Class Notes for CS 819 C - Spring 2002

8. Future Directions

John C. Reynolds
Department of Computer Science
Carnegie Mellon University

Revised May 17, 2002

Future Directions

1. New Forms of Inference
    - Strictly exact and domain-exact assertions
    - A generalization
    - Universally quantified intuitionistic assertions
    - Immiscible assertions
2. Taming Address Arithmetic
    - Problems with skewed sharing of dags, and Schorr-Waite records
    - Possible solution: introduce attributes of heap locations (similar to auxiliary variables of Owicki and Gries).
3. Concurrency
    - Soundness of rules for critical regions
    - Passivity
    - Extending separation logic to temporal logic
4. Storage Allocation
    - Regions
    - Tailor-made garbage collectors
5. The Relation to Type Systems
    - Walker and Morrisett's Alias Types

6. Embedded Code Pointers

    - Continuation-Passing Style (CPS)
    - No free variables in function expressions
    - The reflection operator (#)

Strictly Exact Assertions (Yang)

- An assertion q is strictly exact iff, for all s, h, and h', $[\![q]\!]_{\text{assert}}sh$ and $[\![q]\!]_{\text{assert}}sh'$ implies $h = h'$.
- Syntactically, assertions built from expressions using $\mapsto$ and $*$ are strictly exact.

Then:

$$((q \ * \ \mathbf{true}) \wedge p) \Rightarrow (q \ * \ (q \mathrel{-\!\!*} p)) \quad \text{when } q \text{ is strictly exact.}$$

Domain-Exact Assertions

- An assertion q is domain-exact iff, for all s, h, and h', $[\![q]\!]_{\text{assert}}sh$ and $[\![q]\!]_{\text{assert}}sh'$ implies $\operatorname{dom} h = \operatorname{dom} h'$.
- Syntactically, assertions built from expressions using $\mapsto$, $*$, and quantifiers are domain-exact.

When q is such an assertion, the semidistributive laws given earlier become full distributive laws:

$$(p_1 \ * \ q) \wedge (p_2 \ * \ q) \Rightarrow (p_1 \wedge p_2) \ * \ q$$
$$\text{when } q \text{ is domain-exact}$$

$$\forall x.\ (p \ * \ q) \Rightarrow (\forall x.\ p) \ * \ q$$
$$\text{when } x \text{ not free in } q \text{ and } q \text{ is domain-exact.}$$

A Generalization

Let $\simeq$ be a congruence on heaps, i.e., an equivalence relation on heaps such that, for all h_0, h_0', h_1, h_1',

$$h_0 \simeq h_0' \text{ and } h_1 \simeq h_1' \text{ and } h_0 \perp h_1 \text{ and } h_0' \perp h_1' \text{ implies } h_0 \cdot h_1 \simeq h_0' \cdot h_1'.$$

If

$$\forall s, h, h'.\ [\![p]\!]_{\text{assert}} sh \text{ and } [\![p]\!]_{\text{assert}} sh' \text{ implies } h \simeq h',$$

then p is said to be $\simeq$-*strict*. If

$$\forall s, h, h'.\ [\![q]\!]_{\text{assert}} sh \text{ and } h \simeq h' \text{ implies } [\![q]\!]_{\text{assert}} sh',$$

then q is said to be $\simeq$-*respecting*.

1. If $p \simeq q \stackrel{\text{def}}{=} p = q$, then $\simeq$-exact is strictly exact, and all assertions are $\simeq$-respecting.

2. If $p \simeq q \stackrel{\text{def}}{=} \operatorname{dom} p = \operatorname{dom} q$, then $\simeq$-exact is domain-exact, and $\simeq$-respecting is domain-respecting.

A Generalization (continued)

Proposition 11 *If p is $\simeq$-strict and q is $\simeq$-respecting, then*

$$\frac{p_1 \Rightarrow p \qquad p_2 \Rightarrow p}{((p_1 \ast \mathbf{true}) \land q) \Rightarrow (p_1 \ast (p_2 \mathrel{-\!\!*} q)).}$$

PROOF Suppose that $p_1 \Rightarrow p$ and $p_2 \Rightarrow p$ are valid, and that $[\![(p_1 \ast \mathbf{true}) \land q]\!]_{\text{assert}} sh$ is true. Then there are heaps h_0 and h_1 such that $h_0 \perp h_1$, $h = h_0 \cdot h_1$, $[\![p_1]\!]_{\text{assert}} sh_0$, and $[\![q]\!]_{\text{assert}} sh$.

