Delimited control in System F

Mid-term exam, MPRI 2-4

2020/12/02 — Duration: 2h45

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 either French or English—and recommended to answer in French if this is your mother language.

Part 3 is more difficult and we recommend to do it only after the other parts.

In this problem, we study a computation mechanism called *delimited control* that has some similarity with but is more general than the exception mechanism. Similarly to exceptions, delimited control allows to suspend the current flow of control to jump to some earlier execution point where a handler has been placed, but it also captures the continuation delimited by the handler before aborting so that, after some treatment, the computation may be resumed at the location where it aborted.

1 Syntax and semantics

We assume given a denumerable collection of atomic prompts $\pi \in \mathcal{P}$. We call F_C the extension of explicitly-typed System F with new constructs for control (in red) :

$$\begin{array}{rcl} M & ::= & V \mid M\ M \mid M\ \tau \mid \#\ M\ M \mid \mathsf{control}\ M\ M \\ V & ::= & x \mid () \mid \lambda x : \tau.\, M \mid \Lambda\alpha.V \mid \pi \mid \mathsf{new\_prompt}\ \tau \\ \tau & ::= & \alpha \mid \mathsf{unit} \mid \tau \to \tau \mid \forall\alpha.\,\tau \mid \mathsf{pr}\ \tau \end{array}$$

The expression $()$ has type unit. Intuitively, the new constructs behave as follows :

— $\mathsf{new\_prompt}\ \tau\ ()$ allocates and returns a fresh prompt (of type $\mathsf{pr}\ \tau$) ;

— $\#\ M_1\ M_2$ evaluates a_1 to a prompt π, then evaluates a_2 under π ;

— $\mathsf{control}\ M_1\ M_2$ evaluates a_1 to a prompt π and a_2 to a value v, then captures (and removes) the evaluation context up to the prompt π and turns it into a function that is passed to v.

Formally, we give a call-by-value semantics with a usual left-to-right evaluation order. We need a store μ to track already allocated prompts. Thus, the semantics is a reduction relation on configurations of the form $M \,/\, \mu$.

Evaluation contexts of depth 1, written E, are defined by the grammar :

$$\begin{array}{rcl} E & ::= & [\,]\ M \mid V\ [\,] \mid [\,]\ \tau \\ & \mid & \#\ [\,]\ M \mid \#\ V\ [\,] \mid \mathsf{control}\ [\,]\ M \mid \mathsf{control}\ V\ [\,] \end{array}$$

Given a particular prompt π, we write E_π for a context of depth 1 that is distinct from $\#\ \pi\ [\,]$. (Such a context can be $\#\ \pi'\ [\,]$ where π and π' are distinct prompts.) Arbitrarily deep evaluation contexts F, as well as those, written F_π, that do not contain $\# \pi$ [ ] on the path from the root to the hole are defined as follows :

$$F ::= [\,] \mid F[E] \qquad \text{and} \qquad F_\pi ::= [\,] \mid F_\pi[E_\pi]$$

Question 1

Recall the reduction rules of the standard construct of System F in this setting. □

The new reduction rules are

$$\begin{array}{rcllr}
\mathsf{new\_prompt}\ \tau\ ()\ /\ \mu & \longrightarrow & \pi\ /\ \mu \cup \{\pi\} & \pi \notin \mu & (\textsc{New}) \\
\#\ \pi\ V\ /\ \mu & \longrightarrow & V\ /\ \mu & & (\textsc{Return}) \\
\#\ \pi\ (F_\pi[\mathsf{control}\ \pi\ V])\ /\ \mu & \longrightarrow & V\ (\lambda x.\, F_\pi[x])\ /\ \mu & x \notin F_\pi & (\textsc{Control})
\end{array}$$

Question 2

Could $\mathsf{new\_prompt}\ \tau$ have been treated as a constant ? If so, would it be a constructor or a destructor, and of which arity ? Could $\#\ \_\ \_$ have been treated as a constructor ? (Justify briefly, in just one or two sentences.) □

Question 3

Propose a well-formedness property that one can impose on configuration. Explain why this property is preserved during reduction. □

We write $M \longrightarrow M'$ to mean that $M\ /\ \emptyset \longrightarrow M'\ /\ \mu$ for some store μ when we do not care about prompt allocations.

