Types and Programming Languages (15-814), Fall 2018
Assignment 3: Hacking with recursion

Contact: 15-814 Course Staff

Due Tuesday, October 9, 2018

This assignment is due by 23:59 on the above date and it must be submitted electronically as a PDF file on Canvas. Please use the attached template to typeset your assignment and make sure to include your full name and Andrew ID. As before, problems marked "WB" are subject to the whiteboard policy; all other problems must be done individually.

We have again provided you the syntax, statics, and dynamics for our simple language from class. Please ensure that your terms are syntactically correct and that they have the right type! For your convenience (and to make your programs easier to read), we have included finite products. Finite products generalize binary products in a manner analogous to the way finite sums generalize binary sums. If you wish, you are free to continue using the specialized forms of syntax ($\langle e_1, e_2\rangle$, $\langle\!|e_1, e_2|\!\rangle$, $e{\cdot}l$, $e{\cdot}r$, **case** e_1 $\{\langle x_1, x_2\rangle \Rightarrow e_2\}$, etc.) we saw for binary products. The rules for these are given in the appendices of Assignment 2.

General hint. Consider whether the results from a given task can be used to solve subsequent tasks!

Task 1 (0 points)**.** How long did you spend on this assignment? Please list the questions that you discussed with classmates using the whiteboard policy.

Streams

In lecture 8, we defined the type τ **list** of lists of terms of type τ. In this section, we will explore the type τ **stream** of *streams*, i.e., lists of (countably) infinite length. One could define τ **stream** $=$ **nat** $\to \tau$. This is a mathematically sensible definition, but it unsatisfactory for several reasons. First, it does not capture the operational aspect that a stream should be a sequence of values observed in

succession, with no means of rewinding it or going back in time. However, with the above definition, we can access any element of the stream at any time. First, this definition of τ **stream** makes it difficult to write efficient stream transducers (functions of type (τ **stream**) $\to$ (τ' **stream**)). Indeed, consider a transducer that takes a stream of **nat** as input and produces a running sum. One could implement it as follows:

$$\lambda S.\mathbf{fix}(f.\lambda n.\mathbf{case}\ \mathbf{unfold}(n)\ \{\mathsf{z}\cdot\_ \Rightarrow S(\overline{0}) \mid \mathsf{s}\cdot n' \Rightarrow \mathsf{plus}\,(Sn)\,(fn')\}\}).$$

This transducer produces the n-th entry by recursively computing the sum of the first n entries in S. But because we observe a stream S by observing the ordered sequence of values $S(\overline{0}), S(\overline{1}), \ldots, S(\overline{n}), \ldots$, this means that at each step, we must recompute the running sum up to that point.

Inherent in our operational intuition that streams are infinite sequences of values is that a stream should be a recursive type. At any point, we can observe the value at the start in the sequence (the "head" of the stream), or let it go past us and observe the remainder of the sequence (the "tail" of the stream). This means that an inhabitant of type τ **stream** is composed of a head of type τ and a tail of type τ **stream**. This is analogous to lists, whose type was defined to be:

$$\tau\ \mathbf{list} = \rho(\alpha.\,(\mathsf{nil} : \mathbf{1}) + (\mathsf{cons} : \tau \otimes \alpha)).$$

Given a list $l = \mathbf{fold}(\mathsf{cons}\cdot\langle h, t\rangle) : \tau\ \mathbf{list}$, we could observe its head $h : \tau$ and its tail $t : \tau\ \mathbf{list}$. Observe that the sequential nature of lists is captured operationally here as well: given only the tail t, there is no way to "rewind" t to recover l. Consequently, it seems sensible to try to define τ **stream** as a recursive type.

Task 2 (5 points, WB)**.** Define the type τ **stream** as a recursive type.

Hint. Study the type τ **list** and consider the dynamics of the types used to define it. For your definition of τ **stream** to make sense, there must exist values of type τ **stream**.