To show that $[\![p_2 \mathrel{-\!\!*} q]\!]_{\text{assert}} sh_1$, assume $h_0' \perp h_1$ and $[\![p_2]\!]_{\text{assert}} sh_0'$. Since $p_1 \Rightarrow p$, we have $[\![p]\!]_{\text{assert}} sh_0$, and since $p_2 \Rightarrow p$, we have $[\![p]\!]_{\text{assert}} sh_0'$. Then, since p is $\simeq$-strict, $h_0 \simeq h_0'$, and since $\simeq$ is a congruence, $h = h_0 \cdot h_1 \simeq h_0' \cdot h_1$. Finally, since q is $\simeq$-respecting, $[\![q]\!]_{\text{assert}} s(h_0' \cdot h_1)$. END OF PROOF

Proposition 12 *Suppose, for all h and h', $h \simeq h'$ implies $\operatorname{dom} h = \operatorname{dom} h'$. If q is $\simeq$-respecting, then $p \mathrel{-\!\!*} q$ is $\simeq$-respecting.*

Universally Quantified Intuitionistic Assertions

Suppose $\hat{p}$ is intuitionistic, and let p be

$$\forall x.\ \mathsf{allocated}(x) \Rightarrow \hat{p}.$$

Then

$$\begin{aligned} \mathbf{emp} &\Rightarrow p \\ (\exists x.\ (x \mapsto -,-,-,-) \wedge \hat{p}) &\Rightarrow p \\ (p * p) &\Rightarrow p. \end{aligned}$$

Immiscible Assertions

Suppose p and q are *immiscible* iff, for all stores s and heaps h and h' such that $h \cup h'$ is a function, if $[\![p]\!]_{\text{assert}} sh$ and $[\![q]\!]_{\text{assert}} sh'$ then $h \perp h'$. Then

1. If $\hat{p}$ and $\hat{q}$ are intuitionistic and $\hat{p} \wedge \hat{q} \Rightarrow \mathbf{false}$ is valid, then

   $$\forall x.\ \mathsf{allocated}(x) \Rightarrow \hat{p} \qquad \text{and} \qquad \forall x.\ \mathsf{allocated}(x) \Rightarrow \hat{q}$$

   are immiscible.

2. If p' and q' are immiscible and $p \Rightarrow p'$ and $q \Rightarrow q'$ are valid, then p and q are immiscible.

3. If p and q are immiscible, then

   $$(p * \mathbf{true}) \wedge (q * r) \Rightarrow q * ((p * \mathbf{true}) \wedge r)$$

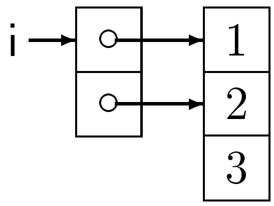
   is valid.

Taming Address Arithmetic

Our definition of dag permits *skewed sharing.* For example,

$$\mathsf{dag}\,((1 \cdot 2) \cdot (2 \cdot 3))\,(\mathsf{i})$$

holds when



Similar problems would occur for the Schorr-Waite proof in the logic with address arithmetic.

A Possible Solution: Auxiliary Attributes

- To redefine **dag** to avoid skewed sharing, give x (but not $x+1$) the attribute **dag-record** whenever $x{:=}\mathbf{cons}(e_1, e_2)$ creates a record in a dag. Then define

$$\begin{aligned}&\mathsf{dag}\,(\tau_1 \cdot \tau_2)\,(i) \text{ iff } \textbf{is-dag-record}(i) \wedge \\ &\qquad \exists i_1, i_2.\ i \mapsto i_1, i_2 \ * \ (\mathsf{dag}\,\tau_1\,(i_1) \wedge \mathsf{dag}\,\tau_2\,(i_2)).\end{aligned}$$

