FORMALIZATION

We consider a variant of λ-calculus with let-bindings, products, mutable references, and injection-case. The language also contains delimited control operators shift and reset.

CORE LANGUAGE		
Expressions:		EXP
e	::=	$c \mid x \mid e\ +\ e \mid e\ *\ e \mid \lambda x.\ e \mid @\ e\ e \mid \texttt{let}\ x\ =\ e\ \texttt{in}\ e$
	$\mid$	$\texttt{fst}\ e \mid \texttt{snd}\ e \mid (e,\ e) \mid \texttt{ref}\ e \mid !\ e \mid e := e$
	$\mid$	$\texttt{inl}\ e \mid \texttt{inr}\ e \mid \texttt{case}\ e\ \texttt{of}\ x \Rightarrow e\ \texttt{or}\ x \Rightarrow e$
	$\mid$	$\texttt{shift}\ x\ \texttt{in}\ e \mid \langle\ e\ \rangle$
Values:		VAL
		structure left abstract
Environments:		$\texttt{ENV} = \texttt{ID} \rightharpoonup \texttt{VAL}$
$\emptyset$	::=	$\lambda x.\bot$ where $\bot$ is a non-terminating computation
$\rho[x \mapsto v]$	::=	$\lambda y.\ \texttt{if}\ (x = y)\ \texttt{then}\ v\ \texttt{else}\ @\ \rho\ y$

DERIVED CONSTRUCTS		
Booleans and conditionals:		
Value: True	=	$\texttt{inl}\ ()$
Value: False	=	$\texttt{inr}\ ()$
$\texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ e$	=	$\texttt{case}\ b\ \texttt{of}\ y \Rightarrow t\ \texttt{or}\ z \Rightarrow e$
Loops and recursion:		
$\texttt{letrec}\ f = \lambda x.\ e_1\ \texttt{in}\ e_2$	=	$\texttt{let}\ f' = \lambda f'.\lambda x.\ \texttt{let}\ f = @\ f'\ f'\ \texttt{in}\ e_1\ \texttt{in}$
		$\texttt{let}\ f = @\ f'\ f'\ \texttt{in}\ e_2$
Loops:		expressed as tail recursive functions
Tree data structures:		
Example tree term: t	=	$\texttt{inr}\ (\texttt{inl}\ 5, \texttt{inl}\ 6)$
Syntactic sugar:		
$y_1\ \mathrel{+}= !\ y_2$	=	$y_1\ := !\ y_1 + !\ y_2$
$\texttt{let}\ (y, y') = e_1\ \texttt{in}\ e_2$	=	$\texttt{let}\ \tilde{y} = e_1\ \texttt{in}\ \texttt{let}\ y = \texttt{fst}\ \tilde{y}\ \texttt{in}\ \texttt{let}\ y' = \texttt{snd}\ \tilde{y}\ \texttt{in}\ e_2$
$e_1\ ;\ e_2$	=	$\texttt{let}\ \_ = e_1\ \texttt{in}\ e_2$

Above we show formal definitions of the language we consider. It serves as both object- and meta-language. We show the syntax of the core languages (typeless, but types can be added), as well as derived constructs that express branches, loops, recursion, and recursive data structures in a standard way. Syntactic sugar used in our presentation is also listed here.

We assume Barendregt's variable convention throughout, such that all bound variables are pairwise different and different from the free variables. This allows several rules (labeled via # in later figures) to be simplified compared to other formulations (no need for variable substitution in transformation).

For transformation, we assume that the target language is the same as object language, unless noted otherwise.

1 STANDARD INTERPRETATION AND TRANSFORMATION

1.1 Standard Metacircular Transformation

$$
\begin{array}{rcl}
[\![.]\!] : & & \mathsf{EXP} \longrightarrow \mathsf{EXP} \\
[\![c]\!] & = & c \\
[\![x]\!] & = & x \\
[\![e_1 + e_2]\!] & = & [\![e_1]\!] + [\![e_2]\!] \\
[\![e_1 * e_2]\!] & = & [\![e_1]\!] * [\![e_2]\!] \\
[\![\lambda y.\, e]\!] & = & \lambda y.\, [\![e]\!] \\
[\![@\ e_1\ e_2]\!] & = & @\ [\![e_1]\!]\ [\![e_2]\!] \\
[\![\mathtt{let}\ y = e_1\ \mathtt{in}\ e_2]\!] & = & \mathtt{let}\ y = [\![e_1]\!]\ \mathtt{in}\ [\![e_2]\!] \\
[\![\mathtt{fst}\ e]\!] & = & \mathtt{fst}\ [\![e]\!] \\
[\![\mathtt{snd}\ e]\!] & = & \mathtt{snd}\ [\![e]\!] \\
[\![\mathtt{ref}\ e]\!] & = & \mathtt{ref}\ [\![e]\!] \\
[\![!\ e]\!] & = & !\ [\![e]\!] \\
[\![e_1 := e_2]\!] & = & [\![e_1]\!] := [\![e_2]\!] \\
[\![(e_1, e_2)]\!] & = & ([\![e_1]\!], [\![e_2]\!]) \\
[\![\mathtt{inl}\ e]\!] & = & \mathtt{inl}\ [\![e]\!] \\
[\![\mathtt{inr}\ e]\!] & = & \mathtt{inr}\ [\![e]\!] \\
[\![\mathtt{case}\ e\ \mathtt{of}\ y_1 \Rightarrow e_1\ \mathtt{or}\ y_2 \Rightarrow e_2]\!] & = & \mathtt{case}\ [\![e]\!]\ \mathtt{of}\ y_1 \Rightarrow [\![e_1]\!]\ \mathtt{or}\ y_2 \Rightarrow [\![e_2]\!] \\
[\![\mathtt{shift}\ k\ \mathtt{in}\ e]\!] & = & \mathtt{shift}\ k\ \mathtt{in}\ [\![e]\!] \\
[\![\langle\ e\ \rangle]\!] & = & \langle\ [\![e]\!]\ \rangle
\end{array}
$$

Here we show the standard metacircular transformation using shift/reset in the target language. But what if we want to use a target language that does not provide shift/reset operators? This can be achieved by moving the uses of shift/reset into the meta-language (so that they are used at the time of translation), and generating target terms in explicit CPS (without shift/reset).