Let the type of bits be **bit** $= (b_0 : \mathbf{1}) + (b_1 : \mathbf{1})$. Abbreviate the bits as $\mathtt{0} = b_0 \cdot \langle\,\rangle$ and $\mathtt{1} = b_1 \cdot \langle\,\rangle$.

Task 3 (10 points, WB)**.** Implement (in any order) the following streams and functions.

1. *zeros* : **bit stream**. The stream whose every entry is the value 0.
2. *hd* : τ **stream** $\to \tau$. Extracts the head (the first entry) of a stream.
3. *tl* : τ **stream** $\to \tau$ **stream**. Drops the first entry from a stream.

4. *map* : $(\tau \to \sigma) \to \tau$ **stream** $\to \sigma$ **stream**. Applies a function $\tau \to \sigma$ to every entry of a stream of type τ **stream** to generate a stream of type σ **stream**.

5. *zip* : $(\tau$ **stream** $\otimes \sigma$ **stream**$) \to (\tau \otimes \sigma)$ **stream**. "Zips" or merges two streams together entry-wise. Explicitly, the i-th entry of the output stream is the pair of the i-th entries of the input streams.

Rather than manually encoding streams each time as you did above, it would be nice to have functions to help us create them. The simplest is the basic stream generating function

$$\mathit{iterates} : \tau \to (\tau \to \tau) \to \tau \textbf{ stream}.$$

Given an $e : \tau$ and an $f : \tau \to \tau$, *iterates* $e\, f$ generates the stream whose entries are the iterates of f applied to e. Explicitly, its first entry is e, and its $(n+1)$-th entry is $f(e_n)$ where e_n is its n-th entry. Equivalently, the n-th entry (counting from 0) is the n-fold application $f(\cdots(f(e))\cdots)$ of f to e.

This function alone is insufficient for generating interesting streams. Consider for example the stream *thirds* whose every third bit is `1` and all other bits are `0`. Because the function iterated by *iterates* is only applied to the value of the previous entry, it cannot keep track of its position in the stream.

We can solve this problem by keeping and mutating state, and generating a stream of observations of that state. To do so, we generalize *iterates* to the stateful stream-generating function

$$\mathit{strgen} : \sigma \to (\sigma \to \sigma) \to (\sigma \to \tau) \to \tau \textbf{ stream}.$$

Given an $s : \sigma$ (the "state"), a $t : \sigma \to \sigma$ (the "state transformer"), and an $f : \sigma \to \tau$ (the "state observer'), *strgen* $s\, t\, f$ generates the stream satisfying:

- $\mathit{hd}(\mathit{strgen}\ s\, t\, f) = f(s)$,
- $\mathit{tl}(\mathit{strgen}\ s\, t\, f) = \mathit{strgen}\ (t s)\, t\, f$.

As an example, we can use *strgen* to implement *thirds* as follows:

$$\mathit{strgen}\ 0\ \mathit{succ}\ (\lambda n.\textbf{if } n \equiv 0 \pmod 3 \textbf{ then } \texttt{1} \textbf{ else } \texttt{0}).$$

Task 4 (5 points, WB)**.** Define the stream generating functions

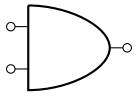
$$\mathit{iterates} : \tau \to (\tau \to \tau) \to \tau \textbf{ stream}$$
$$\mathit{strgen} : \sigma \to (\sigma \to \sigma) \to (\sigma \to \tau) \to \tau \textbf{ stream}.$$

Digital logic

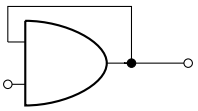
A τ gate is a term of type

$$\tau\ \mathbf{gate} = \tau\ \mathbf{stream} \otimes \tau\ \mathbf{stream} \to \tau\ \mathbf{stream}.$$