- For Schorr-Waite, give the attribute **record** to the address of the beginning of each record. Then add

$$\begin{aligned}&\forall \mathsf{x}.\ \ \textbf{is-record}(\mathsf{x}) \Rightarrow \exists \mathsf{l}, \mathsf{r}, \mathsf{c}, \mathsf{m}.\ \ \mathsf{x} \hookrightarrow \mathsf{l}, \mathsf{r}, \mathsf{c}, \mathsf{m} \\ &\quad \wedge (\textbf{is-record}(\mathsf{l}) \vee \mathsf{l} = \mathbf{nil}) \wedge (\textbf{is-record}(\mathsf{r}) \vee \mathsf{r} = \mathbf{nil}) \\ &\quad \wedge \textbf{is-boolean}(\mathsf{c}) \wedge \textbf{is-boolean}(\mathsf{m})\end{aligned}$$

  to the invariant, and use $\textbf{is-record}(x)$ in place of $\mathsf{allocated}(x)$.

Concurrency

- Non-Interfering Concurrency

$$\frac{\{p_1\}\ c_1\ \{q_1\} \qquad \{p_2\}\ c_2\ \{q_2\}}{\{p_1\ *\ p_2\}\ c_1 \parallel c_2\ \{q_1\ *\ q_2\},}$$

  where no variable free in p_1 or q_1 is modified by c_2, or vice-versa.

- Resources and Critical Regions (need to show soundness)
- Passivity
- Adding separating operators to temporal logic.

Storage Allocation

Regions

$$r := \textbf{allocate-region}\ e$$

$$x := \textbf{cons}(e_1, \ldots, e_n)\ \textbf{in}\ r$$

$$x := \textbf{allocate}\ e\ \textbf{in}\ r$$

$$r := \textbf{allocate-region}\ e\ \textbf{in}\ r'.$$

Tailor-Made Garbage Collectors

View 1:
In the list-reversal program, the garbage collector must preserve the invariant

$$\exists\alpha,\beta.\ (\textbf{list}\,\alpha\,(\mathsf{i},\textbf{nil})\ *\ \textbf{list}\,\beta\,(\mathsf{j},\textbf{nil}))\wedge\gamma^\dagger=\alpha^\dagger{\cdot}\beta,$$

while preserving the input variable γ, and not increasing the length of α.

View 2:
The garbage collector must preserve

$$(\textbf{list}\,\alpha\,(\mathsf{i},\textbf{nil})\ *\ \textbf{list}\,\beta\,(\mathsf{j},\textbf{nil}))\wedge\gamma^\dagger=\alpha^\dagger{\cdot}\beta,$$

while preserving the values of the abstract values α, β, and γ.

The Relation to Type Systems

The following system can be translated into either separation logic or the *alias types* of Walker and Morrisett:

$$\langle\text{value type}\rangle ::= \mathbf{integer} \mid \langle\text{pointer variable}\rangle \qquad (\tau)$$

$$\begin{aligned}\langle\text{context}\rangle ::= \langle\text{variable}\rangle\colon \langle\text{value type}\rangle, \ldots,&\\ \langle\text{variable}\rangle\colon \langle\text{value type}\rangle&\end{aligned} \qquad (\Gamma)$$

$$\begin{aligned}\langle\text{heap type}\rangle ::= {}&\mathbf{emp}\\ &\mid \langle\text{pointer variable}\rangle \mapsto \langle\text{value type}\rangle, \ldots, \langle\text{value type}\rangle\\ &\mid \langle\text{heap type}\rangle \ * \ \langle\text{heap type}\rangle\end{aligned} \qquad (\mathcal{C})$$

$$\langle\text{state type}\rangle ::= \langle\text{heap type}\rangle \ ; \langle\text{context}\rangle \qquad (P)$$