We write $\lceil F_C \rceil$ for the implicitly-typed version of F_C. We write M, N for explicitly-typed terms and a, b for implicitly-typed terms ; we write v for implicitly-typed values. (We overload the notation for contexts and do not distinguish between implicitly-typed and explicitly-typed contexts.)

For implicitly-typed terms, we use let-expressions $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ as syntactic sugar for $(\lambda x.\, a_2)\ a_1$. We may write $\lambda\_.\, a$ for $\lambda k.\, a$ when k is fresh for a. In the examples, we may also use integers and numerical operations, as well as Booleans and conditionals.

In the rest of this section, we work in $\lceil F_C \rceil$, that is, we omit type annotations on source terms.

Question 4

Give the step-by-step reduction (*i.e.* one redex at a time) of the configuration

$$\mathsf{let}\ y = \mathsf{new\_prompt}\ ()\ \mathsf{in}\ 4 - \#\ y\ (1 - \mathsf{control}\ y\ (\lambda k.\, (2 - k\ 3)))\ /\ \emptyset$$

□

Question 5

Let F, F_1, and F_2 be evaluation contexts of System F, *i.e.* that do not contain control operations, and v, v_1, and v_2 be values of System F such that $F_1[v]$ reduces to v_1 and $F_2[v]$ reduces to v_2.

Let a be the term :

$$F\left[\begin{array}{l}\mathsf{let}\ p = \mathsf{new\_prompt}\ ()\ \mathsf{in}\\ \mathsf{let}\ q = \mathsf{new\_prompt}\ ()\ \mathsf{in}\\ \mathsf{let}\ \mathit{return} = \lambda x.\,(\mathsf{control}\ q\ (\lambda\_.\,x))\ \mathsf{in}\\ \mathsf{let}\ \mathit{handle} = \lambda x.\,F_2[x]\ \mathsf{in}\\ \#\ q\ (\mathit{handle}\ (\#\ p\ (\mathit{return}\ (F_1[\mathsf{control}\ p\ (\lambda\_.\,v)]))))\end{array}\right]$$

where variable x is fresh for F_2.

a) Give a configuration $a' \,/\, \mu$ so that $a \,/\, \emptyset \longrightarrow^* a' \,/\, \mu$ where a' is reduced as far as possible—but not necessarily fully evaluated, given that we do not know the exact form of contexts and values. (You do not have to give the intermediate evaluation steps.)

b) Give an expression a_1 that uses the simple exception mechanism described in the course instead of the new control operations and still behaves very much like a. □

Question 6

Assume given a prompt π of some fixed type τ_{exn}. To make the previous correspondence more explicit, we wish to write two functions myraise and mytry in F_C (*i.e.* without using exceptions) so that myraise a behaves as raise a and mytry $(\lambda\_.\,a)$ b behaves as try a with b. Notice that the expression a is passed “wrapped in a thunk” as $\lambda\_.\,a$ to mytry. Why is this so ? Give the code of myraise and mytry . □

2 Type soundness

Here are some of the new typing rules associated with the delimited control operators :

Void
$$\Gamma \vdash () : \mathsf{unit}$$

NewPrompt
$$\Gamma \vdash \mathsf{new\_prompt}\ \tau : \mathsf{unit} \to \mathsf{pr}\ \tau$$

Set
$$\frac{\Gamma \vdash M_1 : \mathsf{pr}\ \tau \qquad \Gamma \vdash M_2 : \tau}{\Gamma \vdash \#\ M_1\ M_2 : \tau}$$

Control
$$\frac{\Gamma \vdash M_1 : \mathsf{pr}\ \tau \qquad \Gamma \vdash M_2 : ?}{\Gamma \vdash \mathsf{control}\ M_1\ M_2 : \sigma}$$

Question 7 (Typing rule)

(Easy, but be aware that other questions depend on this.)

