Gradually-typed functional languages

Final exam, MPRI 2-4

2020/03/06 — Duration: 3h00

Answers are judged by their correctness, but also by their clarity, conciseness, and accuracy. You don't have to justify answers unless explicitly required. Although the questions are in English, it is permitted to answer in French.

A programmer may want to prototype a program in an untyped setting to follow the most direct path from edition to execution. For this reason, dynamic languages provide a form of dynamically-checked typing, where errors are only detected at runtime. In those languages, the programmer has no static guarantees at all about program executions.

Gradually typed languages offer to mix statically-checked and dynamically-checked terms in the same program. The programmer controls the amount of static checks by annotating terms with types: once a term is fully annotated and well-typed, it does not get stuck.

In this exam, we study the gradually-typed λ-calculus GTL. In Part 1, we start with a presentation of the syntax and the type system of GTL. The dynamic semantics of GTL is defined by an elaboration to another language called GTLC which makes explicit *casts*, i.e. dynamic checks of the type of values. We realize these casts as coercions in Part 2. We then introduce the gradually-typed λ-calculus with coercions GTLC and translate GTL to GTLC in Part 3.

To answer a question, you can assume the results of previous questions, even if you did not treat these questions.

1 The gradually-typed λ-calculus GTL

The syntax for terms and types of the gradually-typed λ-calculus GTL is given by the following grammar:

$$\begin{array}{rcll}
t & ::= & x & \text{Variable} \\
 & \mid & t\ t & \text{Application} \\
 & \mid & \lambda(x : T).t & \text{Abstraction} \\
 & \mid & \mathbf{if}\ t\ \mathbf{then}\ t\ \mathbf{else}\ t & \text{Conditional expression} \\
 & \mid & p(\bar{t}) & \text{Primitive} \\
 & \mid & c & \text{Constant} \\
 & & & \\
T & ::= & \square & \text{Unknown type} \\
 & \mid & \varepsilon & \text{Primitive type} \\
 & \mid & T \to T & \text{Arrow type}
\end{array}$$

The syntax for terms is the same as in the simply-typed λ-calculus with constants, conditional expressions, and fully-applied primitives. We assume that the primitive types ε include **int** and **bool**, that constants include integers and booleans, and that primitives include basic arithmetic operations and comparisons.

The only syntactic difference between GTL and the simply-typed λ-calculus lies in the type algebra. In GTL, the types include an unkwown type which denotes the absence of type information about (some part of) a value. The new construction $\Box$ represents a so-called dynamic type, that is, a type for a term whose compability with its context will only be checked at runtime.

Intuitively, a gradual type system only promises to check statically-known information about terms. In other words, every term that enjoys a statically unknown type is ignored by the typechecker: its evaluation may therefore produce a runtime error.

For instance, the term:

$$\lambda(x : \mathbf{bool}).(x + 1)$$

is rejected by the typechecker provided that $t_1 + t_2$ has type **int** if both t_1 and t_2 have type **int**. By contrast, the following term:

$$\lambda(x : \Box).(x + 1)$$

is accepted by the typechecker because the type of x is statically unknown.

We introduce the following definitions:

- $\mathsf{app} : (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int}$ defined as $\lambda(f : \mathbf{int} \to \mathbf{int}).\lambda(x : \mathbf{int}).f\ x$,
- $\mathsf{succ} : \mathbf{int} \to \mathbf{int}$, defined as $\lambda(x : \mathbf{int}).x + 1$, and
- $\mathsf{k} : \mathbf{bool} \to \mathbf{int}$ defined as $\lambda(x : \mathbf{bool}).\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ 0$.

There is no subtyping: for instance, using a value of type **int** where a value of type **bool** is expected produces a runtime error. We write Γ_0 for the typing environment populated with these definitions ($\mathsf{app} : (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int}, \mathsf{succ} : \mathbf{int} \to \mathbf{int}, \mathsf{k} : \mathbf{bool} \to \mathbf{int}$).

Question 1

For each of the following terms, should the term be rejected by the typechecker? Should its evaluation produce a runtime error? You do not have to justify your answers.