Rules of Inference

$$\frac{}{\{P\}\ \mathbf{skip}\ \{P\}}$$

$$\frac{\{P\}\ C\ \{P'\} \qquad \{P'\}\ C'\ \{P''\}}{\{P\}\ C\,;C'\ \{P''\}}$$

$$\frac{P \vdash P' \qquad \{P'\}\ C\ \{P''\}}{\{P\}\ C\ \{P''\}}$$

$$\frac{\{P\}\ C\ \{P'\} \qquad P' \vdash P''}{\{P\}\ C\ \{P''\}}$$

$$\frac{P \vdash E : \mathbf{bool} \qquad \{P\}\ C_1\ \{P'\} \qquad \{P\}\ C_2\ \{P'\}}{\{P\}\ \mathbf{if}\ E\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \{P'\}}$$

$$\frac{P \vdash E : \mathbf{bool} \qquad \{P\}\ C\ \{P\}}{\{P\}\ \mathbf{while}\ E\ \mathbf{do}\ C\ \{P\}}$$

More Rules of Inference

$$\frac{\Gamma \vdash E : \tau}{\{\mathcal{C}\,;\Gamma\}\ x := E\ \{\mathcal{C}\,;[\,\Gamma \mid x{:}\,\tau\,]\}}$$

$$\frac{\Gamma \vdash E : \rho}{\{\mathcal{C} \wedge ((\rho \mapsto -, \ldots, \tau_i, \ldots, -)\ *\ \mathbf{true})\,;\Gamma\}\ x := E.i\ \{\mathcal{C}\,;[\,\Gamma \mid x{:}\,\tau_i\,]\}}$$

$$\frac{}{\{\mathcal{C}\,;\Gamma\}\ x := \mathbf{allocate}\ i\ \{\mathcal{C}\ *\ (\rho \mapsto -, \ldots, -)\,;[\,\Gamma \mid x{:}\,\rho\,]\}}$$

where ρ is not free in $\mathcal{C}$ or Γ.

$$\frac{\Gamma \vdash E : \rho \qquad \Gamma \vdash E' : \tau}{\begin{array}{l}\{\mathcal{C}\ *\ (\rho \mapsto \tau_1, \ldots, \tau_i, \ldots, \tau_n)\,;\Gamma\}\\ E.i := E'\\ \{\mathcal{C}\ *\ (\rho \mapsto \tau_1, \ldots, \tau, \ldots, \tau_n)\,;\Gamma\}\end{array}}$$

$$\frac{\Gamma \vdash E : \rho}{\{\mathcal{C}\ *\ (\rho \mapsto -, \ldots, -)\,;\Gamma\}\ \mathbf{dispose}\ E\ \{\mathcal{C}\,;\Gamma\}}$$

An Example of a Type Analysis

$\{\rho_1 \mapsto \mathbf{int}, \rho_2 \ * \ \rho_2 \mapsto \mathbf{int}, \mathbf{nil}\ ; \mathsf{x}\colon \rho_1\}$

$\mathsf{y} := \mathbf{allocate}\ 2\ ;$

$\{\rho_1 \mapsto \mathbf{int}, \rho_2 \ * \ \rho_2 \mapsto \mathbf{int}, \mathbf{nil} \ * \ \rho_1' \mapsto -, -\ ; \mathsf{x}\colon \rho_1, \mathsf{y}\colon \rho_1'\}$

$\mathsf{y}.1 := 1\ ;$

$\{\rho_1 \mapsto \mathbf{int}, \rho_2 \ * \ \rho_2 \mapsto \mathbf{int}, \mathbf{nil} \ * \ \rho_1' \mapsto \mathbf{int}, -\ ; \mathsf{x}\colon \rho_1, \mathsf{y}\colon \rho_1'\}$

$\{(\rho_1 \mapsto \mathbf{int}, \rho_2 \ * \ \rho_2 \mapsto \mathbf{int}, \mathbf{nil} \ * \ \rho_1' \mapsto \mathbf{int}, -) \wedge$
$\quad (\rho_1 \mapsto -, \rho_2 \ * \ \mathbf{true})\ ; \mathsf{x}\colon \rho_1, \mathsf{y}\colon \rho_1'\}$

$\mathsf{z} := \mathsf{x}.2\ ;$

$\{\rho_1 \mapsto \mathbf{int}, \rho_2 \ * \ \rho_2 \mapsto \mathbf{int}, \mathbf{nil} \ * \ \rho_1' \mapsto \mathbf{int}, -\ ;$
$\quad \mathsf{x}\colon \rho_1, \mathsf{y}\colon \rho_1', \mathsf{z}\colon \rho_2\}$