Give the missing premise in the Control typing Rule. □

We wish to prove type soundness for this extension. As for references, we need to define a typing judgment for configurations, using a store typing Σ that binds prompts to types, with the following typing rules :

Prompt
$$\frac{\Sigma(\pi) = \tau}{\bar\alpha, \Sigma, \Gamma \vdash \pi : \mathsf{pr}\ \tau}$$

Config
$$\frac{\bar\alpha, \Sigma \vdash M : \tau \qquad \bar\alpha \vdash \mu : \Sigma}{\bar\alpha \vdash M \,/\, \mu : \tau}$$

Store
$$\frac{\mu \subseteq \mathit{dom}\,\Sigma \qquad \bar\alpha \vdash \Sigma}{\bar\alpha \vdash \mu : \Sigma}$$

where $\bar\alpha \vdash \Sigma$ means well-formedness, as defined in the course which requires that all free type variables of Σ are included in $\bar\alpha$.

Question 8 (Termination)

Can reduction sequences be infinite in F_C ? (Justify briefly.) $\square$

Question 9 (Value restriction)

The language has the value restriction. a) Say what it is. b) Give an example of a well-typed term M_1 that would reduce to a stuck term if we did not have this restriction. *(Note : this question may be difficult, but the other questions do not depend on it.)* $\square$

Question 10 (Subject reduction)

In order to help for subject reduction, we wish to prove separately that the reduction rule Control preserves typings.

State the property precisely and prove it, with all the details.

You may admit the basic lemmas, but you need to name them. If needed, you may also use the strengthened version of compositionality :

(Compositionality, strengthened) *If $\bar{\alpha}, \Sigma, \Gamma \vdash F[M] : \tau$ then, there exists σ such that $\bar{\alpha}, \Sigma, \Gamma \vdash M : \sigma$ and, for any M' and extension Γ' of Γ with expression variable bindings verifying $\bar{\alpha}, \Sigma, \Gamma' \vdash M' : \sigma$, we have $\bar{\alpha}, \Sigma, \Gamma' \vdash F[M'] : \tau$.* $\square$

Question 11 (Progress)

State the progress lemma and prove it : you should carefully describe the structure of the proof and only detail the case for $\#\ a_1\ a_2$; other cases can just be omitted. You may admit the basic lemmas, but you need to name them. $\square$

3 Encoding references

The goal of this part is to show that F_C, extended with one datatype, can encode references. We actually work again in $\lceil F_C \rceil$. Given two evaluation contexts F_1 and F_2, we write $F_1 \cdot F_2$ for $F_1[F_2]$. Notice that $\cdot$ is associative, which allows to decompose contexts in different manners. We also write $\#_\pi$ for the context $\#\ \pi\ [\,]$.

We write ρ instead of μ for the store defining references to avoid confusion with the store defining prompts. The idea is to represent a configuration $F[a]\ /\ \rho$ as $S_\rho \cdot F[a]$. That is, the store is encoded in an evaluation context S_ρ placed in front of the normal evaluation context F. Thus, the store is accessible during the whole computation of $F[a]$. We grow the store on the left just because it is easier to do so.

To help grow the store, we therefore need access to the toplevel, which we will obtain by running the program under a distinguished toplevel prompt t. Thus, we will be reducing configurations of the form $\#_t \cdot S_\rho \cdot F$. The context S_ρ (or just S for short) is itself a sequence of small cell contexts of the form :

$$C_{\#_{\ell_1}} \cdot \#_{\ell_1} \cdot A_t \cdot \quad \ldots \quad C_{\#_{\ell_n}} \cdot \#_{\ell_n} \cdot A_t$$

Each $C_{\#_{\ell_i}} \cdot \#_{\ell_i} \cdot A_t$ encodes a piece of store $\ell_i \mapsto v_i$ and $C_{\#_{\ell_1}} \cdot \#_{\ell_1} \cdot A_t$ is the one allocated last. You may ignore the little piece of context A_t for now—you just need to know that it does not set any prompt.

