The safe λ-calculus is a fragment of the simply typed λ-calculus introduced in [BO09], motivated by the theory of higher-order recursion schemes. It has some interesting properties, e.g. there is a normalization procedure that never needs to perform any α-renaming (i.e. uses capture-permitting substitution).

Determining the precise complexity of deciding whether two safe λ-terms are $\beta\eta$-convertible has been an open problem until now (to my knowledge). In [BO09, §3], a PSpace-hardness result is established. This is far below the non-elementary lower bound for simply typed $\beta\eta$-convertibility due to Statman, but it is argued that both Statman's proof and its simplification by Mairson fundamentally require unsafe terms. To quote [BO09], this "does not rule out the possibility that another non-elementary problem is encodable in the safe lambda calculus".

We provide such an encoding here: by reduction from the star-free expression equivalence problem, we show that safe $\beta\eta$-convertibility is Tower-complete in the sense of [Sch16]. (We only provide a proof of Tower-hardness, membership holds because it works for STLC already. [TODO: actually this proves that $\beta\eta$ in STLC is Tower-c which is not explicitly written anywhere in the published literature])

First, let us recall the definitions of our objects of interest:

Definition 1 (slight simplification of [BO09])**.** As usual, the *order* of a simple type is the nesting depth of function arrows to the left:

$$\mathrm{ord}(o) = 0 \qquad \mathrm{ord}(A \to B) = \max(\mathrm{ord}(A) + 1, \mathrm{ord}(B))$$

The typing rules of the safe λ-calculus are

$$\frac{}{x : A \vdash x : A} \qquad \frac{\Theta \vdash t : A}{\Theta' \vdash t : A}\ (\Theta \subset \Theta')$$

$$\frac{\Theta \vdash t : A_1 \to \ldots \to A_n \to B \quad \Theta \vdash u_1 : A_1 \ \ldots\ \Theta \vdash u_n : A_n}{\Theta \vdash t\, u_1 \ \ldots\ u_n : B} \text{ when } \mathrm{ord}(B) \leq \inf_{(y:C)\in\Theta} \mathrm{ord}(C)$$

$$\frac{\Theta,\, x_1 : A_1,\, \ldots,\, x_n : A_n \vdash t : B}{\Theta \vdash \lambda x_1. \ldots \lambda x_n. t : A_1 \to \cdots \to A_n \to B} \text{ when } \mathrm{ord}(A_1 \to \cdots \to A_n \to B) \leq \inf_{(y:C)\in\Theta} \mathrm{ord}(C)$$

with the usual convention $\inf(\varnothing) = +\infty$.

Definition 2. *Star-free expressions* are regular expressions without the Kleene star, but with complementation:

$$E, F ::= \varnothing \mid \varepsilon \mid c \mid E \cup F \mid E \cdot F \mid E^c$$

(for c in an alphabet Σ). The *star-free equivalence problem* consists in deciding whether two star-free expressions denote the same language (subset of Σ^*).

Definition 3 (classical)**.** The types $\mathtt{Str}_\Sigma = (o \to o) \to \cdots \to (o \to o) \to o \to o$ (with $|\Sigma|$ occurrences of $(o \to o)$) and $\mathtt{Bool} = o \to o \to o$ are the so-called *Church encodings* of strings and booleans. In the case of a unary alphabet, we set $\mathtt{Nat} = \mathtt{Str}_{\{c\}} = (o \to o) \to o \to o$: it is the type of Church *numerals*.

The Church encoding of a word $w = w_1 \dots w_n$ over an ordered alphabet $\Sigma = \{c_1, \dots, c_n\}$ is $\overline{w} = \lambda f_{c_1}.\ \dots\ \lambda f_{c_n}.\ \lambda x.\ f_{w[1]}\ (\dots\ (f_{w[n]}\ x)) : \mathtt{Str}_\Sigma$. The Church encoding of $n \in \mathbb{N}$ is $\overline{n} = \overline{c \dots c} : \mathtt{Nat}$ with n times c.