The input and output streams represent a sequence of values, one per time step. Typically, we are interested in *logic gates*: certain expressions of type **bit gate**. Gates have a nice pictorial representation (below we see an "AND gate") where the input bit streams are the two wires entering the flat part of the gate on the left, and the output bit stream is the wire exiting on the tip of the gate on the right:



By introducing multiple gates and connecting wires between their outputs and inputs, one can construct various interesting circuits. For example, we can capture a simple form of recursion using "feedback", i.e., by feeding the output wire of a gate back into one of its input wires to form the circuit:



There is a subtlety here: what should be the top "recursive" input when the gate begins processing the bottom input stream? To send a bit out on the output wire that feeds back into the top input, the gate must first have received two inputs, but one of those inputs is from the output wire. Consequently, we must bootstrap the recursion by specifying an initial value for the output wire.

Task 5 (10 points, WB)**.** Write a function

$$f : \tau\ \mathbf{gate} \to \tau \to \tau\ \mathbf{stream} \to \tau\ \mathbf{stream}$$

that connects a gate's output wire to one of its input wires, using the argument of type τ to bootstrap the recursion. Use f to implement a transducer *step* : $\mathbf{nat\ stream} \to \mathbf{nat\ stream}$ that at each step returns the largest natural number seen so far. For example, *step* should output $3, 10, 10, 10, 15, 15, \ldots$ for the stream $3, 10, 2, 5, 15, 3, \ldots$. You may assume a max : **nat gate** that at each step outputs the maximum of its input wires.

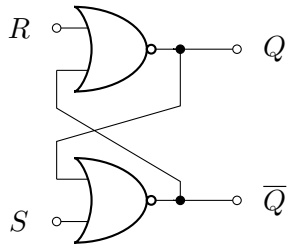
An interesting class of circuits is *flip-flop latches*. Flip-flop latches are circuits that can be used to store a single alterable bit of information. A simple example of a flip-flop latch is an SR latch. It has two input wires, a "set" wire S and a

"reset" wire R. Its output wires Q and $\overline{Q}$ respectively carry the current state and the complement of the current state. When the latch receives `1` on S and `0` on R, it sets Q to `1` (so $\overline{Q}$ becomes `0`). When the latch receives `1` on R and `0` on S, it resets Q to `0` (so $\overline{Q}$ becomes `1`). When both wires receive `0`, it maintains the state. Simultaneously sending `1` to both S and R is prohibited.

Key to the ability of a flip-flop latch to maintain state is feedback or recursion. For example, an SR latch can be implemented using feedback and two `NOR` gates. A `NOR` gate is a logic gate that produces the `NOR` of its inputs, that is, it outputs `1` if and only if both its inputs are `0`. We can implement the logical operation `NOR` as a function $\mathbf{bit} \otimes \mathbf{bit} \to \mathbf{bit}$ as follows:

$$\texttt{NOR} = \lambda p.\mathbf{case}\ p\ \{\langle b_0, b_1\rangle \Rightarrow \mathbf{case}\ l\ \{b_0 \cdot \_ \Rightarrow \mathbf{case}\ r\ \{b_0 \cdot \_ \Rightarrow \texttt{1} \mid b_1 \cdot \_ \Rightarrow \texttt{0}\} \mid b_1 \cdot \_ \Rightarrow \texttt{0}\}\}.$$

We can then implement a `NOR` gate *nor* : $\mathbf{bit\ gate}$ by $nor = \lambda S.map\ \texttt{NOR}\ (zip\ S)$. By feeding the output of each `NOR` to an input of the other gate, we can implement an SR latch as follows:



Task 6 (10 points, WB)**.** Give an implementation of SR latches as a function of type

$$(R : \mathbf{bit\ stream}) \otimes (S : \mathbf{bit\ stream}) \to \mathbf{bit\ stream}.$$

We only ask that you produce the output stream Q: the stream $\overline{Q}$ can easily be recovered from it. Assume that Q is initialized to `0`.