The allocation of a new cell, which (for the moment) takes a (toplevel) prompt t as argument will perform the reduction :

$$\#_t \cdot S \cdot F[\mathit{refat}\, t\ v] \longrightarrow^\star \#_t \cdot C_v \cdot \#_\ell \cdot A_t \cdot S \cdot F[\ell] \qquad \textit{(where ℓ is a fresh prompt)}$$

We actually do not need see the separation between S and F, and can treat $S \cdot F$ as a simple context F_t that does not set t. Thus, we shall have :

$$\#_t \cdot F_t[\mathit{refat}\, t\ v] \longrightarrow^\star \#_t \cdot C_v \cdot \#_\ell \cdot A_t \cdot F_t[\ell] \qquad \textit{(ℓ fresh)} \qquad (\textsc{Refat})$$

We will come back to the implementation of *refat* below.

We give you an (incomplete) implementation of the function *cell* in OCaml, such that the context $C_v[\,]$ is equivalent to *cell* v $[\,]$.

```
let rec cell v (cmd : 'a command) =
    match cmd with
    | Read k         →  $a_{read}$
    | Write (w, k)   →  $a_{write}$
```

It uses a type command α, defined in OCaml (using postfix notation consistency) as :

$$\textbf{type}\ \mathsf{command}\ \alpha = \ \mathit{Read}\ \ \textbf{of}\ \tau_{read} \mid \mathit{Write}\ \textbf{of}\ \tau_{write}$$

The operations (!) and (:=) should behave as follows :

$$\begin{array}{lll} C_v \cdot \#_\ell \cdot F_\ell[!\ \ell] & \longrightarrow^\star & C_v \cdot \#_\ell \cdot F_\ell[v] \\ C_v \cdot \#_\ell \cdot F_\ell[\ell := w] & \longrightarrow^\star & C_v \cdot \#_\ell \cdot F_\ell[()] \end{array}$$

where the context F_ℓ is of the form $A_t \cdot S_\ell \cdot F$, *i.e.* it includes A_t and may contain a sequence S_ℓ of previously allocated cells (hence that do not set the prompt ℓ) and a pure context F that does not set any prompt.

In fact, looking in more details, these reductions should have the following intermediate states :

$$\begin{array}{lllll} C_v \cdot \#_\ell \cdot F_\ell[!\ \ell] & \longrightarrow^\star & C_v \cdot \mathit{Read}\ \langle \#_\ell \cdot F_\ell \rangle & \longrightarrow^\star & C_v \cdot \#_\ell \cdot F_\ell[v] \\ C_v \cdot \#_\ell \cdot F_\ell[\ell := w] & \longrightarrow^\star & C_v \cdot \mathit{Write}\ (w, \langle \#_\ell \cdot F_\ell \rangle) & \longrightarrow^\star & C_w \cdot \#_\ell \cdot F_\ell[()] \end{array}$$

where $\langle F \rangle$ is just an abbreviation for the reification of the context as a function $\lambda x.\, F[x]$. That is, a command receiving the current continuation with its prompt is passed to the small cell interpreter before the computation is resumed.

Question 12

The operation $\mathsf{control}\ \pi\ v$ captures the continuation up to the prompt π, but π excluded. Define a function *deepcontrol* that behaves as follows :

$$\#\ \pi\ (F_\pi[\mathit{deepcontrol}\ \pi\ v])\ /\ \mu \longrightarrow^\star v\ (\lambda x.\, \#\ \pi\ (F_\pi[x]))\ /\ \mu \qquad (\textsc{Deep-Control})$$

□

Question 13 (Using reference cells)

Give the definitions of the operations (!) and (:=). □

Question 14 (Implementing the cell)

Give the expressions a_{write} and a_{read}. $\square$

Question 15 (The helper datatype)

Give the types τ_{write} and τ_{read}. $\square$

Question 16 (Hiding the store)

We now return to the small A_t context parameterized by the toplevel prompt added after each cell allocation, as described in rule REFAT). Its role is actually to skip the store interpreter when returning normally from the computation by aborting to the toplevel prompt.