1.2 Standard CPS Transformation using shift/reset

$$
\begin{array}{rcl}
\llbracket . \rrbracket : & & \mathsf{EXP} \longrightarrow \mathsf{EXP} \\
\llbracket c \rrbracket & = & c \\
\llbracket x \rrbracket & = & x \\
\llbracket e_1 + e_2 \rrbracket & = & \llbracket e_1 \rrbracket \ \underline{+}\ \llbracket e_2 \rrbracket \\
\llbracket e_1 * e_2 \rrbracket & = & \llbracket e_1 \rrbracket \ \underline{*}\ \llbracket e_2 \rrbracket \\
\llbracket \lambda y.\ e \rrbracket & = & \underline{\lambda} y.\underline{\lambda} k.\ \overline{\langle}\ \utilde{@}\ \ k\ \llbracket e \rrbracket\ \overline{\rangle} \\
\llbracket\ @\ e_1\ e_2 \rrbracket & = & \overline{\texttt{shift}}\ k\ \overline{\texttt{in}}\ \underline{@}\ (\underline{@}\ \llbracket e_1 \rrbracket\llbracket e_2 \rrbracket)(\utilde{\lambda}\ a.\overline{@}\ k\ a) \\
\#\quad \llbracket \texttt{let}\ y = e_1\ \texttt{in}\ e_2 \rrbracket & = & \overline{\texttt{shift}}\ k\ \overline{\texttt{in}}\ \underline{\texttt{let}}\ y = \llbracket e_1 \rrbracket\ \underline{\texttt{in}}\ \overline{\langle}\ \overline{@}\ k\ \llbracket e_2 \rrbracket\ \overline{\rangle} \\
\llbracket \texttt{fst}\ e \rrbracket & = & \underline{\texttt{fst}}\ \llbracket e \rrbracket \\
\llbracket \texttt{snd}\ e \rrbracket & = & \underline{\texttt{snd}}\ \llbracket e \rrbracket \\
\llbracket \texttt{ref}\ e \rrbracket & = & \underline{\texttt{ref}}\ \llbracket e \rrbracket \\
\llbracket !\ e \rrbracket & = & \underline{!}\ \llbracket e \rrbracket \\
\llbracket e_1 := e_2 \rrbracket & = & \llbracket e_1 \rrbracket\ \underline{:=}\ \llbracket e_2 \rrbracket \\
\llbracket (e_1, e_2) \rrbracket & = & \underline{(}\llbracket e_1 \rrbracket\ \underline{,}\ \llbracket e_2 \rrbracket\underline{)} \\
\llbracket \texttt{inl}\ e \rrbracket & = & \underline{\texttt{inl}}\ \llbracket e \rrbracket \\
\llbracket \texttt{inr}\ e \rrbracket & = & \underline{\texttt{inr}}\ \llbracket e \rrbracket \\
\llbracket \texttt{case}\ e\ \texttt{of}\ y_1 \Rightarrow e_1\ \texttt{or}\ y_2 \Rightarrow e_2 \rrbracket & = & \overline{\texttt{shift}}\ k\ \overline{\texttt{in}}\ \utilde{\texttt{let}}\ k_1 = \utilde{\lambda}\ a.\ \overline{@}\ k\ a\ \utilde{\texttt{in}} \\
 & & \underline{\texttt{case}}\ \llbracket e \rrbracket\ \underline{\texttt{of}}\ y_1 \Rightarrow \overline{\langle}\ \utilde{@}\ \ k_1\ \llbracket e_1 \rrbracket\ \overline{\rangle}\ \underline{\texttt{or}}\ y_2 \Rightarrow \overline{\langle}\ \utilde{@}\ \ k_1\ \llbracket e_2 \rrbracket\ \overline{\rangle} \\
\#\quad \llbracket \texttt{shift}\ k\ \texttt{in}\ e \rrbracket & = & \overline{\texttt{shift}}\ k_1\ \overline{\texttt{in}}\ \underline{\texttt{let}}\ k = \underline{\lambda} v.\ \underline{\lambda} k_2.\ \underline{@}\ k_2(\overline{@}\ k_1\ v)\ \underline{\texttt{in}}\ \overline{\langle}\ \llbracket e \rrbracket\ \overline{\rangle} \\
\llbracket \langle\ e\ \rangle \rrbracket & = & \overline{\langle}\ \llbracket e \rrbracket\ \overline{\rangle}
\end{array}
$$

Here we show the standard CPS transformation using shift/reset. The rules are adapted from **?**, and the # symbol denotes rules that are simplified due to Barendregt's variable convention. We also adopted the overline/underline notation from **?**, such that overline denotes static/meta-language constructs, and underline denotes dynamic/target-language constructs. Departing slightly from **?**, we introduce another **wavy underline notation** to implement proper tail calls. Wavy underline denotes target-language terms just as normal underline, but wavy terms will be normalized with respect to the following contraction rules while the target expression is built up:

$$
\begin{array}{rcl}
\utilde{\lambda}\ y.\ \utilde{@}\ \ e\ y & \rightarrow & e \\
\utilde{\texttt{let}}\ y = y_1\ \utilde{\texttt{in}}\ e & \rightarrow & e[y \leftarrow y_1]
\end{array}
$$

Note that the wavy underline notation for let means that let-bindings should be removed if and only if the RHS of the let-binding is just a variable (symbol). This rule not strictly necessary for properly tail-recursive calls, but it removes unnecessary symbol bindings for case expression in abstraction.

Note that we don't need to model "code types", because we assume that object-level ASTs are modeled as proper tree types, using recursive types, products, and sums.

1.3 Standard CPS Transformation

It is of course also possible to express the CPS transformation without shift/reset entirely by switching the meta-languge code to CPS. This can be achieved formally by applying the same transformation as above to the meta-language translation code. The result is that shift/reset are fully erased from the right-hand sides of the translation.