1. $(\lambda(x : \Box).x)\ 0$
2. $\mathsf{app}\ (\lambda(x : \Box).\mathsf{succ}\ x)\ 0$
3. $\mathsf{app}\ (\lambda(x : \Box).\mathsf{k}\ x)\ 0$
4. $\mathsf{app}\ (\lambda(x : \mathbf{bool}).\mathsf{k}\ x)\ 0$

$\Box$

A gradual type system cannot use a purely syntactic equality to determine if two types are compatible. We write $T \simeq T'$, read "T is compatible with T'", for an inductive binary relation defined by the following rules:

$$\frac{}{T \simeq T} \qquad \frac{}{\Box \simeq T} \qquad \frac{}{T \simeq \Box} \qquad \frac{T_1 \simeq T_1' \qquad T_2 \simeq T_2'}{T_1 \to T_2 \simeq T_1' \to T_2'}$$

Question 2

Is $\simeq$ an equivalence relation? Justify your answer. $\square$

Typing environments are defined as usual as $\Gamma ::= \bullet \mid \Gamma, (x : T)$. We omit the definition of the judgment $(x : T) \in \Gamma$. We assume the existence of a typing rule for each constant. The following typing rules define the judgment $\Gamma \vdash t : T$ for GTL:

$$\textsc{Var}\ \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \textsc{If}\ \frac{\Gamma \vdash t_1 : \mathbf{bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 : T}$$

$$\textsc{Primitive}\ \frac{p : T_1 \times \ldots \times T_n \to T_p \qquad \forall i \in [1..n],\ \Gamma \vdash t_i : T_i}{\Gamma \vdash p(t_1, \ldots, t_n) : T_p} \qquad \textsc{Lam}\ \frac{\Gamma, (x : T_1) \vdash t : T_2}{\Gamma \vdash \lambda(x : T_1).t : T_1 \to T_2}$$

$$\textsc{App1}\ \frac{\Gamma \vdash t_1 : \square \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1\ t_2 : \square} \qquad \textsc{App2}\ \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_3 \qquad T_1 \simeq T_3}{\Gamma \vdash t_1\ t_2 : T_2}$$

We say that a term t is well-typed in Γ if there exists a type T such that $\Gamma \vdash t : T$.

Question 3

Compare "succ true" and "$((\lambda(x : \square).x)$ succ) true" with respect to their typability in the environment Γ_0. $\square$

Let us write M for a term of the untyped λ-calculus without constants, i.e., conforming to the grammar:

$$M ::= x \mid \lambda x.M \mid M\ M$$

Question 4

Strictly speaking, can we say that any closed term M of the untyped λ-calculus can be accepted as a closed well-typed term of GTL (assuming that $\lambda x.M$ is translated as $\lambda(x : \square).M$)? Justify your answer. $\square$

Question 5

Give an inductive definition for $[\![\bullet]\!]$ such that if M is closed, then $[\![M]\!]$ is a well-typed closed term of GTL that is observationally equivalent to M. Do not justify your answer. $\square$

Question 6

Are the well-typed closed terms of the simply typed λ-calculus with constants included in the well-typed closed terms of GTL? Justify your answer. $\square$

2 A calculus of coercions

We now focus on a language of coercions. Coercions are terms which denote a runtime conversion between two types. The syntax for coercion terms is defined as follows:

$$\begin{array}{rcll} \kappa & ::= & \mathsf{id} & \text{Identity} \\ & | & T! & \text{Upcast} \\ & | & T? & \text{Downcast} \\ & | & \kappa \to \kappa & \text{Arrow} \\ & | & \kappa ; \kappa & \text{Sequence} \\ & | & \mathsf{fail} & \text{Failure} \end{array}$$

A coercion may denote an identity id which has no effect on the type, an upcast $T!$ which converts from T to $\Box$, a downcast $T?$ which tries to convert from $\Box$ to T, or an arrow cast $\kappa_1 \to \kappa_2$ which converts the input of a function with κ_1 and its output with κ_2. The composition $\kappa_1; \kappa_2$ first converts a type with κ_1 and then converts the resulting type with κ_2. Finally, a coercion failure is written fail. Such a failure will stop the program execution.

We write $\kappa : T_1 \rhd T_2$ to denote the fact that κ converts from T_1 to T_2.

Question 7

Propose a rule for $\kappa : T_1 \rhd T_2$ when $\kappa = \mathsf{fail}$. □

Question 8

Give the other rules for the judgment $\kappa : T_1 \rhd T_2$. □

Any coercion encodes a potential cast from a type to another type. Reciprocally, any cast from T_1 to T_2 can be encoded as a coercion using the function $(\!| T_1 \rhd T_2 |\!)$:

$$\begin{array}{rcl|rcl} (\!| \varepsilon_2 \rhd \varepsilon_1 |\!) & = & \mathsf{id} \text{ if } \varepsilon_1 = \varepsilon_2 & (\!| \Box \rhd \varepsilon |\!) & = & \varepsilon? \\ (\!| \varepsilon_2 \rhd \varepsilon_1 |\!) & = & \mathsf{fail} \text{ if } \varepsilon_1 \neq \varepsilon_2 & (\!| T_1 \to T_2 \rhd \varepsilon |\!) & = & \mathsf{fail} \\ (\!| \Box \rhd \Box |\!) & = & \mathsf{id} & (\!| \varepsilon \rhd T_1 \to T_2 |\!) & = & \mathsf{fail} \\ (\!| \varepsilon \rhd \Box |\!) & = & \varepsilon! & (\!| T_1 \to T_2 \rhd \Box |\!) & = & (T_1 \to T_2)! \\ & & & (\!| \Box \rhd T_1 \to T_2 |\!) & = & (T_1 \to T_2)? \end{array}$$

$$(\!| T_1 \to T_2 \rhd T_1' \to T_2' |\!) = \begin{cases} \mathsf{fail} \text{ if } (\!| T_1' \rhd T_1 |\!) = \mathsf{fail} \text{ or } (\!| T_2 \rhd T_2' |\!) = \mathsf{fail} \\ \mathsf{id} \text{ if } (\!| T_1' \rhd T_1 |\!) = \mathsf{id} \text{ and } (\!| T_2 \rhd T_2' |\!) = \mathsf{id} \\ (\!| T_1' \rhd T_1 |\!) \to (\!| T_2 \rhd T_2' |\!) \text{ otherwise} \end{cases}$$

Question 9

Do we have $(\!| T_1 \rhd T_2 |\!) : T_1 \rhd T_2$? Justify your answer. □

We equip coercions with a reduction defined by the judgment $\kappa_1 \rightsquigarrow \kappa_2$:

$$\begin{array}{rcl|rcl} T_1!; T_2? & \rightsquigarrow & (\!| T_1 \rhd T_2 |\!) & \mathsf{fail}; \kappa & \rightsquigarrow & \mathsf{fail} \\ (\kappa_1 \to \kappa_2); (\kappa_1' \to \kappa_2') & \rightsquigarrow & (\kappa_1'; \kappa_1 \to \kappa_2; \kappa_2') & \kappa; \mathsf{fail} & \rightsquigarrow & \mathsf{fail} \\ \mathsf{id}; \kappa & \rightsquigarrow & \kappa & \mathsf{fail} \to \kappa & \rightsquigarrow & \mathsf{fail} \\ \kappa; \mathsf{id} & \rightsquigarrow & \kappa & \kappa \to \mathsf{fail} & \rightsquigarrow & \mathsf{fail} \end{array}$$

To complete these (contraction) rules, we introduce coercion contexts $\mathbb{K}$ defined as $\mathbb{K} ::= [] \mid \mathbb{K}; \kappa \mid \kappa; \mathbb{K} \mid \mathbb{K} \to \kappa \mid \kappa \to \mathbb{K}$ as well as a reduction under context:

$$\frac{\kappa_1 \cong \mathbb{K}[\kappa_1'] \qquad \kappa_1 \cong \mathbb{K}[\kappa_2'] \qquad \kappa_1' \rightsquigarrow \kappa_2'}{\kappa_1 \rightsquigarrow \kappa_2}$$

where $\cong$ is equality modulo associativity of the sequencing operator ";".

Question 10
Reduce the following coercion terms:

1. $(\mathbf{int}? \to \mathbf{bool}!); (\mathbf{int}! \to \mathbf{bool}?)$
2. $(\mathbf{bool}? \to \mathbf{bool}!); (\mathbf{int}! \to \mathbf{bool}?)$

□

Question 11
Show the following lemma.

Lemma 1 *If $\kappa : T_1 \rhd T_2$ and $\kappa \rightsquigarrow \kappa'$ hold, then $\kappa' : T_1 \rhd T_2$ holds.*

□

Question 12
Let us assume without proof that the reduction of the coercion calculus is normalizing. What are the well-typed normal forms of the coercion calculus? □

We use the meta-variable g to range over the well-typed coercions in normal forms which are not fail or id.

3 GTLC, a gradually-typed λ-calculus with coercions

As a final step to give a semantics to GTL, we now introduce GTLC, a gradually-typed λ-calculus with coercions:

$$\begin{array}{rcll}
e & ::= & x & \text{Variable} \\
 & \mid & e\,e & \text{Application} \\
 & \mid & \lambda(x : T).e & \text{Abstraction} \\
 & \mid & \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 & \text{Conditional expressions} \\
 & \mid & p(\overline{e}) & \text{Primitive} \\
 & \mid & c & \text{Constant} \\
 & \mid & \langle \kappa \rangle e & \text{Coercion} \\
 & \mid & \mathbf{error} & \text{Cast error}
\end{array}$$

GTL and GTLC share the same syntax for types and almost the same syntax for terms. GTLC offers two extra constructions: one for explicit casts of e using κ, written $\langle \kappa \rangle e$, and another to denote cast errors written **error**.

