B. Appendix: Models

B.1 Observational Congruence

Programs Write $(\nu\tilde{l})(M,\sigma) \Downarrow (\nu\tilde{l}')(V,\sigma')$ for $(\nu\tilde{l})(M,\sigma) \rightarrow^* (\nu\tilde{l}')(V,\sigma')$; and $(\nu\tilde{l})(M,\sigma) \Downarrow$ for $(\nu\tilde{l})(M,\sigma) \Downarrow (\nu\tilde{l}')(V,\sigma')$ for some $(\nu\tilde{l}')(V,\sigma')$. Let $\Gamma;\Delta \vdash M_{1,2} : \alpha$. Then we write $\Gamma;\Delta \vdash (\nu\tilde{l}_1)(M_1,\sigma_1) \cong (\nu\tilde{l}_2)(M_2,\sigma_2)$ if, for each typed closing context $C[\,\cdot\,]$ of type Unit which is typable under Δ and in which no labels from $\tilde{l}_{1,2}$ occur, we have:

$$(\nu\tilde{l}_1)(C[M_1],\ \sigma_1)\Downarrow \quad \text{iff} \quad (\nu\tilde{l}_2)(C[M_2],\ \sigma_2)\Downarrow$$

We often write $(\nu\tilde{l}_1)(M_1,\sigma_1) \cong (\nu\tilde{l}_2)(M_2,\sigma_2)$, leaving type information implicit.

Models Given models $\mathcal{M}_i^{\Gamma;\Delta} = (\nu\tilde{l}_i)(\{y_i : V_{i1},..,y_i : V_{in}\},\sigma_i)$ for $i = 1,2$, we set $\Gamma;\Delta \vdash \mathcal{M}_1 \approx \mathcal{M}_2$ iff

$$(\nu\tilde{l}_1)(\langle V_{11},..,V_{1n}\rangle,\sigma_1) \cong (\nu\tilde{l}_2)(\langle V_{21},..,V_{2n}\rangle,\sigma_2)$$

B.2 Semantics

Let $\Gamma;\Delta \vdash e : \alpha$, $\Gamma;\Delta \vdash \mathcal{M}$ and $\mathcal{M} = (\nu\tilde{l})(\xi,\sigma)$. Then the *interpretation of e under* (ξ,σ), denoted $[\![e]\!]_{\xi,\sigma}$ is inductively given by:

$$[\![x]\!]_{\xi,\sigma} = \xi(x) \qquad [\![!e]\!]_{\xi,\sigma} = \sigma([\![e]\!]_{\xi,\sigma})$$
$$[\![()]\!]_{\xi,\sigma} = () \quad [\![\mathtt{n}]\!]_{\xi,\sigma} = \mathsf{n} \qquad [\![\mathtt{b}]\!]_{\xi,\sigma} = \mathsf{b} \quad [\![l]\!]_{\xi,\sigma} = l$$
$$[\![\mathsf{op}(\tilde{e})]\!]_{\xi,\sigma} = \mathsf{op}([\![\tilde{e}]\!]_{\xi,\sigma}) \qquad [\![\langle e,e'\rangle]\!]_{\xi,\sigma} = \langle [\![e]\!]_{\xi,\sigma}, [\![e']\!]_{\xi,\sigma}\rangle$$
$$[\![\pi_i(e)]\!]_{\xi,\sigma} = \pi_i([\![e]\!]_{\xi,\sigma}) \qquad [\![\mathtt{inj}_i(e)]\!]_{\xi,\sigma} = \mathsf{inj}_i([\![e]\!]_{\xi,\sigma})$$

Then we define semantics of the assertions as follows (the new notations are illustrated below): All omitted cases are by de Morgan duality. Let u,u',u'' be fresh.