$$
\begin{array}{rcl}
\llbracket . \rrbracket : & & \mathsf{EXP} \longrightarrow \mathsf{EXP} \\
\llbracket c \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \kappa\ c \\
\llbracket x \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \kappa\ x \\
\llbracket e_1 + e_2 \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \llbracket e_1 \rrbracket(\overline{\lambda} y_1.\ \overline{@}\ \llbracket e_2 \rrbracket(\overline{\lambda} y_2.\ \overline{@}\ \kappa(y_1 \underline{+} y_2))) \\
\llbracket e_1 * e_2 \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \llbracket e_1 \rrbracket(\overline{\lambda} y_1.\ \overline{@}\ \llbracket e_2 \rrbracket(\overline{\lambda} y_2.\ \overline{@}\ \kappa(y_1 \underline{*} y_2))) \\
\llbracket \lambda y.\ e \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \kappa\ (\underline{\lambda} y.\ \underline{\lambda} k.\ \overline{@}\ \llbracket e \rrbracket\ (\overline{\lambda}\ m. \utilde{@}\ \ k\ m)) \\
\llbracket\ @\ e_1\ e_2 \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \llbracket e_1 \rrbracket(\overline{\lambda} m.\ \overline{@}\ \llbracket e_2 \rrbracket(\overline{\lambda} n.\ \underline{@}\ (\ \underline{@}\ m\ n)\ (\utilde{\lambda}\ a.\ \overline{@}\ \kappa\ a))) \\
\#\quad \llbracket \mathtt{let}\ y = e_1\ \mathtt{in}\ e_2 \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \llbracket e_1 \rrbracket(\overline{\lambda} y_1.\ \underline{\mathtt{let}}\ y = y_1\ \underline{\mathtt{in}}\ \overline{@}\ \llbracket e_2 \rrbracket\ \kappa) \\
\llbracket \mathtt{fst}\ e \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \llbracket e \rrbracket(\overline{\lambda} y.\ \overline{@}\ \kappa\ (\underline{\mathtt{fst}}\ y)) \\
\llbracket \mathtt{snd}\ e \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \llbracket e \rrbracket(\overline{\lambda} y.\ \overline{@}\ \kappa\ (\underline{\mathtt{snd}}\ y)) \\
\llbracket \mathtt{ref}\ e \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \llbracket e \rrbracket(\overline{\lambda} y.\ \overline{@}\ \kappa\ (\underline{\mathtt{ref}}\ y)) \\
\llbracket !\ e \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \llbracket e \rrbracket(\overline{\lambda} y.\ \overline{@}\ \kappa\ (\underline{!}\ y)) \\
\llbracket e_1 := e_2 \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \llbracket e_1 \rrbracket(\overline{\lambda} y_1.\ \overline{@}\ \llbracket e_2 \rrbracket(\overline{\lambda} y_2.\ \overline{@}\ \kappa\ (y_1 \underline{:=} y_2))) \\
\llbracket (e_1, e_2) \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \llbracket e_1 \rrbracket(\overline{\lambda} y_1.\ \overline{@}\ \llbracket e_2 \rrbracket(\overline{\lambda} y_2.\ \overline{@}\ \kappa\ (\underline{(}y_1\ \underline{,}\ y_2\underline{)}))) \\
\llbracket \mathtt{inl}\ e \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \llbracket e \rrbracket(\overline{\lambda} y.\ \overline{@}\ \kappa\ (\underline{\mathtt{inl}}\ y)) \\
\llbracket \mathtt{inr}\ e \rrbracket & = & \overline{\lambda}\kappa.\ \overline{@}\ \llbracket e \rrbracket(\overline{\lambda} y.\ \overline{@}\ \kappa\ (\underline{\mathtt{inr}}\ y)) \\
\llbracket \mathtt{case}\ e\ \mathtt{of}\ y_1 \Rightarrow e_1\ \mathtt{or}\ y_2 \Rightarrow e_2 \rrbracket & = & \overline{\lambda}\kappa.\ \utilde{\mathtt{let}}\ k = \utilde{\lambda}\ a.\ \overline{@}\ \kappa\ a\ \utilde{\mathtt{in}}\ \overline{@}\ \llbracket e \rrbracket(\overline{\lambda} v. \\
& & \underline{\mathtt{case}}\ v\ \underline{\mathtt{of}}\ y_1 \Rightarrow \overline{@}\ \llbracket e_1 \rrbracket(\overline{\lambda}\ m.\ \utilde{@}\ \ k\ m)\ \underline{\mathtt{or}}\ y_2 \Rightarrow \overline{@}\ \llbracket e_2 \rrbracket(\overline{\lambda}\ n.\ \utilde{@}\ \ k\ n)) \\
\#\quad \llbracket \mathtt{shift}\ k\ \mathtt{in}\ e \rrbracket & = & \overline{\lambda}\kappa.\ \underline{\mathtt{let}}\ k = \underline{\lambda} a.\underline{\lambda}\kappa_1.\ \underline{@}\ \kappa_1(\overline{@}\ \kappa\ a)\ \underline{\mathtt{in}}\ \overline{@}\ \llbracket e \rrbracket(\overline{\lambda} m.m) \\
\llbracket \langle\ e\ \rangle \rrbracket & = & \overline{\lambda}\kappa.\overline{@}\ \kappa(\overline{@}\ \llbracket e \rrbracket(\overline{\lambda} m.m))
\end{array}
$$

Here we show the standard CPS transformation without using shift/reset. Similar to the figure before, rules and overline/underline notations are adapted from ?. The # symbol still denotes rules that are simplified due to Barendregt's variable convention, and we use the same wavy underline notation to handle properly tail-recursive calls.

2 FORWARD-MODE AD

For AD (both forward-mode and reverse-mode), we use the following variable sugaring with $\hat{}$ notation. Note that the variable sugaring is not strictly necessary but we find it convenient for + and ∗ rules. Also note that this variable sugaring are always used at position where we know for sure that the sugared variables bind with Real typed values, so that they must have gradients (denoted via variables with ').
Also note that for AD (both forward-mode and reverse-mode), we drop shift/reset terms from the source language, since the focus is to provide a semantics for AD in a standard language, and shift/reset will play a crucial role for the semantics of AD in reverse mode.
Note that our AD supports mutable states in the source language.