The simple values s of GTLC are defined as $s ::= c \mid \lambda(x : T).e$. A simple value may be wrapped by a suspended coercion. Hence, the values v are defined as $v ::= s \mid \langle g \rangle s$. For each primitive p, we assume there exists δ_p which interprets the primitive p as a total function from values to value.

The reduction rules for GTLC are defined as follows:

$$
\begin{array}{rcll}
(\lambda(x : T).e)\, v & \rightarrow & e[x \mapsto v] & \\
p(\overline{v}) & \rightarrow & v & \text{if } \delta_p(\overline{v}) = v \\
\textbf{if } \mathsf{true} \textbf{ then } e_1 \textbf{ else } e_2 & \rightarrow & e_1 & \\
\textbf{if } \mathsf{false} \textbf{ then } e_1 \textbf{ else } e_2 & \rightarrow & e_2 & \\
\langle \mathsf{id} \rangle s & \rightarrow & s & \\
\langle \kappa \rangle s & \rightarrow & \langle \kappa' \rangle s & \text{if } \kappa \rightsquigarrow \kappa' \\
\langle \kappa \rangle \langle g \rangle s & \rightarrow & \langle g ; \kappa \rangle s & \\
\langle \mathsf{fail} \rangle s & \rightarrow & \textbf{error} & \\
(\langle \kappa_1 \rightarrow \kappa_2 \rangle s)\, v & \rightarrow & \langle \kappa_2 \rangle (s\, (\langle \kappa_1 \rangle v)) &
\end{array}
$$

Question 13

Give a syntax to evaluation contexts such that the terms of GTLC are evaluated following a call-by-value weak reduction strategy stopping when a cast error occurs. Give also the rules to evaluate a term under an evaluation context. □

Question 14

What are the outcomes of the evaluations of the following terms:

1. $\langle \textbf{bool}? \rangle \langle \textbf{int}! \rangle 42$
2. $\langle \textbf{bool}? \rangle 42$

□

The type system for GTLC reuses the same rules as the type system for the simply-typed λ-calculus. A rule must be provided for cast expressions and error expressions though.

Question 15

Give a typing rule for cast expressions and error expressions. □

Question 16

Define a type-directed translation from a typing derivation in GTL to a term of GTLC. This translation must produce a well-typed term of GTLC. □

Question 17

We assume that subject reduction holds. Prove the type soundness of GTLC by proving the progress of evaluation of well-typed terms that are not values. You will only study the case where the well-typed term is an application. □

4 Solutions

Question 1

1. Accepted and evaluates without runtime error.
2. Accepted and evaluates without runtime error.
3. Accepted but produces a runtime error.
4. Rejected and would produce a runtime error.

Question 2

No, it is not transitive since **bool** $\simeq \square$ and $\square \simeq$ **int** do not imply **bool** $\simeq$ **int**.

Question 3

The first term is not well-typed while the second is. Indeed, in the first term, there is a clash between the argument type of `succ` which is **int** and the actual argument `true` which has type **bool**. In the second term, the β-redex $(\lambda(x : \square).x)$ `succ` can be assigned the statically unknown type $\square$ hence it turns the term `succ` into a dynamically checked term.

Question 4

No, as shown by the previous question, "`succ true`" is rejected while it is a perfectly fine untyped λ-term.

Question 5

$$\begin{array}{rcl} [\![x]\!] & = & x \\ [\![\lambda x.M]\!] & = & \lambda(x : \square).[\![M]\!] \\ [\![M_1\ M_2]\!] & = & ((\lambda(x : \square).x)\ [\![M_1]\!])\ [\![M_2]\!] \end{array}$$

Question 6

The well-typed terms of the simply typed λ-calculus are included in the well-typed terms of GTL. Indeed, any typing derivation of the simply typed λ-calculus can be formed with a subset of the typing rules of GTL. Rules Var, Primitive, and Lam are syntactically identical in the two systems. The rule for applications of the simply-typed λ-calculus is an instance of the rule App2 where the type compatibility judgment is obtained by a mere reflexivity.

