Lifting Resugaring by Lazy Desugaring

ANONYMOUS AUTHOR(S)

A APPENDIX

A.1 Extension for black-box core evaluator

Discussion in this subsection is based on Section 3 of the main text. One may notice the traditional resugaring approach does not need the whole evaluation rules of core language, a black-box stepper is enough instead. Our approach can also work by given a black-box stepper with a tricky extension, but a little more information is needed. Here we introduce the extension firstly. We use $\rightarrow_c$ to denote a reduction step of core language's expression in the black-box stepper, and $\rightarrow_e$ to denote a step in the extension evaluator for the mixed language. We may use $\rightarrow_m$ to denote the one-step reduction in our mixed language, defined in Section 3.2.

$$\frac{\begin{array}{c}\forall\ i.\ e_i \in \texttt{CoreExp} \\ (\texttt{CoreHead}\ e_1\ \ldots\ e_n)\ \rightarrow_c\ e'\end{array}}{(\texttt{CoreHead}\ e_1\ \ldots\ e_n)\ \rightarrow_e\ e'} \quad (\textsc{CoreRed})$$

$$\frac{\begin{array}{c}\forall\ i.\ tmp_i = (e_i \in \texttt{SurfExp}\ ?\ \texttt{tmpe}\ :\ e_i),\ \textit{where}\ \texttt{tmpe}\ \textit{is any reduciable}\ \texttt{CoreExp}\ \textit{term} \\ (\texttt{CoreHead}\ tmp_1\ \ldots\ tmp_i\ \ldots\ tmp_n)\ \rightarrow_c\ (\texttt{CoreHead}\ tmp_1\ \ldots\ tmp_i'\ \ldots\ tmp_n)\end{array}}{\begin{array}{c}(\texttt{CoreHead}\ e_1\ \ldots\ e_i\ \ldots\ e_n)\ \rightarrow_e\ (\texttt{CoreHead}\ e_1\ \ldots\ e_i'\ \ldots\ e_n) \\ \textit{where}\ e_i\ \rightarrow_m\ e_i'\ \textit{if}\ e_i\ \in\ \texttt{SurfExp},\ \textit{else}\ e_i\ \rightarrow_e\ e_i'\end{array}} \quad (\textsc{CoreExt1})$$

$$\frac{\begin{array}{c}\forall\ i.\ tmp_i = (e_i \in \texttt{SurfExp}\ ?\ \texttt{tmpe}\ :\ e_i),\ \textit{where}\ \texttt{tmpe}\ \textit{is any reduciable}\ \texttt{CoreExp}\ \textit{term} \\ (\texttt{CoreHead}\ tmp_1\ \ldots\ tmp_n)\ \rightarrow_c\ e'\ \text{// not reduced in subexpressions}\end{array}}{(\texttt{CoreHead}\ e_1\ \ldots\ e_n)\ \rightarrow_e\ e'[e_1/tmp_1\ \ldots\ e_n/tmp_n]} \quad (\textsc{CoreExt2})$$

Then we should replace the rules `CoreRed1` and `CoreRed2` by the following rule.

$$\frac{(\texttt{CoreHead}\ e_1\ \ldots\ e_n)\ \rightarrow_e\ e'}{(\texttt{CoreHead}\ e_1\ \ldots\ e_n)\ \rightarrow_m\ e'} \quad (\textsc{ExtRed})$$

Putting them in simple words. For expression $(\texttt{CoreHead}\ e_1\ \ldots\ e_n)$ whose subexpressions contain `SurfExp`, replacing all `SurfExp` subexpressions not in core language with any reducible core language's term `tmpe`. Then getting a result after inputting the new expression e' to the original black-box stepper. If reduction appears at a subexpression at e_i or what the e_i replaced by, then the stepper with the extension should return $(\texttt{CoreHead}\ e_1\ \ldots\ e_i'\ \ldots\ e_n)$, where e_i' is e_i after the mixed language's one-step reduction ($\rightarrow_m$) or after core language's reduction with extension ($\rightarrow_e$) (rule `CoreExt1`, an example in Figure 1). Otherwise, the stepper should return e', with all the replaced subexpressions replacing back (rule `CoreExt2`, an example in Figure 2). The extension will not violate the properties of original core language's evaluator. It is obvious that the evaluator with the extension will reduce at the subexpression as it needs in core language, if the reduction appears in a subexpression. The stepper with extension behaves the same as mixing the evaluation

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

rules of core language and desugaring rules of surface language. The extension is just to make it works when the evaluator of core language is a black-box stepper, by getting context rules using the `tmpe`. That's why the extension is tricky.

```
(if (and e1 e2) true false)
        ⇓replace
(if tmpe1 true false)
        ⇓blackbox
(if tmpe1' true false)
        ⇓desugar
(if (if e1 e2 false) true false)
```

Fig. 1. `CoreExt1`'s Example

```
(if (if true ture false) (and ...) (or ...))
        ⇓replace
(if (if true ture false) tmpe2 tmpe3)
        ⇓blackbox
(if true tmpe2 tmpe3)
        ⇓replaceback
(if true (and ...) (or ...))
```

Fig. 2. `CoreExt2`'s Example

But something goes wrong when substitution takes place during `CoreExt2`. For a expression like `(let x 2 (Sugar x y))` as an example, it should reduce to `(Sugar 2 y)` by the `CoreRed2` rule, but got `(Sugar x y)` by the `CoreExt2` rule. So when using the extension of black-box stepper's rule (`ExtRed2`), we need some other information about in which subexpression a substitution will occur (the substitution can be got by a similar idea as the tricky extension). Then for these subexpressions, we need to do the same substitution before replacing back. For example, we know the third subexpression of a expression headed with `let` is to be substituted. we should first try `(let x 2 x)`, `(let x 2 y)` in one-step reduction to get the substitution {2/x}, then, getting `(Sugar 2 y)`.

A.2 Implementing hygiene for non-hygienic rewriting

Discussion in this subsection is to describe how to modify reduction rules of mixed language to achieve a hygienic `Let` sugar based on the non-hygienic `let` in core language, based on Section 5.1.2.

```
      (Let x 1 (+ x (Let x 2 (+ x 1))))

-->     { a one-step try on the outermost Let }
      (try (let (x 1) (+ x (Let x 2 (+ x 1)))))
```

Here during the one-step try, we should get `(+ 1 (Let 1 2 (+ 1 1)))` by after 2 reductions in the mixed language, which is an ill-formed expression. But we can reject the one-step try's result if getting a expression out of the syntax of mixed language, and then apply $\rightarrow_m$ on the subexpression where the ill-form exists recursively. It is a different "lazy desugaring", by delaying expansion of sugar if it causes an error.