$$\textbf{Variable Sugaring: } \hat{y} \quad = \quad (y, y')$$

2.1 Forward-mode AD Transformation

$$\begin{array}{rcl}
\texttt{Transform}(f) & = & \lambda x.\, \texttt{let } \hat{y} = @\ \overrightarrow{\mathcal{D}}\llbracket f \rrbracket\ (x, 1)\ \texttt{in}\ \ y' \\
\texttt{where } \overrightarrow{\mathcal{D}}\llbracket . \rrbracket : & & \texttt{EXP} \longrightarrow \texttt{EXP} \quad \text{is defined as below:} \\
 & & \\
\overrightarrow{\mathcal{D}}\llbracket c \rrbracket. & = & c \ \text{ if } c \notin \mathbb{R} \\
\overrightarrow{\mathcal{D}}\llbracket c = \rrbracket & = & (c, 0) \ \text{ if } c \in \mathbb{R} \\
\overrightarrow{\mathcal{D}}\llbracket y \rrbracket & = & y \\
\overrightarrow{\mathcal{D}}\llbracket e_1 + e_2 \rrbracket & = & \texttt{let } \hat{y}_1 = \overrightarrow{\mathcal{D}}\llbracket e_1 \rrbracket \texttt{ in} \\
 & & \texttt{let } \hat{y}_2 = \overrightarrow{\mathcal{D}}\llbracket e_2 \rrbracket \texttt{ in} \\
 & & (y_1 + y_2, y_1' + y_2') \\
\overrightarrow{\mathcal{D}}\llbracket e_1 * e_2 \rrbracket & = & \texttt{let } \hat{y}_1 = \overrightarrow{\mathcal{D}}\llbracket e_1 \rrbracket \texttt{ in} \\
 & & \texttt{let } \hat{y}_2 = \overrightarrow{\mathcal{D}}\llbracket e_2 \rrbracket \texttt{ in} \\
 & & (y_1 * y_2, y_1 * y_2' + y_1' * y_2) \\
\overrightarrow{\mathcal{D}}\llbracket \lambda y.\, e \rrbracket & = & \lambda y.\, \overrightarrow{\mathcal{D}}\llbracket e \rrbracket \\
\overrightarrow{\mathcal{D}}\llbracket @\ e_1\ e_2 \rrbracket & = & @\ \overrightarrow{\mathcal{D}}\llbracket e_1 \rrbracket\ \overrightarrow{\mathcal{D}}\llbracket e_2 \rrbracket \\
\overrightarrow{\mathcal{D}}\llbracket \text{let } y = e_1 \text{ in } e_2 \rrbracket & = & \texttt{let } y = \overrightarrow{\mathcal{D}}\llbracket e_1 \rrbracket \texttt{ in } \overrightarrow{\mathcal{D}}\llbracket e_2 \rrbracket \\
\overrightarrow{\mathcal{D}}\llbracket \texttt{fst } e \rrbracket & = & \texttt{fst } \overrightarrow{\mathcal{D}}\llbracket e \rrbracket \\
\overrightarrow{\mathcal{D}}\llbracket \texttt{snd } e \rrbracket & = & \texttt{snd } \overrightarrow{\mathcal{D}}\llbracket e \rrbracket \\
\overrightarrow{\mathcal{D}}\llbracket \texttt{ref } e \rrbracket & = & \texttt{ref } \overrightarrow{\mathcal{D}}\llbracket e \rrbracket \\
\overrightarrow{\mathcal{D}}\llbracket !\ e \rrbracket & = & !\ \overrightarrow{\mathcal{D}}\llbracket e \rrbracket \\
\overrightarrow{\mathcal{D}}\llbracket e_1 := e_2 \rrbracket & = & \overrightarrow{\mathcal{D}}\llbracket e_1 \rrbracket := \overrightarrow{\mathcal{D}}\llbracket e_2 \rrbracket \\
\overrightarrow{\mathcal{D}}\llbracket (e_1, e_2) \rrbracket & = & (\overrightarrow{\mathcal{D}}\llbracket e_1 \rrbracket, \overrightarrow{\mathcal{D}}\llbracket e_2 \rrbracket) \\
\overrightarrow{\mathcal{D}}\llbracket \texttt{inl } e \rrbracket & = & \texttt{inl } \overrightarrow{\mathcal{D}}\llbracket e \rrbracket \\
\overrightarrow{\mathcal{D}}\llbracket \texttt{inr } e \rrbracket & = & \texttt{inr } \overrightarrow{\mathcal{D}}\llbracket e \rrbracket \\
\overrightarrow{\mathcal{D}}\llbracket \texttt{case } e \texttt{ of } y_1 \Rightarrow e_1 \texttt{ or } y_2 \Rightarrow e_2 \rrbracket & = & \texttt{case } \overrightarrow{\mathcal{D}}\llbracket e \rrbracket \texttt{ of } y_1 \Rightarrow \overrightarrow{\mathcal{D}}\llbracket e_1 \rrbracket \texttt{ or } y_2 \Rightarrow \overrightarrow{\mathcal{D}}\llbracket e_2 \rrbracket
\end{array}$$

Here we show the transformation for forward-mode AD. Note that there is no metalanguage redex generated in the transformation, so we elide underline denotations here, and by default let all constructs on the RHS be dynamic/target language constructs. Rules that are different from standard transformation are highlighted by color.

3 REVERSE-MODE AD

3.1 Reverse-mode AD Transformation using Target-Language shift/reset

$$
\begin{array}{rcl}
\texttt{Transform}(f) & = & \lambda x.\ \texttt{let}\ \hat{x} = (x, \texttt{ref}\ 0)\ \texttt{in} \\
 & & \quad \langle\ \texttt{let}\ \hat{z} = @\ \overleftarrow{\mathcal{D}}[\![f]\!]\ \hat{x}\ \texttt{in}\ z' := 1.0\ \rangle; \\
 & & \quad !\ x' \\
\texttt{where}\ \overleftarrow{\mathcal{D}}[\![.]\!]: & & \texttt{EXP} \longrightarrow \texttt{EXP}\quad \text{is defined as below:} \\
 & & \\
\overleftarrow{\mathcal{D}}[\![c]\!] & = & c\ \ \text{if}\ c \notin \mathbb{R} \\
\overleftarrow{\mathcal{D}}[\![c]\!] & = & (c, \texttt{ref}\ 0)\ \ \text{if}\ c \in \mathbb{R} \\
\overleftarrow{\mathcal{D}}[\![y]\!] & = & y \\
\overleftarrow{\mathcal{D}}[\![e_1 + e_2]\!] & = & \texttt{let}\ \hat{y}_1 = \overleftarrow{\mathcal{D}}[\![e_1]\!]\ \texttt{in} \\
 & & \texttt{let}\ \hat{y}_2 = \overleftarrow{\mathcal{D}}[\![e_2]\!]\ \texttt{in} \\
 & & \texttt{shift}\ \ k\ \texttt{in}\ \texttt{let}\ \hat{y} = (y_1 + y_2, \texttt{ref}\ 0)\ \texttt{in} \\
 & & \qquad\qquad @\ k\ \hat{y}; \\
 & & \qquad\qquad y_1' \ \mathrel{+}= !\ y'; \\
 & & \qquad\qquad y_2' \ \mathrel{+}= !\ y' \\
\overleftarrow{\mathcal{D}}[\![e_1 * e_2]\!] & = & \texttt{let}\ \hat{y}_1 = \overleftarrow{\mathcal{D}}[\![e_1]\!]\ \texttt{in} \\
 & & \texttt{let}\ \hat{y}_2 = \overleftarrow{\mathcal{D}}[\![e_2]\!]\ \texttt{in} \\
 & & \texttt{shift}\ \ k\ \texttt{in}\ \texttt{let}\ \hat{y} = (y_1 * y_2, \texttt{ref}\ 0)\ \texttt{in} \\
 & & \qquad\qquad @\ k\ \hat{y}; \\
 & & \qquad\qquad y_1' \ \mathrel{+}= !\ y' * y_2; \\
 & & \qquad\qquad y_2' \ \mathrel{+}= !\ y' * y_1 \\
\overleftarrow{\mathcal{D}}[\![\lambda y.\ e]\!] & = & \lambda y.\ \overleftarrow{\mathcal{D}}[\![e]\!] \\
\overleftarrow{\mathcal{D}}[\![@\ e_1\ e_2]\!] & = & @\ \overleftarrow{\mathcal{D}}[\![e_1]\!]\ \overleftarrow{\mathcal{D}}[\![e_2]\!] \\
\overleftarrow{\mathcal{D}}[\![\texttt{let}\ y = e_1\ \texttt{in}\ e_2]\!] & = & \texttt{let}\ y = \overleftarrow{\mathcal{D}}[\![e_1]\!]\ \texttt{in}\ \overleftarrow{\mathcal{D}}[\![e_2]\!] \\
\overleftarrow{\mathcal{D}}[\![\texttt{fst}\ e]\!] & = & \texttt{fst}\ \overleftarrow{\mathcal{D}}[\![e]\!] \\
\overleftarrow{\mathcal{D}}[\![\texttt{snd}\ e]\!] & = & \texttt{snd}\ \overleftarrow{\mathcal{D}}[\![e]\!] \\
\overleftarrow{\mathcal{D}}[\![\texttt{ref}\ e]\!] & = & \texttt{ref}\ \overleftarrow{\mathcal{D}}[\![e]\!] \\
\overleftarrow{\mathcal{D}}[\![!\ e]\!] & = & !\ \overleftarrow{\mathcal{D}}[\![e]\!] \\
\overleftarrow{\mathcal{D}}[\![e_1 := e_2]\!] & = & \overleftarrow{\mathcal{D}}[\![e_1]\!] := \overleftarrow{\mathcal{D}}[\![e_2]\!] \\
\overleftarrow{\mathcal{D}}[\![(e_1, e_2)]\!] & = & (\overleftarrow{\mathcal{D}}[\![e_1]\!], \overleftarrow{\mathcal{D}}[\![e_2]\!]) \\
\overleftarrow{\mathcal{D}}[\![\texttt{inl}\ e]\!] & = & \texttt{inl}\ \overleftarrow{\mathcal{D}}[\![e]\!] \\
\overleftarrow{\mathcal{D}}[\![\texttt{inr}\ e]\!] & = & \texttt{inr}\ \overleftarrow{\mathcal{D}}[\![e]\!] \\
\overleftarrow{\mathcal{D}}[\![\texttt{case}\ e\ \texttt{of}\ y_1 \Rightarrow e_1\ \texttt{or}\ y_2 \Rightarrow e_2]\!] & = & \texttt{case}\ \overleftarrow{\mathcal{D}}[\![e]\!]\ \texttt{of}\ y_1 \Rightarrow \overleftarrow{\mathcal{D}}[\![e_1]\!]\ \texttt{or}\ y_2 \Rightarrow \overleftarrow{\mathcal{D}}[\![e_2]\!]
\end{array}
$$