- $\mathcal{M} \models e_1 = e_2$ if $\mathcal{M}[u : e_1] \approx \mathcal{M}[u : e_2]$.
- $\mathcal{M} \models C_1 \wedge C_2$ if $\mathcal{M} \models C_1$ and $\mathcal{M} \models C_2$.
- $\mathcal{M} \models \neg C$ if not $\mathcal{M} \models C$.
- $\mathcal{M} \models \forall x^{\alpha}.C$ if (1) $\forall e.(\mathcal{M}[x : e] \models C)$ and $\forall V.(\mathcal{M}[x : V] \models C)$ when α is any type; and (2) $\forall \mathcal{M}'.((\nu l)(\mathcal{M}'/x) \approx \mathcal{M} \supset \mathcal{M}' \models C)$ s.t. $\mathcal{M}'(x) = l$ when α is a reference type.
- $\mathcal{M} \models \forall \mathrm{X}.C$ if for all closed type α, $\mathcal{M}\cdot\mathrm{X}:\alpha \models C$.
- $\mathcal{M} \models [!x]C$ if $\forall \mathcal{M}'.(\mathcal{M} \overset{[!x]}{\approx} \mathcal{M}' \supset \mathcal{M}' \models C)$.
- $\mathcal{M} \models \{C\}e\bullet e'{=}x\{C'\}$ if, whenever $\mathcal{M}[u{:}N] \Downarrow \mathcal{M}_0$ and $\mathcal{M}_0/u \models C$ for some N, we have $\mathcal{M}[x : L] \Downarrow \mathcal{M}' \models C'$ where we set $L \stackrel{\text{def}}{=} \mathtt{let}\ u = e\ \mathtt{in}\ u' = e'\ \mathtt{in}\ \mathtt{let}\ u'' = N\ \mathtt{in}\ uu'$.
- $\mathcal{M} \models e_1 \hookrightarrow e_2$ if $(\nu\tilde{l})(\xi,\sigma) \approx \mathcal{M}$ implies $[\![e_2]\!]_{\xi,\sigma} \in \mathsf{ncl}(\mathsf{fl}([\![e_1]\!]_{\xi,\sigma}),\sigma)$

Above we use the following notations (assuming well-typedness): $\mathcal{M}[u{:}N] \Downarrow \mathcal{M}'$ appears in § 3.1. $\mathcal{M}[e \mapsto V]$ denotes the obvious substitution (with e of a reference type). $\mathcal{M}/u = (\nu\tilde{l})(\xi,\sigma)$ if $\mathcal{M} = (\nu\tilde{l})(\xi\cdot u{:}V,\sigma)$; otherwise $\mathcal{M}/u = \mathcal{M}$. For $\mathcal{M}_{1,2}$ of the same type, $\mathcal{M}_1 \overset{[!x]}{\approx} \mathcal{M}_2$ iff $\forall V.(\mathcal{M}_1[x \mapsto V] \approx \mathcal{M}_2[x \mapsto V])$.

In the satisfaction of $\forall x^{\alpha}.C$ above, we consider the case the location is hidden. In $\forall \mathrm{X}.C$, we augment a model $\mathcal{M}$ with a map from type variables to closed types.

For evaluation formula, the defining clause says:

> *In any initial hypothetical state satisfying C evolvable from the current state, the application of e_1 to e_2 (both evaluated in the current state) terminates and the result z and the final state satisfy C'.*

Following [6, 18], we consider hypothetical initial state since a function can be invoked any time later, not only at the present state. The satisfaction of its generalised located assertion (which subsumes its finite counterpart):

$$\mathcal{M} \models \{C\}e\bullet e'{=}x\{C'\}@\{z\,|\,E(z)\}$$

iff it satisfies the clause of the evaluation formula above and the following, letting $\mathcal{M}_0 \stackrel{\text{def}}{=} (\nu\tilde{l})(\xi,\sigma_0)$ and $\mathcal{M}' \approx (\nu\tilde{l}\tilde{l}')(\xi,\sigma')$, $\forall\tilde{V}.((\nu\tilde{l})(\xi,\sigma_0[\tilde{l}_1 \mapsto \tilde{V}]) \approx (\nu\tilde{l}\tilde{l}')(\xi,\sigma'[\tilde{l}_1 \mapsto \tilde{V}]))$ where $l \in \{\tilde{l}_1\}$ iff $(\nu\tilde{l})(\xi\cdot z : l,\sigma_0) \models E$. This says:

> *The value stored at each location z satisfying $\neg E(z)$ in $\mathcal{M}_0$, is exactly preserved when the application at $\mathcal{M}_0$ results in $\mathcal{M}'$, taking $\mathcal{M}'$ up to $\approx$.*

For formal details, see [2, C.2].