$\mathsf{y}.2 := \mathsf{z}\ ;$

$\{\rho_1 \mapsto \mathbf{int}, \rho_2 \ * \ \rho_2 \mapsto \mathbf{int}, \mathbf{nil} \ * \ \rho_1' \mapsto \mathbf{int}, \rho_2\ ;$
$\quad \mathsf{x}\colon \rho_1, \mathsf{y}\colon \rho_1', \mathsf{z}\colon \rho_2\}$

$\{\rho_1 \mapsto \mathbf{int}, \rho_2 \ * \ \rho_2 \mapsto \mathbf{int}, \mathbf{nil} \ * \ \rho_1' \mapsto \mathbf{int}, \rho_2\ ; \mathsf{x}\colon \rho_1, \mathsf{y}\colon \rho_1'\}$

$\mathbf{dispose}\ \mathsf{x}\ ;$

$\{\rho_2 \mapsto \mathbf{int}, \mathbf{nil} \ * \ \rho_1' \mapsto \mathbf{int}, \rho_2\ ; \mathsf{x}\colon \rho_1, \mathsf{y}\colon \rho_1'\}$

$\mathsf{x} := \mathsf{y}$

$\{\rho_2 \mapsto \mathbf{int}, \mathbf{nil} \ * \ \rho_1' \mapsto \mathbf{int}, \rho_2\ ; \mathsf{x}\colon \rho_1', \mathsf{y}\colon \rho_1'\}$

$\{\rho_1' \mapsto \mathbf{int}, \rho_2 \ * \ \rho_2 \mapsto \mathbf{int}, \mathbf{nil}\ ; \mathsf{x}\colon \rho_1'\}$

Embedded Code Pointers

- To encompass embedded code pointers, we must move beyond the simple imperative language and Hoare logic.
- One possibility is a higher-order functional language with side effects, with two restrictions:
  - Continuation-passing style (CPS) is used, so that functions receive their return addresses in the same way as they receive other parameters.
  - Free variables are prohibited in expressions that denote functions, so that functions can be represented by code pointers, rather than by closures that pair code pointers with data.

Flat Commands versus Continuation Passing Style

$$
\begin{aligned}
\langle \mathrm{fc} \rangle ::= {} & \langle \mathrm{var} \rangle := \langle \mathrm{exp} \rangle \; ; \langle \mathrm{fc} \rangle \\
& \mid \langle \mathrm{var} \rangle := \mathbf{cons}(\langle \mathrm{exp} \rangle, \ldots, \langle \mathrm{exp} \rangle) \; ; \langle \mathrm{fc} \rangle \\
& \mid \langle \mathrm{var} \rangle := [\langle \mathrm{exp} \rangle] \; ; \langle \mathrm{fc} \rangle \\
& \mid [\langle \mathrm{exp} \rangle] := \langle \mathrm{exp} \rangle \; ; \langle \mathrm{fc} \rangle \\
& \mid \mathbf{dispose} \; \langle \mathrm{exp} \rangle \; ; \langle \mathrm{fc} \rangle \\
& \mid \mathbf{if} \; \langle \mathrm{boolexp} \rangle \; \mathbf{then} \; \langle \mathrm{fc} \rangle \; \mathbf{else} \; \langle \mathrm{fc} \rangle \\
& \mid \mathbf{goto} \; \langle \mathrm{var} \rangle \\
& \mid \langle \mathrm{var} \rangle(\langle \mathrm{exp} \rangle, \ldots, \langle \mathrm{exp} \rangle)
\end{aligned}
$$