Transformation of reverse-mode AD with shift/reset and mutable state in the target language (identical to interpretation except for handling of environments). Rules that are different from standard transformation are highlighted by color.

3.2 Reverse-mode AD Transformation using Meta-Language Shift/Reset

$$
\begin{aligned}
\texttt{Transform}(f) \quad &= \quad \underline{\lambda} x.\ \underline{\texttt{let}}\ \hat{x} = \underline{(}x\ \underline{,}\ \underline{\texttt{ref}}\ 0\underline{)}\ \underline{\texttt{in}} \\
&\qquad \underline{@}\ (\underline{@}\ \overleftarrow{\mathcal{D}}[\![f]\!]\ \hat{x})\ (\underset{\sim}{\lambda}\, z.\ \underline{\texttt{let}}\ \hat{z} = z\ \underline{\texttt{in}}\ z' \ \underline{:=}\ 1.0)\underline{;} \\
&\qquad \underline{!}\ x' \\
\texttt{where}\ \overleftarrow{\mathcal{D}}[\![.]\!] : \quad & \quad \texttt{EXP} \longrightarrow \texttt{EXP} \quad \text{is defined as below:}
\end{aligned}
$$

$$
\begin{aligned}
\overleftarrow{\mathcal{D}}[\![c]\!] \quad &= \quad c \ \text{ if } c \notin \mathbb{R} \\
\overleftarrow{\mathcal{D}}[\![c]\!] \quad &= \quad \underline{(}c\ \underline{,}\ \underline{\texttt{ref}}\ 0\underline{)} \ \text{ if } c \in \mathbb{R} \\
\overleftarrow{\mathcal{D}}[\![y]\!] \quad &= \quad y \\
\overleftarrow{\mathcal{D}}[\![e_1 + e_2]\!] \quad &= \quad \overline{\texttt{shift}}\ k\ \overline{\texttt{in}}\ \underline{\texttt{let}}\ \hat{y}_1 = \overleftarrow{\mathcal{D}}[\![e_1]\!]\ \underline{\texttt{in}} \\
&\qquad\qquad \underline{\texttt{let}}\ \hat{y}_2 = \overleftarrow{\mathcal{D}}[\![e_2]\!]\ \underline{\texttt{in}} \\
&\qquad\qquad \underline{\texttt{let}}\ \hat{y} = \underline{(}\ y_1 \ \underline{+}\ y_2 \ \underline{,}\ \underline{\texttt{ref}}\ 0\underline{)}\ \underline{\texttt{in}} \\
&\qquad\qquad \overline{@}\ k\ \hat{y}\underline{;} \\
&\qquad\qquad y_1' \ \underline{+=}\ \underline{!}\ y'\underline{;} \\
&\qquad\qquad y_2' \ \underline{+=}\ \underline{!}\ y' \\
\overleftarrow{\mathcal{D}}[\![e_1 * e_2]\!] \quad &= \quad \overline{\texttt{shift}}\ k\ \overline{\texttt{in}}\ \underline{\texttt{let}}\ \hat{y}_1 = \overleftarrow{\mathcal{D}}[\![e_1]\!]\ \underline{\texttt{in}} \\
&\qquad\qquad \underline{\texttt{let}}\ \hat{y}_2 = \overleftarrow{\mathcal{D}}[\![e_2]\!]\ \underline{\texttt{in}} \\
&\qquad\qquad \underline{\texttt{let}}\ \hat{y} = \underline{(}\ y_1 \ \underline{*}\ y_2 \ \underline{,}\ \underline{\texttt{ref}}\ 0\underline{)}\ \underline{\texttt{in}} \\
&\qquad\qquad \overline{@}\ k\ \hat{y}\underline{;} \\
&\qquad\qquad y_1' \ \underline{+=}\ \underline{!}\ y' \ \underline{*}\ y_2 \ \underline{;} \\
&\qquad\qquad y_2' \ \underline{+=}\ \underline{!}\ y' \ \underline{*}\ y_1 \\
\overleftarrow{\mathcal{D}}[\![\lambda y.\ e]\!] \quad &= \quad \underline{\lambda} y.\ \underline{\lambda} k.\ \overline{\langle}\ \underset{\sim}{@}\ \ k\ \overleftarrow{\mathcal{D}}[\![e]\!]\ \overline{\rangle} \\
\overleftarrow{\mathcal{D}}[\![\ @\ e_1\ e_2]\!] \quad &= \quad \overline{\texttt{shift}}\ k\ \overline{\texttt{in}}\ \underline{@}\ (\underline{@}\ \overleftarrow{\mathcal{D}}[\![e_1]\!]\overleftarrow{\mathcal{D}}[\![e_2]\!])(\underset{\sim}{\lambda}\ a.\overline{@}\ k\ a) \\
\#\quad \overleftarrow{\mathcal{D}}[\![\texttt{let}\ y = e_1\ \texttt{in}\ e_2]\!] \quad &= \quad \overline{\texttt{shift}}\ k\ \overline{\texttt{in}}\ \underline{\texttt{let}}\ y = \overleftarrow{\mathcal{D}}[\![e_1]\!]\ \underline{\texttt{in}}\ \overline{\langle}\ \overline{@}\ k\ \overleftarrow{\mathcal{D}}[\![e_2]\!]\ \overline{\rangle} \\
\overleftarrow{\mathcal{D}}[\![\texttt{fst}\ e]\!] \quad &= \quad \underline{\texttt{fst}}\ \overleftarrow{\mathcal{D}}[\![e]\!] \\
\overleftarrow{\mathcal{D}}[\![\texttt{snd}\ e]\!] \quad &= \quad \underline{\texttt{snd}}\ \overleftarrow{\mathcal{D}}[\![e]\!] \\
\overleftarrow{\mathcal{D}}[\![\texttt{ref}\ e]\!] \quad &= \quad \underline{\texttt{ref}}\ \overleftarrow{\mathcal{D}}[\![e]\!] \\
\overleftarrow{\mathcal{D}}[\![!\ e]\!] \quad &= \quad \underline{!}\ \overleftarrow{\mathcal{D}}[\![e]\!] \\
\overleftarrow{\mathcal{D}}[\![e_1 := e_2]\!] \quad &= \quad \overleftarrow{\mathcal{D}}[\![e_1]\!]\ \underline{:=}\ \overleftarrow{\mathcal{D}}[\![e_2]\!] \\
\overleftarrow{\mathcal{D}}[\![(e_1, e_2)]\!] \quad &= \quad \underline{(}\overleftarrow{\mathcal{D}}[\![e_1]\!]\ \underline{,}\ \overleftarrow{\mathcal{D}}[\![e_2]\!]\underline{)} \\
\overleftarrow{\mathcal{D}}[\![\texttt{inl}\ e]\!] \quad &= \quad \underline{\texttt{inl}}\ \overleftarrow{\mathcal{D}}[\![e]\!] \\
\overleftarrow{\mathcal{D}}[\![\texttt{inr}\ e]\!] \quad &= \quad \underline{\texttt{inr}}\ \overleftarrow{\mathcal{D}}[\![e]\!] \\
\overleftarrow{\mathcal{D}}[\![\texttt{case}\ e\ \texttt{of}\ y_1 \Rightarrow e_1\ \texttt{or}\ y_2 \Rightarrow e_2]\!] \quad &= \quad \overline{\texttt{shift}}\ k\ \overline{\texttt{in}}\ \underset{\sim}{\texttt{let}}\ k_1 = \underset{\sim}{\lambda}\ a.\ \overline{@}\ k\ a\ \underset{\sim}{\texttt{in}} \\
&\quad \underline{\texttt{case}}\ \overleftarrow{\mathcal{D}}[\![e]\!]\ \underline{\texttt{of}}\ y_1 \Rightarrow \overline{\langle}\ \underset{\sim}{@}\ \ k_1\ [\![e_1]\!]\ \overline{\rangle}\ \underline{\texttt{or}}\ y_2 \Rightarrow \overline{\langle}\ \underset{\sim}{@}\ \ k_1\ [\![e_2]\!]\ \overline{\rangle}
\end{aligned}
$$