Mutual recursion

Some languages support defining mutually-recursive functions. For example, one could imagine writing the following ML program defining the mutually-recursive functions `even` and `odd`:

```
fun odd  n = case n of Z   => false
                     | S p => even p
and even n = case n of Z   => true
                     | S p => odd p
```

In this example function bodies have the type

$$\mathit{odd} : \mathbf{nat} \to \mathbf{bool}, \mathit{even} : \mathbf{nat} \to \mathbf{bool}, n : \mathbf{nat} \vdash \mathbf{case}\ n\ \{\cdots\} : \mathbf{bool}$$

and the functions are both closed expressions of type $\mathbf{nat} \to \mathbf{bool}$.

More generally, we may wish to define the mutually-recursive functions:

```
fun f1 x = e1
and f2 x = e2
```

where the bodies e_i have types

$$f_1 : \sigma_1 \to \tau_1, f_2 : \sigma_2 \to \tau_2, x : \sigma_1 \vdash e_1 : \tau_1$$
$$f_1 : \sigma_1 \to \tau_1, f_2 : \sigma_2 \to \tau_2, x : \sigma_2 \vdash e_2 : \tau_2$$

We saw in Lecture 8 how to define recursive functions using the construct $\mathbf{fix}(x.e)$. In this the task below, your task will be to implement mutual recursion for functions.

Task 7 (20 points, WB)**.** Let e_1 and e_2 be expressions satisfying

$$f_1 : \sigma_1 \to \tau_1, f_2 : \sigma_2 \to \tau_2, x : \sigma_1 \vdash e_1 : \tau_1$$
$$f_1 : \sigma_1 \to \tau_1, f_2 : \sigma_2 \to \tau_2, x : \sigma_2 \vdash e_2 : \tau_2$$

Define the mutually-recursive functions $f_1 : \sigma_1 \to \tau_1$ and $f_2 : \sigma_2 \to \tau_2$ by giving closed terms d_1 and d_2 satisfying $\cdot \vdash d_i : \sigma_i \to \tau_i$ and $d_i x = [d_1, d_2/f_1, f_2]e_i$ for $i = 1, 2$.

Hint. Remember, $\mathbf{fix}(x.e)$ is not limited to constructing terms of function type! It can also construct terms of type product type, of sum type, etc.

Hint. Many solutions are possible! To get closed terms, you will need to somehow bind the variables f_1, f_2, and x in e_1 and in e_2. One known solution simultaneously binds multiple variables; an other binds each of the variables separately.

Hint. Be careful if you try to combine eagerness and recursion. To manage eagerness, you may find useful the fact that $\lambda x.e$ is always a value.

A Syntax

Types τ and terms e are given by the following grammars, where I ranges over finite sets:

$$
\begin{aligned}
\tau &::= \alpha \mid \tau_1 \to \tau_2 \mid \textstyle\sum_{i \in I}(i : \tau_i) \mid \bigotimes_{i \in I}(i : \tau_i) \mid \mathbin{\&}_{i \in I}(i : \tau_i) \mid \rho(\alpha.\, \tau) \\
e &::= x \\
&\quad \mid \lambda x.e \mid e_1 e_2 \\
&\quad \mid i \cdot e \mid \textbf{case}\ e\ \{i \cdot x \Rightarrow e_i\}_{i \in I} \\
&\quad \mid \langle i \hookrightarrow e_i \rangle_{i \in I} \mid \textbf{case}\ e_1\ \{\langle i \hookrightarrow x_i \rangle_{i \in I} \Rightarrow e_2\} \\
&\quad \mid \langle\!| i \hookrightarrow e_i |\!\rangle_{i \in I} \mid e \cdot i \\
&\quad \mid \textbf{fold}(e) \mid \textbf{unfold}(e) \\
&\quad \mid \textbf{fix}(x.e)
\end{aligned}
$$

The above presentation of the syntax uses some abbreviations to help simplify presentation and for conciseness. For example, where $I = \{i_1, \ldots, i_n\}$, we abbreviate $\langle i_1 \hookrightarrow e_{i_1}, \ldots, i_n \hookrightarrow e_{i_n} \rangle$ by $\langle i \hookrightarrow e_i \rangle_{i \in I}$. Similarly, $\{i \cdot x \Rightarrow e_i\}_{i \in I}$ abbreviates $\{i_1 \cdot x_{i_1} \Rightarrow e_{i_1} \mid \cdots \mid i_n \cdot x_{i_n} \Rightarrow e_{i_n}\}$.