Give a (binary) function *abort* so that $abort\, t\, [\,]$ is equivalent to A_t $\square$

Question 17 (Allocating a new cell)

Write the function *refat* that satisfies Rule REFAT, given above. $\square$

Question 18 (Wrapping up)

Finally, we need to wrap the execution of the program in a toplevel prompt.

Write a function *run* so that the program $run\ (\lambda ref.\, a)$ behaves as a with primitive references. (Notice that a should use *ref* and not *refat* to allocate reference cells.) $\square$

Question 19

You do not need to have solved all the preceding questions to be able to answer this one

Does this encoding of references tell you something about termination of programs in F_C as initially defined ? (Justify briefly.) $\square$

4 Solutions

Question 1

$$\begin{array}{rcl} (\lambda x:\tau.\,M)\ V \,/\, \mu & \longrightarrow & M[x \leftarrow V] \,/\, \mu \\ (\Lambda\alpha.V)\ \tau \,/\, \mu & \longrightarrow & V[\alpha \leftarrow \tau] \,/\, \mu \end{array} \qquad \frac{M \,/\, \mu \longrightarrow M' \,/\, \mu'}{E[M] \,/\, \mu \longrightarrow E[M'] \,/\, \mu'} \qquad \square$$

Question 2

Yes, it could be treated as a constant, and more precisely as a destructor of arity 1 : when applied to the unit value, it reduces to a freshly allocated prompt.

Question 2 (continued)

No, it cannot be treated as a constructor, since when fully applied to two values, it reduces. (Besides, it interacts with $\mathsf{control}\ \_\ \_$.)

Question 3

In a configuration M/μ, all prompts that appear in M should be in μ. This invariant is obviously preserved by the reduction rules : the only rule that introduces a new prompt in a term also adds it to the store and not rule ever removes a prompt from μ. The invariant is also true of the initial configuration if the source program does not mention prompts at all.

Question 4

$$\begin{array}{rl} \mathsf{let}\ p = \mathsf{new\_prompt}\ ()\ \mathsf{in}\ 4 - \#\ p\ (1 - \mathsf{control}\ p\ (\lambda k.\,(2 - k\ 3)))\,/\,\emptyset & \longrightarrow \\ \mathsf{let}\ p = \pi\ \mathsf{in}\ 4 - \#\ p\ (1 - \mathsf{control}\ p\ (\lambda k.\,(2 - k\ 3)))\,/\,\{\pi\} & \longrightarrow \\ 4 - \#\ \pi\ (1 - \mathsf{control}\ \pi\ (\lambda k.\,(2 - k\ 3)))\,/\,\{\pi\} & \longrightarrow \\ 4 - (\lambda k.\,(2 - k\ 3))\ (\lambda x.\,(1 - x))\,/\,\{\pi\} & \longrightarrow \\ 4 - (2 - (\lambda x.\,(1 - x))\ 3)\,/\,\{\pi\} & \longrightarrow \\ 4 - (2 - (1 - 3))\,/\,\{\pi\} & \longrightarrow \\ 4 - (2 - (-2))\,/\,\{\pi\} & \longrightarrow \\ 4 - 4\,/\,\{\pi\} & \longrightarrow \\ 0\,/\,\{\pi\} & \end{array}$$

Question 5

This evaluates to $F[v_2]\,/\,\{\pi, \pi'\}$ for some prompts π and π'.

Question 5 (continued)

$$F[\mathsf{try}\ F_1[\mathsf{raise}\ v]\ \mathsf{with}\ \lambda x.\, F_2[x]]$$

Question 6

The computation a must be wrapped in a thunk because $\mathsf{mytry}\ a\ b$ is not a primitive construct but an application, so the arguments a and b are evaluated before reduction : in particular a is evaluated before the handler b has been "installed".