Transformation of reverse-mode AD with metalanguage that contains shift/reset. Rules that are different from standard transformation are labeled by color.

3.3 Reverse-mode AD Transformation in CPS

$$
\begin{array}{rcl}
\texttt{Transform}(f) & = & \underline{\lambda} x.\ \underline{\texttt{let}}\ \hat{x} = (x\ \underline{,}\ \underline{\texttt{ref}}\ 0)\ \underline{\texttt{in}} \\
& & \quad \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket f \rrbracket\ (\overline{\lambda} m.\ \underline{@}\ (\underline{@}\ m\ \hat{x})\ (\underset{\sim}{\lambda}\, z.\ \underline{\texttt{let}}\ \hat{z} = z\ \underline{\texttt{in}}\ z' \underline{:=} 1.0))\underline{;} \\
& & \quad \underline{!}\ x' \\
\texttt{where}\ \overleftarrow{\mathcal{D}}\llbracket . \rrbracket : & & \texttt{EXP} \longrightarrow \texttt{EXP} \ \text{ is defined as below:} \\
\\
\overleftarrow{\mathcal{D}}\llbracket c \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \kappa\ c \ \text{ if } c \notin \mathbb{R} \\
\overleftarrow{\mathcal{D}}\llbracket c \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \kappa\ \underline{(c\ \underline{,}\ \underline{\texttt{ref}}\ 0)} \ \text{ if } c \in \mathbb{R} \\
\overleftarrow{\mathcal{D}}\llbracket y \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \kappa\ y \\
\overleftarrow{\mathcal{D}}\llbracket e_1 + e_2 \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e_1 \rrbracket (\overline{\lambda} p_1.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e_2 \rrbracket (\overline{\lambda} p_2.\ \ \texttt{\#\#} \\
& & \quad \underline{\texttt{let}}\ \hat{y}_1 = p_1\ \underline{\texttt{in}}\ \ \underline{\texttt{let}}\ \hat{y}_2 = p_2\ \underline{\texttt{in}} \\
& & \quad \underline{\texttt{let}}\ \hat{y} = \underline{(}\ y_1\ \underline{+}\ y_2\ \ \underline{,}\ \underline{\texttt{ref}}\ 0\underline{)}\ \underline{\texttt{in}} \\
& & \quad \overline{@}\ \kappa\ \hat{y}\underline{;} \\
& & \quad y_1'\ \underline{+=}\ \underline{!}\ y'\underline{;} \\
& & \quad y_2'\ \underline{+=}\ \underline{!}\ y')) \\
\overleftarrow{\mathcal{D}}\llbracket e_1 * e_2 \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e_1 \rrbracket (\overline{\lambda} p_1.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e_2 \rrbracket (\overline{\lambda} p_2.\ \ \texttt{\#\#} \\
& & \quad \underline{\texttt{let}}\ \hat{y}_1 = p_1\ \underline{\texttt{in}}\ \ \underline{\texttt{let}}\ \hat{y}_2 = p_2\ \underline{\texttt{in}} \\
& & \quad \underline{\texttt{let}}\ \hat{y} = \underline{(}\ y_1\ \underline{*}\ y_2\ \ \underline{,}\ \underline{\texttt{ref}}\ 0\underline{)}\ \underline{\texttt{in}} \\
& & \quad \overline{@}\ \kappa\ \hat{y}\underline{;} \\
& & \quad y_1'\ \underline{+=}\ \ \underline{!}\ y'\ \underline{*}\ y_2\ \underline{;} \\
& & \quad y_2'\ \underline{+=}\ \ \underline{!}\ y'\ \underline{*}\ y_1\ )) \\
\overleftarrow{\mathcal{D}}\llbracket \lambda y.\ e \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \kappa\ (\underline{\lambda} y.\ \underline{\lambda} k.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e \rrbracket (\overline{\lambda}\ m.\ \underset{\sim}{@}\ \ k\ m)) \\
\overleftarrow{\mathcal{D}}\llbracket\ @\ e_1\ e_2 \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e_1 \rrbracket (\overline{\lambda} m.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e_2 \rrbracket (\overline{\lambda} n.\ \underline{@}\ (\ \underline{@}\ m\ n)\ (\underset{\sim}{\lambda}\ a.\ \overline{@}\ \kappa\ a))) \\
\texttt{\#}\ \ \overleftarrow{\mathcal{D}}\llbracket \texttt{let}\ y = e_1\ \texttt{in}\ e_2 \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e_1 \rrbracket (\overline{\lambda} y_1.\ \underline{\texttt{let}}\ y = y_1\ \underline{\texttt{in}}\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e_2 \rrbracket\ \kappa) \\
\overleftarrow{\mathcal{D}}\llbracket \texttt{fst}\ e \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e \rrbracket (\overline{\lambda} y.\ \overline{@}\ \kappa\ (\underline{\texttt{fst}}\ y)) \\
\overleftarrow{\mathcal{D}}\llbracket \texttt{snd}\ e \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e \rrbracket (\overline{\lambda} y.\ \overline{@}\ \kappa\ (\underline{\texttt{snd}}\ y)) \\
\overleftarrow{\mathcal{D}}\llbracket \texttt{ref}\ e \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e \rrbracket (\overline{\lambda} y.\ \overline{@}\ \kappa\ (\underline{\texttt{ref}}\ y)) \\
\overleftarrow{\mathcal{D}}\llbracket !\ e \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e \rrbracket (\overline{\lambda} y.\ \overline{@}\ \kappa\ (\ \underline{!}\ y)) \\
\overleftarrow{\mathcal{D}}\llbracket e_1 := e_2 \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e_1 \rrbracket (\overline{\lambda} y_1.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e_2 \rrbracket (\overline{\lambda} y_2.\ \overline{@}\ \kappa\ (y_1\ \underline{:=}\ y_2))) \\
\overleftarrow{\mathcal{D}}\llbracket (e_1, e_2) \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e_1 \rrbracket (\overline{\lambda} y_1.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e_2 \rrbracket (\overline{\lambda} y_2.\ \overline{@}\ \kappa\ (\underline{(}y_1\ \underline{,}\ y_2\underline{)}))) \\
\overleftarrow{\mathcal{D}}\llbracket \texttt{inl}\ e \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e \rrbracket (\overline{\lambda} y.\ \overline{@}\ \kappa\ (\underline{\texttt{inl}}\ y)) \\
\overleftarrow{\mathcal{D}}\llbracket \texttt{inr}\ e \rrbracket & = & \overline{\lambda} \kappa.\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e \rrbracket (\overline{\lambda} y.\ \overline{@}\ \kappa\ (\underline{\texttt{inr}}\ y)) \\
\overleftarrow{\mathcal{D}}\llbracket \texttt{case}\ e\ \texttt{of}\ y_1 \Rightarrow e_1\ \texttt{or}\ y_2 \Rightarrow e_2 \rrbracket & = & \overline{\lambda} \kappa.\ \underset{\sim}{\texttt{let}}\ k = \underset{\sim}{\lambda}\ a.\ \overline{@}\ \kappa\ a\ \underset{\sim}{\texttt{in}}\ \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e \rrbracket (\overline{\lambda} v.\ \underline{\texttt{case}}\ v \\
& & \underline{\texttt{of}}\ y_1 \Rightarrow \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e_1 \rrbracket (\overline{\lambda}\ m.\ \underset{\sim}{@}\ \ k\ m) \\
& & \underline{\texttt{or}}\ y_2 \Rightarrow \overline{@}\ \overleftarrow{\mathcal{D}}\llbracket e_2 \rrbracket (\overline{\lambda}\ n.\ \underset{\sim}{@}\ \ k\ n))
\end{array}
$$