As discussed in class, the types $\mathbf{0}$ and $\mathbf{1}$ are special cases of sums and products with $I = \emptyset$.

B Statics

Whenever applicable ((I-+), (E-&), etc.), we have the obvious side-condition that $i \in I$. We assume for the sake of presentation that $I = \{i_1, \ldots, i_n\}$.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}\ (\textsc{Lam})$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}\ (\textsc{App})$$

$$\frac{\Gamma \vdash e : \tau_i}{\Gamma \vdash i \cdot e : \sum_{i \in I}(i : \tau_i)}\ (\text{I-+})$$

$$\frac{\Gamma \vdash e : \sum_{i \in I}(i : \tau_i) \quad \Gamma, x : \tau_{i_1} \vdash e_{i_1} : \tau \quad \cdots \quad \Gamma, x : \tau_{i_n} \vdash e_{i_n} : \tau}{\Gamma \vdash \textbf{case}\ e\ \{i \cdot x_i \Rightarrow e_i\}_{i \in I} : \tau}\ (\text{E-+})$$

$$\frac{\Gamma \vdash e_{i_1} : \tau_{i_1} \quad \cdots \quad \Gamma \vdash e_{i_n} : \tau_{i_n}}{\Gamma \vdash \langle i \hookrightarrow e_i \rangle_{i \in I} : \bigotimes_{i \in I}(i : \tau_i)}\ (\text{I-}\otimes)$$

$$\frac{\Gamma \vdash e_0 : \bigotimes_{i\in I}(i : \tau_i) \quad \Gamma, x_{i_1} : \tau_{i_1}, \ldots, x_{i_n} : \tau_{i_n} \vdash e_1 : \tau}{\Gamma \vdash \mathbf{case}\ e_0\ \{\langle i \hookrightarrow x_i\rangle_{i\in I} \Rightarrow e_1\} : \tau}\ (\text{E-}\otimes)$$

$$\frac{\Gamma \vdash e_{i_1} : \tau_{i_1} \quad \cdots \quad \Gamma \vdash e_{i_n} : \tau_{i_n}}{\Gamma \vdash \langle\!|i \hookrightarrow e_i|\!\rangle_{i\in I} : \&_{i\in I}(i : \tau_i)}\ (\text{I-}\&) \qquad \frac{\Gamma \vdash e : \&_{i\in I}(i : \tau_i)}{\Gamma \vdash e \cdot i : \tau_i}\ (\text{E-}\&)$$

$$\frac{\Gamma \vdash e : [\rho(\alpha.\,\tau)/\alpha]\tau}{\Gamma \vdash \mathbf{fold}(e) : \rho(\alpha.\,\tau)}\ (\textsc{Fold}) \qquad \frac{\Gamma \vdash e : \rho(\alpha.\,\tau)}{\Gamma \vdash \mathbf{unfold}(e) : [\rho(\alpha.\,\tau)/\alpha]\tau}\ (\textsc{Unfold})$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathbf{fix}(x.e) : \tau}\ (\textsc{Fix})$$

C Dynamics

We assume for the sake of presentation that $I = \{i_1, \ldots, i_n\}$.