Note that $\mathtt{Bool}$ only has two closed β-normal inhabitants, $\mathtt{true} = \lambda xy.\ x$ and $\mathtt{false} = \lambda xy.\ y$. We will want to prove the following:

Theorem 4. *Given a safe λ-term t, it is* Tower*-hard to decide whether $t =_{\beta\eta} \mathtt{true}$.*

This will proceed by reduction from star-free equivalence, which was given in [Sch16, §3.1] as an example of a Tower-complete problem. Any elementary complexity reduction will do, in fact we will even have a polynomial time reduction (or exponential time if we insist on writing out the full type derivation for the term we build).

First, note that star-free equivalence reduces to star-free *emptiness*: given two expressions E and F, the language denoted by $(E^c \cup F)^c \cup (E \cup F^c)^c$ is empty if and only if E and F are equivalent. This new problem can be solved by a kind of bounded search:

Lemma 5 (classical)**.** *Suppose that the expression E of size n denotes a non-empty language. This language then contains a word of length at most* $\mathrm{tower}(n)$*.*

(Recall that $\mathrm{tower}(n+1) = 2^{\mathrm{tower}(n)}$.)

Proof sketch. We translate E to an equivalent nondeterministic finite automaton (NFA), whose number of states bounds the length of a shortest word (indeed such a word has an accepting run that visits each state at most once). This can be done by induction of E, using any standard construction on NFA for union and concatenation; the costliest operation is complementation, for which we use determinization, inducing a single exponential state blowup. ☐

Remark 6. If we wanted to inductively build *deterministic* finite automata, then the exponential blowup would occur on the treatment of concatenation. In fact it is a well-understood phenomenon that the source of complexity lies in the *alternations* of concatenation and complementation (cf. dot-depth / Straubing-Thérien hierarchy).

Thus, we only have to test E against all inputs of length at most $\mathrm{tower}(n)$. We do so using the following lemmas (notation: $A[B] = A[B/o]$; note that any $t : A$ can be typecast to a term of type $A[B]$):

Lemma 7 (see [BO09, Remark 2.5(iii)])**.** *One can build a safe λ-term that is $\beta\eta$-convertible to* $\overline{\mathrm{tower}(n)} : \mathtt{Nat}$ *in polynomial time.*

Lemma 8. *For any finite alphabet Σ and $\# \notin \Sigma$, there exists a safe λ-term* $\mathtt{enum} : \mathtt{Nat}[A_{\mathtt{enum}}] \to \mathtt{Str}_{\Sigma \cup \{\#\}}$ *such that for any $m \in \mathbb{N}$,* $\mathtt{enum}\ \overline{m}$ *reduces to the encoding of a #-separated list containing all words in Σ^* of length at most m.*

Lemma 9. *There exists a simply typed λ-term* $\texttt{exists} : (\texttt{Str}_\Sigma[\alpha] \to \texttt{Bool}) \to \texttt{Str}_{\Sigma\cup\{\#\}}[F_{\texttt{exists}}(\alpha)] \to \texttt{Bool}$*, where α is a type variable, whose instantations $\alpha = A$ are all safe and test whether some word (over Σ) in a given list of words (encoded using the separator $\#$) satisfies a given predicate.*

Lemma 10. *E can be turned in polynomial time into an equivalent safe λ-term* $t_E : \texttt{Str}_\Sigma[A_E] \to \texttt{Bool}$.

We conclude by composing the four terms built in the above lemmas and $\texttt{not} : \texttt{Bool} \to \texttt{Bool}$; this is allowed because safe terms of type $A[T] \to B$ and $B[U] \to C$ can be composed into a safe term of type $A[T[U]] \to C$ whenever $\mathrm{ord}(B) \leq \mathrm{ord}(A[T])$.