Transformation of source language for reverse-mode AD in CPS (meta language doesn't contain shift/reset). Rules that are different from standard CPS transformation are highlighted by color. Note that in the plus rule and the multiplication rule (labeled by ##), we avoided using variable sugaring in $\overline{\lambda} p_1$ and $\overline{\lambda} p_2$ so that we can introduce dynamic let-binding for them. The dynamic let-bindings are necessary to preserve sharing, evaluation order, and asymptotic complexity, since the RHS of them are accessed multiple times via y_1, y_1', y_2, and y_2'.

4 EXAMPLES

Below we show examples of CPS transformation for reverse-mode AD. The results of transformation are only composed of target language expressions (all metalanguage redex have been removed). For clarity we elide the underline notations. We also drop the @ symbol in application for readability.

Loop Example: the function f computes power(x, l) in a loop

$$
\begin{array}{rcll}
\text{Term: } f & = & \lambda x.\ \texttt{let}\ l = \texttt{inr}\ (\texttt{inr}\ (\texttt{inl}\ ()))\ \texttt{in} & \\
& & \texttt{let}\ f_0 = \lambda f.\lambda ll.\lambda ac.\ \texttt{case}\ ll & \\
& & \texttt{of}\ y_1 \Rightarrow ac & \text{// if } l \text{ is 0, return } ac \\
& & \texttt{or}\ y_2 \Rightarrow f\ f\ y_2\ (x_0 * ac)\ \texttt{in} & \text{// if } l \text{ is not 0, recurse on } x_0 * ac \text{ and } l-1 \\
& & f_0\ f_0\ l\ 1 & \\
\multicolumn{4}{l}{\text{Transformation:}} \\
\texttt{Transform}(f) & = & \lambda x.\ \texttt{let}\ \hat{x} = (x, \texttt{ref}\ 0)\ \texttt{in}\ \ (\lambda \tilde{x}.\ (\lambda k. & \\
& & \quad \texttt{let}\ l = \texttt{inr}\ (\texttt{inr}\ (\texttt{inl}\ ()))\ \texttt{in} & \\
& & \quad \texttt{let}\ f_0 = \lambda f.\lambda k_1.\ k_1\ (\lambda ll.\lambda k_2.\ k_2\ (\lambda ac.\lambda k_3. & \\
& & \quad \texttt{case}\ ll\ \texttt{of}\ y_1 \Rightarrow k_3\ ac\ \texttt{or}\ y_2 \Rightarrow & \\
& & \qquad (f\ f)(\lambda a_3.\ (a_3\ y_2)\ (\lambda a_2. & \\
& & \qquad \texttt{let}\ \hat{z}_1 = x\ \texttt{in} & \\
& & \qquad \texttt{let}\ \hat{z}_2 = ac\ \texttt{in} & \\
& & \qquad \texttt{let}\ \hat{z} = (z_1 * z_2, \texttt{ref}\ 0)\ \texttt{in} & \\
& & \qquad\quad (a_2\ \hat{z})k_3; & \\
& & \qquad\quad z_1' \mathrel{+}= !\ z' * z_2; & \\
& & \qquad\quad z_2' \mathrel{+}= !\ z' * z_1)))))\ \texttt{in} & \\
& & \quad (f_0\ f_0)(\lambda a_{22}.\ (a_{22}\ l)\ (\lambda a_{11}.\ (a_{11}\ (1, \texttt{ref}\ 0))\ k))) & \\
& & \quad \hat{x}\ (\lambda y.\texttt{let}\ \hat{y} = y\ \texttt{in}\ y' := 1.0); & \\
& & \quad !\ x' &
\end{array}
$$