Question 6 (continued)

$$\begin{array}{rcl}
\mathsf{myraise} & \overset{\mathsf{def}}{=} & \lambda x.\, \mathsf{control}\ \pi\ (\lambda\_.\, x) \\
\mathsf{mytry} & \overset{\mathsf{def}}{=} & \lambda x.\, \lambda handle. \\
 & & \quad \mathsf{let}\ q = \mathsf{new\_prompt}\ ()\ \mathsf{in} \\
 & & \quad \mathsf{let}\ return = \lambda x.\, (\mathsf{control}\ q\ (\lambda\_.\, x))\ \mathsf{in} \\
 & & \quad \#\ q\ (handle\ (\#\ \pi\ (return\ (x\ ()))))
\end{array}$$

Question 7

$$\dfrac{\textsc{Control} \\ \Gamma \vdash M_1 : \mathsf{pr}\ \tau \qquad \Gamma \vdash M_2 : (\sigma \to \tau) \to \tau}{\Gamma \vdash \mathsf{control}\ M_1\ M_2 : \sigma}$$

Question 8

Our solution to Question 6 is indeed well-typed for any type τ_{exn}. Hence, reduction sequences can be infinite in F_C since F_C can simulate simple exceptions, which enable to write fix-points, as seen in the course.

Question 9

We only allow generalization on value forms V (which include variables).

Question 9 (continued)

Without this restriction, we would allow to have polymorphic prompts, which could then be set at a type and used for control at another type. For example, take M equal to :

$$\begin{array}{l}\mathsf{let}\ p : \forall\alpha.\ \mathsf{pr}\ (\alpha \to \alpha) = \Lambda\alpha.(\mathsf{new\_prompt}\ (\alpha \to \alpha)\ ())\ \mathsf{in} \\ \big(\#\ (p\ \mathit{int})\ \big(\mathsf{let}\ x : \sigma = \mathsf{control}\ (p\ \mathit{bool})\ (\lambda(k : \tau).\ \mathit{not})\ \mathsf{in}\ \mathit{incr}\big)\big)\ \ 0\end{array}$$

where τ is $(\sigma \to (\mathit{bool} \to \mathit{bool})) \to (\mathit{bool} \to \mathit{bool})$. and σ is any type. This is well-typed, while $M\ /\ \emptyset$ reduces to the stuck configuration $\mathit{not}\ 0\ /\ \{\pi\}$.

Question 10

Lemma If $\bar\alpha \vdash \#\ \pi\ (F_\pi[\mathsf{control}\ \pi\ v])\ /\ \mu : \tau$ (**1**) then $\bar\alpha \vdash v\ (\lambda x.\, F_\pi[x])\ /\ \mu : \tau$ (**2**).

Proof By inversion of configuration typing, (1) implies that there exists a store typing Σ of domain μ (**3**) such that $\bar\alpha, \Sigma \vdash \#\ \pi\ (F_\pi[\mathsf{control}\ \pi\ v]) : \tau$. In turn, by inversion of typing (Rule Set), this implies that $\bar\alpha, \Sigma \vdash \pi : \mathsf{pr}\ \tau$ and $\bar\alpha, \Sigma \vdash F_\pi[\mathsf{control}\ \pi\ v] : \tau$ (**4**).

We now show that there exists σ such that $\bar\alpha, \Sigma \vdash \mathsf{control}\ \pi\ v : \sigma$(**5**) and $\bar\alpha, \Sigma, x : \sigma \vdash F_\pi[x] : \tau$ (**6**). These properties directly follow by compositionality applied with $\emptyset$ for Γ and $x : \sigma$ for Γ' applied to (4), taking $x : \sigma$ for Γ', since then $\bar\alpha, \Sigma, x : \sigma \vdash x : \sigma$.