Proof of Lemma 8. Let $\Sigma = \{c_1, \dots, c_{|\Sigma|}\}$. Take $A_{\texttt{enum}} = \texttt{Str}_\Sigma \to \texttt{Str}_\Sigma$ and

$$\texttt{enum} = \lambda x.\ x\ (\lambda f.\ \lambda s.\ \texttt{conc}_{|\Sigma|+1}\ (f\ \overline{\#})\ (f\ (\texttt{conc}\ \overline{c_1}\ s))\ \dots\ (f\ (\texttt{conc}\ \overline{c_{|\Sigma|}}\ s)))\ (\lambda y.\ y)\ \overline{\#}$$

where $\texttt{conc}_k : \texttt{Str} \to \dots \to \texttt{Str} \to \texttt{Str}$ is the concatenation of k strings, which is safely λ-definable according to [BO09, Theorem 2.8], and $\texttt{conc} = \texttt{conc}_2$. ☐

Proof of Lemma 9. Let $\Sigma = \{c_1, \dots, c_{|\Sigma|}\}$. Take $F_{\texttt{exists}}(\alpha) = \texttt{Str}_\Sigma[\alpha] \to \texttt{Bool}$ and $\texttt{exists} = \lambda p.\ \lambda s.\ s\ u_1\ \dots\ u_{|\Sigma|}\ v\ p\ \overline{\varepsilon}$ where

$$u_i = \lambda f.\ \lambda x.\ f\ (\texttt{conc}\ x\ \overline{c_i}) \qquad u_\# = \lambda f.\ \lambda x.\ \texttt{or}\ (p\ x)\ (f\ \overline{\varepsilon})$$

where $\texttt{or} : \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}$ can be safely defined following the recipe of [BO09, Remark 2.5(ii)]. (Explanation of this code: the accumulator of the "right fold" contains a function that takes the maximal $\#$-free factor strictly before the current position, and tells you whether all blocks up to a certain point satisfy the predicate p.) Note that all instantiations $\alpha = A$ of this term are safe. ☐

Proof of Lemma 10. By induction on the expression.

- We take $A_\varnothing = o$ and $t_\varnothing = \lambda s.\ \texttt{false} : \texttt{Str}_\Sigma \to \texttt{Bool}$.
- In the unsafe λ-calculus, testing for the empty word could be done with type $\texttt{Str}_\Sigma \to \texttt{Bool}$, but here this is not possible anymore as discussed in [BO09, §2]. We use instead $A_\varepsilon = \texttt{Bool}$ and

$$t_\varepsilon = \lambda s.\ s\ (\lambda x.\ \texttt{false})\ \dots\ (\lambda x.\ \texttt{false})\ \texttt{true}$$

- To test whether the word contains a single letter, say, the first one in the alphabet Σ (call it c_1), we use $A_{c_1} = \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}$ and

$$t_{c_1} = \lambda s.\ s\ (\lambda fxy.\ f\ \texttt{true}\ (\texttt{and}\ (\texttt{not}\ x)\ y))\ t'\ \dots\ t'\ \texttt{and}\ \texttt{false}\ \texttt{true}$$

  where $t' = \lambda fxy.\ f\ x\ \texttt{false}$, and $\texttt{and}$ is defined from $\texttt{or}$ and $\texttt{not}$.
- Complementation is implemented by post-composing with $\texttt{not}$.

- To handle a union, we take $A_{E\cup F} = \mathtt{Str}[A_E] \to \mathtt{Str}[A_F] \to \mathtt{Bool}$ and

  $$t_{E\cup F} = \lambda s.\ s\ (\lambda fxy.\ f\ (\mathtt{conc}\ \overline{c_1}\ x)\ (\mathtt{conc}\ \overline{c_1}\ y))\ \ldots\ (\lambda xy.\ \mathtt{or}\ (t_E\ x)\ (t_F\ y))\ \overline{\varepsilon}\ \overline{\varepsilon}$$

  Observe that t_E and t_F appear only once; otherwise, our construction would not run in polynomial time.