List Example: the function f computes the product of all reals in list l with x recursively

$$
\begin{array}{rcll}
\text{Term: } f & = & \lambda x.\ \texttt{let}\ l = \texttt{inr}\ (5, \texttt{inr}\ (6, \texttt{inl}\ ()))\ \texttt{in} & \\
& & \texttt{let}\ f_0 = \lambda f.\lambda ll.\ \texttt{case}\ ll & \\
& & \texttt{of}\ y_1 \Rightarrow x & \text{// if } l \text{ is empty, return } x \\
& & \texttt{or}\ y_2 \Rightarrow (\texttt{fst}\ y_2) * ((f\ f)(\texttt{snd}\ y_2))\ \texttt{in} & \text{// if } l \text{ is not empty, return } l.\text{head}{*}f(l.\text{tail}) \\
& & (f_0\ f_0)\ l & \\
\multicolumn{4}{l}{\text{Transformation:}} \\
\texttt{Transform}(f) & = & \lambda x.\ \texttt{let}\ \hat{x} = (x, \texttt{ref}\ 0)\ \texttt{in}\ \ (\lambda \tilde{x}.\ (\lambda k. & \\
& & \quad \texttt{let}\ l = \texttt{inr}\ ((5, \texttt{ref}\ 0), \texttt{inr}\ ((6, \texttt{ref}\ 0), \texttt{inl}\ ()))\ \texttt{in} & \\
& & \quad \texttt{let}\ f_0 = \lambda f.\lambda k_1.\ k_1\ (\lambda ll.\lambda k_2. & \\
& & \quad \texttt{case}\ ll\ \texttt{of}\ y_1 \Rightarrow k_2\ \tilde{x}\ \texttt{or}\ y_2 \Rightarrow & \\
& & \qquad (f\ f)(\lambda a_1.\ (a_1\ (\texttt{snd}\ y_2))\ (\lambda a. & \\
& & \qquad \texttt{let}\ \hat{z}_1 = \texttt{fst}\ y_2\ \texttt{in} & \\
& & \qquad \texttt{let}\ \hat{z}_2 = a\ \texttt{in} & \\
& & \qquad \texttt{let}\ \hat{z} = (z_1 * z_2, \texttt{ref}\ 0)\ \texttt{in} & \\
& & \qquad\quad k_2\ \hat{z}; & \\
& & \qquad\quad z_1' \mathrel{+}= !\ z' * z_2; & \\
& & \qquad\quad z_2' \mathrel{+}= !\ z' * z_1)))\ \texttt{in} & \\
& & \quad (f_0\ f_0)(\lambda a_{11}.(a_{11}\ l)\ k))) & \\
& & \quad \hat{x}\ (\lambda y.\texttt{let}\ \hat{y} = y\ \texttt{in}\ y' := 1.0); & \\
& & \quad !\ x' &
\end{array}
$$

Tree Example: the function f compute the sum of all reals in the binary tree t, with x in each leaf node

$$
\begin{array}{rcl}
\text{Term: } f & = & \lambda x.\ \texttt{let}\ t = \texttt{inr}\ (5, (\texttt{inl}\ (), \texttt{inl}\ ()))\ \texttt{in} \\
& & \texttt{let}\ f_0 = \lambda f.\lambda tt.\ \texttt{case}\ tt \\
& & \texttt{of}\ y_1 \Rightarrow x \qquad\qquad \text{// if } t \text{ is empty, return x} \\
& & \texttt{or}\ y_2 \Rightarrow (\texttt{fst}\ y_2) + \\
& & \qquad (f\ f\ (\texttt{fst}\ (\texttt{snd}\ y_2))) + \\
& & \qquad (f\ f\ (\texttt{snd}\ (\texttt{snd}\ y_2)))\ \texttt{in} \quad \text{// otherwise, return } t.\text{value}{+}f(t.\text{left}) + f(t.\text{right}) \\
& & (f_0\ f_0)\ t
\end{array}
$$

Transformation:

$$
\begin{array}{rcl}
\texttt{Transform}(f) & = & \lambda x.\ \texttt{let}\ \hat{x} = (x, \texttt{ref}\ 0)\ \texttt{in}\ \ (\lambda \tilde{x}.\ (\lambda k. \\
& & \quad \texttt{let}\ t = \texttt{inr}\ ((5, \texttt{ref}\ 0), (\texttt{inl}\ (), \texttt{inl}\ ()))\ \texttt{in} \\
& & \quad \texttt{let}\ f_0 = \lambda f.\lambda k_1.\ k_1\ (\lambda tt.\lambda k_2. \\
& & \quad \texttt{case}\ tt\ \texttt{of}\ y_1 \Rightarrow k_2\ \tilde{x}\ \texttt{or}\ y_2 \Rightarrow \\
& & \qquad (f\ f)(\lambda a_1.\ (a_1\ (\texttt{fst}\ (\texttt{snd}\ y_2)))\ (\lambda a. \\
& & \qquad \texttt{let}\ \hat{z}_1 = \texttt{fst}\ y_2\ \texttt{in} \\
& & \qquad \texttt{let}\ \hat{z}_2 = a\ \texttt{in} \\
& & \qquad \texttt{let}\ \hat{z} = (z_1 + z_2, \texttt{ref}\ 0)\ \texttt{in} \\
& & \qquad\ (f\ f)(\lambda b_1.\ (b_1\ (\texttt{snd}\ (\texttt{snd}\ y_2)))\ (\lambda b. \\
& & \qquad\quad \texttt{let}\ \hat{w}_1 = \hat{z}\ \texttt{in} \\
& & \qquad\quad \texttt{let}\ \hat{w}_2 = b\ \texttt{in} \\
& & \qquad\quad \texttt{let}\ \hat{w} = (w_1 + w_2, \texttt{ref}\ 0)\ \texttt{in} \\
& & \qquad\qquad k_2\ \hat{w}; \\
& & \qquad\qquad w_1' \mathrel{+}= !\ w'; \\
& & \qquad\qquad w_2' \mathrel{+}= !\ w'))) \\
& & \qquad\ z_1' \mathrel{+}= !\ z'; \\
& & \qquad\ z_2' \mathrel{+}= !\ z'))\ \texttt{in} \\
& & \quad (f_0\ f_0)(\lambda a_{11}.(a_{11}\ t)\ k))) \\
& & \quad \hat{x}\ (\lambda y.\texttt{let}\ \hat{y} = y\ \texttt{in}\ y' := 1.0); \\
& & \quad !\ x'
\end{array}
$$