C. Derivations for Examples in Section 5

This appendix lists the derivations omitted in Section 5.

C.1 Derivation for [*LetRef*]

We can derive [*LetRef*] as follows.

1. $\{C\}\ M :_m \{C_0\}$	(premise)
2. $\{C_0[!x/m] \wedge x\#\tilde{e}\}\ N :_u \{C'\}$ with $x \notin \mathsf{fpn}(\tilde{e})$	(premise)
3. $\{C\}\ \mathtt{ref}(M) :_x \{\#x.C_0[!x/m]\}$	(1,Ref)
4. $\{C\}\ \mathtt{ref}(M) :_x \{\#x.(C_0[!x/m] \wedge x\#\tilde{e})\}$	(Subs *n*-times)
5. $\{C\}\ \mathtt{ref}(M) :_x \{\nu y.(C_0[!x/m] \wedge x\#\tilde{e} \wedge x = y)\}$	(Conseq)
6. $\{C_0[!x/m] \wedge x\#\tilde{e} \wedge x = y\}\ N :_u \{C' \wedge x = y\}$	(2, Invariance)
7. $\{C\}\ \mathtt{let}\ x = \mathtt{ref}(M)\ \mathtt{in}\ N :_u \{\nu y.(C' \wedge x = y)\}$	(5,6,LetOpen)
8. $\{C\}\ \mathtt{let}\ x = \mathtt{ref}(M)\ \mathtt{in}\ N :_u \{\nu x.C'\}$	(Conseq)

Lines 5 and 8 use the standard logical law (discussed below). Lines 4 and 7 use the following derived/admissible proof rules:

$$[Subs]\ \frac{\{C\}\ M :_u \{C'\} \quad u \notin \mathsf{fpn}(e)}{\{C[e/i]\}\ M :_u \{C'[e/i]\}}$$

$$[LetOpen]\ \frac{\{C\}\ M :_x \{\nu\tilde{y}.C_0\} \quad \{C_0\}\ N :_u \{C'\}}{\{C\}\ \mathtt{let}\ x = M\ \mathtt{in}\ N :_u \{\nu\tilde{y}.C'\}}$$

[*LetOpen*] opens the "scope" of $\tilde{y}$ to N. The crucial step is Line 5, which turns freshness "#" into locality "ν" through the standard law of equality and existential, $C \equiv \exists y.(C \wedge x = y)$ with y fresh.

C.2 Derivation for `IncUnShared`

For illustration, we contrast the inference of `IncShared` with:

$$\mathtt{IncUnShared} \stackrel{\text{def}}{=} a{:=}\mathtt{Inc}; b{:=}\mathtt{Inc}; c_1{:=}(!a)(); c_2{:=}(!b)(); (!c_1{+}!c_2)$$

This program assigns to a and b two separate instances of `Inc`. This lack of sharing between a and b in `IncUnShared` is captured by the following derivation:

1.$\{\mathsf{T}\}\ \mathtt{Inc} :_m \{\nu x.\mathsf{inc}'(u,x,0)\}$
2.$\{\mathsf{T}\}\ a := \mathtt{Inc}\ \{\nu x.\mathsf{inc}'(!a,x,0)\}$
3.$\{\mathsf{inc}'(!a,x,0)\}\ b := \mathtt{Inc}\ \{\nu y.\mathsf{inc}''(0,0)\}$
4.$\{\mathsf{inc}''(0,0)\}\ c_1 := (!a)()\ \{\mathsf{inc}''(1,0) \wedge !c_1 = 1\}$
5.$\{\mathsf{inc}''(1,0)\}\ c_2 := (!b)()\ \{\mathsf{inc}''(1,1) \wedge !c_2 = 1\}$
6.$\{!c_1 = 1 \wedge !c_2 = 1\}\ (!c_1) + (!c_2) :_u \{u = 2\}$
7.$\{\mathsf{T}\}\ \mathtt{IncUnShared} :_u \{\nu xy.u = 2\}$
8.$\{\mathsf{T}\}\ \mathtt{IncUnShared} :_u \{u = 2\}$