$$\frac{}{\lambda x.e\ \mathit{val}}\ (\to\text{-}\textsc{Val}) \qquad \frac{}{(\lambda x.e_1)e_2 \mapsto [e_2/x]e_1}\ (\textsc{App-Red})$$

$$\frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2}\ (\textsc{App-Step-L}) \qquad \frac{e_1\ \mathit{val} \quad e_2 \mapsto e_2'}{e_1 e_2 \mapsto e_1 e_2'}\ (\textsc{App-Step-R})$$

$$\frac{e\ \mathit{val}}{i \cdot e\ \mathit{val}}\ (\textsc{val/inj}) \qquad \frac{e \mapsto e'}{i \cdot e \mapsto i \cdot e'}\ (\mapsto/\textsc{inj})$$

$$\frac{e \mapsto e'}{\mathbf{case}\ e\ \{i \cdot x_i \Rightarrow e_i\}_{i\in I} \mapsto \mathbf{case}\ e'\ \{i \cdot x_i \Rightarrow e_i\}_{i\in I}}\ (\mapsto/\mathbf{case}/\textsc{subj-}\textstyle\sum)$$

$$\frac{v_j\ \mathit{val}}{\mathbf{case}\ (j \cdot v_j)\ \{i \cdot x_i \Rightarrow e_i\}_{i\in I} \mapsto [v_j/x_j]e_j}\ (\mapsto/\mathbf{case}/\textsc{inj})$$

$$\frac{v_{i_1}\ \mathit{val} \quad \cdots \quad v_{i_n}\ \mathit{val}}{\langle i \hookrightarrow v_i\rangle_{i\in I}\ \mathit{val}}\ (\textsc{ETuple-val})$$

$$\frac{e_{i_1}\ \mathit{val} \quad \cdots \quad e_{i_{j-1}}\ \mathit{val} \quad e_{i_j} \mapsto e_{i_j}' \quad e_i' = e_i\ (\text{for } i \neq i_j)}{\langle i \hookrightarrow e_i\rangle_{i\in I} \mapsto \langle i \hookrightarrow e_i'\rangle_{i\in I}}\ (\textsc{ETuple-Step})$$

$$\frac{e_0 \mapsto e_0'}{\mathbf{case}\ e_0\ \{\langle i \hookrightarrow x_i\rangle_{i\in I} \Rightarrow e_1\} \mapsto \mathbf{case}\ e_0'\ \{\langle i \hookrightarrow x_i\rangle_{i\in I} \Rightarrow e_1\}}\ (\mapsto/\mathbf{case}/\textsc{subj-}\otimes)$$

$$\frac{\langle i \hookrightarrow v_i\rangle_{i\in I}\ \mathit{val}}{\mathbf{case}\ \langle i \hookrightarrow v_i\rangle_{i\in I}\ \{\langle i \hookrightarrow x_i\rangle_{i\in I} \Rightarrow e\} \mapsto [v_{i_1}, \ldots, v_{i_n}/x_{i_1}, \ldots, x_{i_n}]e}\ (\mapsto/\mathbf{case}/\otimes)$$

$$\frac{}{\langle\!|i \hookrightarrow e_i|\!\rangle_{i \in I}\ \mathit{val}} \qquad \frac{e \mapsto e'}{e \cdot i \mapsto e' \cdot i} \qquad \frac{}{\langle\!|i \hookrightarrow e_i|\!\rangle_{i \in I} \cdot i \mapsto e_i}$$

$$\frac{e\ \mathit{val}}{\mathbf{fold}(e)\ \mathit{val}} \qquad \frac{e \mapsto e'}{\mathbf{fold}(e) \mapsto \mathbf{fold}(e')}$$

$$\frac{e \mapsto e'}{\mathbf{unfold}(e) \mapsto \mathbf{unfold}(e')} \qquad \frac{\mathbf{fold}(e)\ \mathit{val}}{\mathbf{unfold}(\mathbf{fold}(e)) \mapsto e}$$

$$\frac{}{\mathbf{fix}(x.e) \mapsto [\mathbf{fix}(x.e)/x]e}\ (\textsc{Fix-Step})$$