By inversion of typing rules Control and Store applied to (5), we have $\bar\alpha, \Sigma \vdash v : (\sigma \to \tau) \to \tau$. By the abstraction typing rule applied to (6), we have $\bar\alpha, \Sigma \vdash \lambda x.\, F_\pi[x] : \sigma \to \tau$. Hence, by the application typing rule, we have $\bar\alpha, \Sigma \vdash v\ (\lambda x.\, F_\pi[x]) : \tau$, which together with (3) implies (2), as expected.

Question 11

Progress Suppose $\bar\alpha \vdash M\ /\ \mu : \tau$ and $M\ /\ \mu$ cannot be reduced. Then M is either a value or an *undelimited control effect* of the form $F_\pi[\mathsf{control}\ \pi\ V]$.

Proof The proof is by structural induction on M. From the hypothesis, by inversion of configuration typing, we have $\Gamma \vdash M : \tau$ (**1**) where Γ is $\bar\alpha, \Sigma$ for some Σ such that $\mathit{dom}\,\Sigma = \mu$.

Case $\#\ M_1\ M_2$. By inversion of typing we have $\Gamma \vdash M_1 : \mathsf{pr}\ \tau$ (**2**) and $\Gamma \vdash M_2 : \tau$ for some type τ. The configurations $M_1\ /\ \mu$ and $M_2\ /\ \mu$ are well-typed in environment $\bar\alpha$. If M_1 is an undelimited control, then so is M. Otherwise, by IH, M_1 is a value V. By the classification lemma applied to (2), it must be a prompt π. If M_2 is a value, then $M\ /\ \mu$ reduces by Rule Set. Otherwise, by IH, M_2 must be an undelimited control $F_\pi[\mathsf{control}\ \pi'\ V]$. If π' is π, then $M\ /\ \mu$ reduces by Rule Control. Otherwise, $M\ /\ \mu$ is also an undelimited control.

Other cases : omitted.

Question 12

$$\mathit{deepcontrol} \quad \overset{\mathsf{def}}{=} \quad \lambda p.\, \lambda w.\, \mathsf{control}\ p\ (\lambda k.\, w\ (\lambda x.\, \#\ p\ (k\ x)))$$

Question 13

$$\begin{aligned}(!) &\overset{\mathsf{def}}{=} \lambda r. \qquad \mathit{deepcontrol}\, r\ (\lambda k.\, \mathsf{Read}\ k)\\ (:=) &\overset{\mathsf{def}}{=} \lambda r.\ \lambda w.\, \mathit{deepcontrol}\, r\ (\lambda k.\, \mathsf{Write}\ (w, k))\end{aligned}$$

Question 14

$$\begin{aligned}a_{read} &\overset{\mathsf{def}}{=} \mathit{cell}\ \ v\ (k\ v)\\ a_{write} &\overset{\mathsf{def}}{=} \mathit{cell}\ \ w\ (k\ ())\end{aligned}$$

Question 15

$$\begin{aligned}\tau_{read} &\overset{\mathsf{def}}{=} \alpha \to \mathsf{command}\ \alpha\\ \tau_{write} &\overset{\mathsf{def}}{=} \alpha \times (\mathsf{unit} \to \mathsf{command}\ \alpha)\end{aligned}$$

Question 16

$$\mathit{abort} \overset{\mathsf{def}}{=} \lambda t.\, \lambda x.\, \mathsf{control}\ t\ (\lambda\_.\, x)$$

Question 17

```
let $refat\ \ t\ \ v =$
  let $r = new\_prompt\ ()$ in
  $control\ \ t\ \ (\lambda k.\ \ set\ t\ (cell\ v\ (set\ r\ (abort\ t\ (k\ \ r)))))$
```

Question 18

```
let $run\ a =$
  let $t = new\_prompt\ ()$ in
  $set\ \ t\ \ (a\ (refat\ \ t))$
```

Question 19

We can encode references, so we can write programs that do not terminate. However, this encoding of references uses the datatype command α which is recursive. So it does not tell us anything on F_C as initially defined, without recursive types.

(In fact, the definition of command α only uses positive recursive occurrences, which alone happens not to be sufficient to introduce non termination.)