Above $\mathsf{inc}''(n,m) = \mathsf{inc}'(!a,x,n) \wedge \mathsf{inc}'(!b,y,m) \wedge x \neq y$. Note $x \neq y$ is guaranteed by [*LetRef*]. This is in contrast to the derivation for `IncShared`, where, in Line 3, x is automatically shared after "$b := !a$" which leads to scope extrusion.

Figure 2 `mutualParity` derivations

1.	$\{(n \geq 1 \supset IsEven'(!y, gh, n-1, xy)) \ \wedge \ n = 0\}$ `f` $:_z \{z = Odd(n) \wedge !x = g \wedge !y = h\}@\emptyset$	(Const)
2.	$\{(n \geq 1 \supset IsEven'(!y, gh, n-1, xy)) \ \wedge \ n \geq 1\}$ `not`$((!y)(n-1)) :_z \{z = Odd(n) \wedge !x = g \wedge !y = h\}@\emptyset$	(Simple, App)
3.	$\{n \geq 1 \supset IsEven'(!y, gh, n-1, xy)\}$ `if` $n = 0$ `then f else not`$((!y)(n-1)) :_m \{z = Odd(n) \wedge !x = g \wedge !y = h\}@\emptyset$	(IfH)
4.	$\{\top\}\ \lambda n.$`if` $n = 0$ `then f else not`$((!y)(n-1)) :_u$ $\{\ \forall gh, n \geq 1.\{IsEven'(h, gh, n-1, xy)\}u \bullet n = z\{z = Odd(n) \wedge !x = g \wedge !y = h\}@\emptyset\}@\emptyset$	(Abs, $\forall$)
5.	$\{\top\}\ M_x :_u \{\ \forall gh, n \geq 1.(IsEven(h, gh, n-1, xy) \supset IsOdd(u, gh, n, xy))\}@\emptyset$	(Conseq)
6.	$\{\top\}\ x := M_x\{\ \forall gh, n \geq 1.(IsEven(h, gh, n-1, xy) \supset IsOdd(!x, gh, n, xy)) \wedge !x = g\}@x$	(Assign)
7.	$\{\top\}\ y := M_y\{\ \forall gh, n \geq 1.(IsOdd(g, gh, n-1, xy) \supset IsEven(!y, gh, n, xy)) \wedge !y = h\}@y$	(Similar with Line 6)
8.	$\{\top\}$ `mutualParity` $\{\forall gh.n \geq 1.((IsEven(h, gh, n-1, xy) \wedge IsOdd(g, gh, n-1, xy)) \supset (IsEven(!y, gh, n, xy) \wedge IsOdd(!x, gh, n, xy) \wedge !x = g \wedge !y = h)\ \}@xy$	($\wedge$-Post)
9.	$\{\top\}$ `mutualParity` $\{\forall n \geq 1 gh.((IsEven(h, gh, n-1, xy) \wedge IsOdd(g, gh, n-1, xy) \wedge !x = g \wedge !y = h) \supset (IsEven(!y, gh, n, xy) \wedge IsOdd(!x, gh, n, xy) \wedge !x = g \wedge !y = h)\}@xy$	(Conseq)
10.	$\{\top\}$ `mutualParity` $\{\forall n \geq 1 gh.((IsEven(!y, gh, n-1, xy) \wedge IsOdd(!x, gh, n-1, xy) \wedge !x = g \wedge !y = h) \supset (IsEven(!y, gh, n, xy) \wedge IsOdd(!x, gh, n, xy) \wedge !x = g \wedge !y = h)\}@xy$	(Conseq)
11.	$\{\top\}$ `mutualParity` $\{\forall n \geq 1.(\exists gh.(IsEven(!x, gh, n-1, xy) \wedge IsOdd(!y, gh, n-1, xy) \wedge !x = g \wedge !y = h) \supset \exists gh.(IsEven(!y, gh, n, xy) \wedge IsOdd(!x, gh, n, xy) \wedge !x = g \wedge !y = h)\}@xy$	(Conseq)
12.	$\{\top\}$ `mutualParity` $\{\exists gh.IsOddEven(gh, !x!y, xy, n)\}@xy$	

