives in one faculty of health & life sciences. Research in Learning Technology **22** (2014), https://doi.org/10.3402/rlt.v22.22770
27. Sangiorgi, D., Walker, D.: The Pi-Calculus - a theory of mobile processes. Cambridge University Press (2001)
28. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: PARLE. LNCS, vol. 817, pp. 398–413. Springer (1994), https://doi.org/10.1007/3-540-58184-7_118
29. Weller, S.: Academic Practice Developing as a Professional in Higher Education. SAGE: London (2015)