- The remaining case, concatenation, is the most delicate:

  - First, we introduce a new symbol $\square \notin \Sigma$ and build a term of type $\mathtt{Str}_{\Sigma\cup\{\square\}}[\mathtt{Str}_\Sigma[A_E] \to (\mathtt{Str}_\Sigma[A_F] \to \mathtt{Bool}) \to \mathtt{Bool}] \to \mathtt{Bool}$ that distinguishes, among the words of $\Sigma^*\square\Sigma^*$, those that belong to $E\square F$ (we do not care what it computes on the rest of $(\Sigma \cup \{\square\})^*$):

    $$t_{E\square F} = \lambda s.\ s\ v_{c_1}\ \ldots\ v_{c_{|\Sigma|}}\ v_\square\ (\lambda xf.\ f\ \overline{\varepsilon})\ \overline{\varepsilon}\ (\lambda y.\ \mathtt{false})$$

    where

    $$v_c = \lambda kxf.\ k\ (\mathtt{conc}\ x\ \overline{c})\ (\lambda y.\ f\ (\mathtt{conc}\ \overline{c}\ y))$$
    $$v_\square = \lambda kxf.\ k\ \overline{\varepsilon}\ \underbrace{(\lambda y.\ \mathtt{and}\ (t_E\ x)\ (t_F\ y))}$$

    Note that the underbraced subterm has type $\mathtt{Str}_\Sigma[A_F] \to \mathtt{Bool}$ and contains a free variable of type $\mathtt{Str}_\Sigma[A_E]$. The safety condition tells us that we need to have $\mathrm{ord}(A_E) \geq \mathrm{ord}(A_F)+1$. We can always make sure that we are in such a situation: by composing t_E m times with a safe λ-term $\mathtt{copy} : \mathtt{Str}_\Sigma[\mathtt{Str}_\Sigma] \to \mathtt{Str}_\Sigma$ which realizes the identity function on Σ^* (its existence is left to the reader…) we can get a term $t_E^{(m)} : \mathtt{Str}_\Sigma[A_E^{(m)}] \to \mathtt{Bool}$ with $\mathrm{ord}(A_E^{(m)}) = \mathrm{ord}(A_E)+2m$ that recognizes the same language.

  - Then, we show that $\mathtt{123} \mapsto \square\mathtt{123}\#\mathtt{1}\square\mathtt{23}\#\mathtt{12}\square\mathtt{3}\#\mathtt{123}\square : \Sigma^* \to \Gamma^*$ where $\Gamma = \Sigma \cup \{\square, \#\}$ (a typical *polyregular function*) is defined by $\mathtt{split} : \mathtt{Str}_\Sigma[\mathtt{Str}_\Gamma[\mathtt{Str}_\Gamma] \to (\mathtt{Str}_\Gamma \to \mathtt{Str}_\Gamma) \to \mathtt{Str}_\Gamma] \to \mathtt{Str}_\Gamma$:

    $$\mathtt{split} = \lambda s.\ s\ v'_{c_1}\ \ldots\ v'_{c_{|\Sigma|}}\ (\lambda xf.\ \mathtt{conc}_3\ (f\ \overline{\#})\ (\mathtt{copy}\ x)\ \overline{\square})\ \overline{\varepsilon}\ (\lambda y.\ \overline{\varepsilon})$$

    $$v'_c = \lambda kxf.\ k\ (\mathtt{conc}\ x\ \overline{c})\ (\lambda y.\ \mathtt{conc}_4\ (f\ (\mathtt{conc}\ \overline{c}\ y))\ (\mathtt{copy}\ x)\ \overline{\square c}\ y)$$

    Thanks to our use of $\mathtt{Str}_\Gamma[\mathtt{Str}_\Gamma]$ and $\mathtt{copy}$, this λ-term is safe.

  - Finally, the term that we want is $\lambda s.\ \mathtt{exists}\ t_{E\square F}\ (\mathtt{split}\ s)$ (here we use $\mathtt{exists}$ from Lemma 9). $\square$

References

[BO09] William Blum and C.-H. Luke Ong. The Safe Lambda Calculus. *Logical Methods in Computer Science*, 5(1), February 2009.

[Sch16] Sylvain Schmitz. Complexity hierarchies beyond elementary. *ACM Transactions on Computation Theory*, 8(1):3:1–3:36, 2016.