Question 7

Since a failure produces an error, it is safe to statically interpret it as a universal conversion:

$$\frac{}{\mathsf{fail} : T_1 \rhd T_2}\ \textsc{CC-Failure}$$

Question 8

$$\frac{}{\mathsf{id} : T \rhd T}\ \textsc{CC-Id} \qquad \frac{}{T! : T \rhd \square}\ \textsc{CC-UpCast} \qquad \frac{}{T? : \square \rhd T}\ \textsc{CC-DownCast}$$

$$\frac{\kappa_1 : T_1 \rhd T_1' \qquad \kappa_1 : T_2' \rhd T_2}{\kappa_1 \to \kappa_2 : T_1' \to T_2' \rhd T_1 \to T_2}\ \textsc{CC-Arrow}$$

$$\frac{\kappa_1 : T_3 \rhd T_2 \qquad \kappa_2 : T_2 \rhd T_1}{\kappa_1 ; \kappa_2 : T_3 \rhd T_1}\ \textsc{CC-Seq}$$

Question 9

Question 10

TODO bug

1.
$$\begin{aligned}
&(\mathbf{int}! \to \mathbf{bool}?) ; (\mathbf{int}? \to \mathbf{bool}!) && \rightsquigarrow \\
&\mathbf{int}! ; \mathbf{int}? \to \mathbf{bool}? ; \mathbf{bool}! && \rightsquigarrow \\
&\mathsf{id} \to \mathbf{bool}? ; \mathbf{bool}! && \rightsquigarrow
\end{aligned}$$

2.
$$\begin{aligned}
&(\mathbf{bool}! \to \mathbf{bool}?) ; (\mathbf{int}? \to \mathbf{bool}!) && \rightsquigarrow \\
&\mathbf{bool}! ; \mathbf{int}? \to \mathbf{bool}? ; \mathbf{bool}! && \rightsquigarrow \\
&\mathsf{fail} \to \mathbf{bool}? ; \mathbf{bool}! && \rightsquigarrow \\
&\mathsf{fail} && \rightsquigarrow
\end{aligned}$$

Question 11

By case analysis of the reduction step.

Case $T_1! ; T_2? \rightsquigarrow (\!| T_1 \rhd T_2 |\!)$. We have $T_1! ; T_2? : T_2 \rhd T_1$ by hypothesis. By Question 9, we also have: $(\!| T_1 \rhd T_2 |\!) : T_2 \rhd T_1$.

Case $(\tilde{\kappa_1} \to \tilde{\kappa_2}) ; (\tilde{\kappa_1}' \to \tilde{\kappa_2}') \rightsquigarrow (\tilde{\kappa_1} ; \tilde{\kappa_1}' \to \tilde{\kappa_2} ; \tilde{\kappa_2}')$. By inversion of the hypothesis:

- $\tilde{\kappa_1} \to \tilde{\kappa_2} : T_1 \rhd T_3 (\mathbf{1})$
- $\tilde{\kappa_1'} \to \tilde{\kappa_2'} : T_3 \rhd T_2 (\mathbf{2})$

By inversion of (1):

- $T_1 = T_{11} \to T_{12}(\mathbf{3})$, $T_3 = T_{31} \to T_{32}(\mathbf{4})$.
- $\tilde{\kappa_1} : T_{11} \triangleright T_{31}(\mathbf{5})$, $\tilde{\kappa_2} : T_{12} \triangleright T_{32}(\mathbf{6})$

By inversion of (2):

- $T_2 = T_{21} \to T_{22}(\mathbf{7})$,
- $\tilde{\kappa_1}' : T_{31} \triangleright T_{21}(\mathbf{8})$ $\tilde{\kappa_2}' : T_{32} \triangleright T_{22}(\mathbf{9})$

By applying CC-Seq on (5) and (8), we get $\tilde{\kappa_1}; \tilde{\kappa_1}' : T_{11} \triangleright T_{21}(\mathbf{10})$.
By applying CC-Seq on (6) and (9), we get $\tilde{\kappa_2}; \tilde{\kappa_2}' : T_{12} \triangleright T_{22}(\mathbf{11})$.
Finally, applying CC-Arrow on (10) and (11) give the desired conclusion.

Case $\mathit{id}; \kappa \leadsto \kappa$.

Case $\kappa; \mathit{id} \leadsto \kappa$.
These two cases are similar.

Case $\mathit{fail}; \kappa \leadsto \mathit{fail}$.

Case $\kappa; \mathit{fail} \leadsto \mathit{fail}$.

Case $\mathit{fail} \to \kappa \leadsto \mathit{fail}$.

Case $\kappa \to \mathit{fail} \leadsto \mathit{fail}$.
All these cases are trivial since $\mathsf{fail} : T_2 \triangleright T_1$ for any T_1 and T_2.

Question 12

Question 14