$$
\begin{aligned}
\langle \mathrm{cpst} \rangle ::= {} & \mathbf{let} \; \langle \mathrm{var} \rangle = \langle \mathrm{exp} \rangle \; \mathbf{in} \; \langle \mathrm{cpst} \rangle \\
& \mid \mathbf{let} \; \langle \mathrm{var} \rangle = \mathbf{cons}(\langle \mathrm{exp} \rangle, \ldots, \langle \mathrm{exp} \rangle) \; \mathbf{in} \; \langle \mathrm{cpst} \rangle \\
& \mid \mathbf{let} \; \langle \mathrm{var} \rangle = [\langle \mathrm{exp} \rangle] \; \mathbf{in} \; \langle \mathrm{cpst} \rangle \\
& \mid [\langle \mathrm{exp} \rangle] := \langle \mathrm{exp} \rangle \; ; \langle \mathrm{cpst} \rangle \\
& \mid \mathbf{dispose} \; \langle \mathrm{exp} \rangle \; ; \langle \mathrm{cpst} \rangle \\
& \mid \mathbf{if} \; \langle \mathrm{boolexp} \rangle \; \mathbf{then} \; \langle \mathrm{cpst} \rangle \; \mathbf{else} \; \langle \mathrm{cpst} \rangle \\
& \mid \langle \mathrm{var} \rangle(\langle \mathrm{exp} \rangle, \ldots, \langle \mathrm{exp} \rangle)
\end{aligned}
$$

Continuation Passing Style - An Example

Consider a function for appending lists (a la LISP):

$$\begin{array}{l}\textbf{letrec}\ \mathsf{append}(\mathsf{x},\mathsf{y}) = \\ \quad \textbf{if}\ \mathsf{x} = \textbf{nil then}\ \mathsf{y}\ \textbf{else cons}([\mathsf{x}], \mathsf{append}([\mathsf{x}+1], \mathsf{y})).\end{array}$$

In continuation passing style, this becomes

$$\begin{array}{l}\textbf{letrec}\ \mathsf{append}(\mathsf{x},\mathsf{y},\mathsf{r}) = \textbf{if}\ \mathsf{x} = \textbf{nil then}\ \mathsf{r}(\mathsf{y})\ \textbf{else} \\ \quad \textbf{let}\ \mathsf{a} = [\mathsf{x}], \mathsf{b} = [\mathsf{x}+1], \\ \qquad \mathsf{k}(\mathsf{z}) = \textbf{let}\ \mathsf{w} = \textbf{cons}(\mathsf{a},\mathsf{z})\ \textbf{in}\ \mathsf{r}(\mathsf{w}) \\ \quad \textbf{in append}(\mathsf{b},\mathsf{y},\mathsf{k}).\end{array}$$

The Reflection Operator

If t is a CPS term, then $\# \, t$ is an assertion that holds for a state if t never aborts when executed in that state — in this situation we say that it is *safe* to execute t in that state.

Suppose

- c is a command satisfying $\{p\}\ c\ \{q\}$.
- t is a CPS term such that executing t has the same effect as executing c and then calling the continuation $r(x_1, ...x_n)$.

Then we can specify t by the assertion

$$(p \, * \, (\forall x_1, \ldots, x_m. \ q \mathrel{-\!\!*} \# \, r(x_1, \ldots, x_n))) \Rightarrow \# \, t.$$

- Roughly, this asserts that it is safe to execute t in any state satisfying p and mapping r into a procedure such that it is safe to execute $r(x_1, ..., x_n)$ in a state satisfying q.
- More precisely, it asserts that it is safe to execute t in any state where part of the heap satisfies p, and r is mapped into a procedure such that it is safe to execute $r(x_1, ..., x_n)$ when the part of the heap that satisfied p is replaced by a part that satisfies q.
- Still more precisely, $x_1, \ldots, x_m \in \{x_1, \ldots, x_n\}$ are the variables whose value may be changed by t before it executes $r(x_1, ..., x_n)$.

An Example

$$
\begin{array}{l}
\textbf{letrec}\ \mathsf{append}(\mathsf{x},\mathsf{y},\mathsf{r}) = \textbf{if}\ \mathsf{x} = \textbf{nil}\ \textbf{then}\ \mathsf{r}(\mathsf{y})\ \textbf{else} \\
\quad \textbf{let}\ \mathsf{a} = [\mathsf{x}], \mathsf{b} = [\mathsf{x}+1], \\
\qquad \mathsf{k}(\mathsf{z}) = \textbf{let}\ \mathsf{w} = \textbf{cons}(\mathsf{a},\mathsf{z})\ \textbf{in}\ \mathsf{r}(\mathsf{w}) \\
\quad \textbf{in append}(\mathsf{b},\mathsf{y},\mathsf{k})
\end{array}
$$

is specified by

$$
\begin{array}{l}
\Big((\mathsf{list}\,\alpha\,(\mathsf{x},\mathbf{nil}) * \mathsf{list}\,\beta\,(\mathsf{y},\mathbf{nil})) \\
\qquad * \Big(\forall \mathsf{z}.\ \ (\mathsf{list}\,\alpha\,(\mathsf{x},\mathbf{nil}) * \mathsf{list}\,\alpha\,(\mathsf{z},\mathsf{y}) * \mathsf{list}\,\beta\,(\mathsf{y},\mathbf{nil})) \\
\qquad\qquad -\!\!* \ \#\, \mathsf{r}(\mathsf{z})\Big)\Big) \\
\quad \Rightarrow \#\, \mathsf{append}(\mathsf{x},\mathsf{y},\mathsf{r}).
\end{array}
$$

Eliminating Global Variables

$$
\begin{array}{l}
\textbf{letrec}\ \mathsf{append}(\mathsf{x},\mathsf{y},\mathsf{r}) = \textbf{if}\ \mathsf{x} = \textbf{nil then}\ \mathsf{r}(\mathsf{y})\ \textbf{else} \\
\quad \textbf{let}\ \mathsf{a} = [\mathsf{x}], \mathsf{b} = [\mathsf{x}+1], \\
\qquad \mathsf{k}(\mathsf{z}) = \textbf{let}\ \mathsf{w} = \textbf{cons}(\mathsf{a},\mathsf{z})\ \textbf{in}\ \mathsf{r}(\mathsf{w}) \\
\quad \textbf{in append}(\mathsf{b},\mathsf{y},\mathsf{k})
\end{array}
$$

can be transformed into

$$
\begin{array}{l}
\textbf{letrec}\ \mathsf{append}(\mathsf{x},\mathsf{y},\mathsf{r},\mathsf{rr}) = \textbf{if}\ \mathsf{x} = \textbf{nil then}\ \mathsf{r}(\mathsf{rr},\mathsf{y})\ \textbf{else} \\
\quad \textbf{let}\ \mathsf{a} = [\mathsf{x}], \mathsf{b} = [\mathsf{x}+1], \mathsf{kk} = \textbf{cons}(\mathsf{r},\mathsf{a},\mathsf{rr}), \\
\qquad \mathsf{k}(\mathsf{kk},\mathsf{z}) = \\
\qquad\quad (\textbf{let}\ \mathsf{r} = [\mathsf{kk}], \mathsf{a} = [\mathsf{kk}+1], \mathsf{rr} = [\mathsf{kk}+2] \\
\qquad\quad \textbf{in}\ (\textbf{dispose}\ \mathsf{kk}, \mathsf{kk}+1, \mathsf{kk}+2\ ; \\
\qquad\qquad \textbf{let}\ \mathsf{w} = \textbf{cons}(\mathsf{a},\mathsf{z})\ \textbf{in}\ \mathsf{r}(\mathsf{rr},\mathsf{w})) \\
\quad \textbf{in append}(\mathsf{b},\mathsf{y},\mathsf{k},\mathsf{kk}),
\end{array}
$$

which is specified by

$$
\begin{array}{l}
\Big((\mathsf{list}\,\alpha\,(\mathsf{x},\textbf{nil}) * \mathsf{list}\,\beta\,(\mathsf{y},\textbf{nil})) \\
\qquad * \ \Big(\forall \mathsf{z}.\ \ (\mathsf{list}\,\alpha\,(\mathsf{x},\textbf{nil}) * \mathsf{list}\,\alpha\,(\mathsf{z},\mathsf{y}) * \mathsf{list}\,\beta\,(\mathsf{y},\textbf{nil})) \\
\qquad\quad -\!\!* \ \#\ \mathsf{r}(\mathsf{rr},\mathsf{z})\Big)\Big) \\
\quad \Rightarrow \#\ \mathsf{append}(\mathsf{x},\mathsf{y},\mathsf{r},\mathsf{rr}).
\end{array}
$$