C.3 Derivation for `mutualParity` and `safeEven`

Let us define:

$$M_x \stackrel{\text{def}}{=} \lambda n.\texttt{if } y = 0 \texttt{ then f else not}((!y)(n-1))$$
$$M_y \stackrel{\text{def}}{=} \lambda n.\texttt{if } y = 0 \texttt{ then t else not}((!x)(n-1))$$

We also use:

$$IsOdd'(u, gh, n, xy) = IsOdd(u, gh, n, xy) \wedge !x = g \wedge !y = h$$
$$IsEven'(u, gh, n, xy) = IsEven(u, gh, n, xy) \wedge !x = g \wedge !y = h$$

We use the following derived rules and one standard structure rule appeared in [18].

$$[Simple] \quad \frac{-}{\{C[e/u]\}e :_u \{C\}}$$

$$[IfH] \quad \frac{\{C \wedge e\} M_1 :_u \{C'\} \quad \{C \wedge \neg e\} M_2 :_u \{C'\}}{\{C\}\ \texttt{if } e \texttt{ then } M_1 \texttt{ else } M_2 :_u \{C'\}}$$

$$[\wedge\text{-}Post] \quad \frac{\{C\}M :_u \{C_1\} \quad \{C\}M :_u \{C_2\}}{\{C\}M :_u \{C_1 \wedge C_2\}}$$

Figure 2 lists the derivation for `MutualParity`. In Line 5, we use the following axiom for the evaluation formula from [18]:

$$\{C \wedge A\}\ e_1 \bullet e_2 = z\{C'\} \quad \equiv \quad A \supset \{C\}e_1 \bullet e_2 = z\{C'\}$$

where A is stateless formula and we here set $A = IsEven(h, gh, n-1, xy)$. Line 9 is the standard logical implication $(\forall x.(C_1 \supset C_2) \supset (\exists x.C_1 \supset \exists x.C_2))$. Now we derive for `safeEven`. Let us define:

$$ValEven(u) = \forall n.\{\top\}u \bullet n = z\{z = Even(n)\}@\emptyset$$
$$C_0 = !x = g \wedge \ !y = h \ \wedge \ IsOdd(g, gh, n, xy)$$
$$Even_a = C_0 \ \wedge \ \forall n.\{C_0\}u \bullet n = z\{C_0\}@xy$$
$$Even_b = \forall n.\{C_0\}u \bullet n = z\{z = Even(n)\}@xy$$

The derivation is similar to `safeFact`.

1. $\{\top\}\lambda n.\texttt{t} :_m \{\top\}@\emptyset$
2. $\{\top\}$`mutualParity`$\,;\,!y :_u \{\exists gh.IsOddEven(gh, gu, xy, n)\}@xy$
3. $\{\top\}$`mutualParity`$\,;\,!y :_u \{\exists gh.(Even_a \ \wedge \ Even_b)\}@xy$
4. $\{xy\#ij\}$`mutualParity`$\,;\,!y :_u$ $\{\exists gh.(xy\#ij \ \wedge \ Even_a \ \wedge \ Even_b)\}@xy$
5. $\{\top\}$`safeEven` $:_u \{\nu\#xy\exists gh.(Even_a \ \wedge \ Even_b)\}@\emptyset$
6. $\{\top\}m \bullet () = u\{\nu\#xy\exists gh.(Even_a \wedge Even_b)\}$ $\supset \{\top\}m \bullet () = u\{ValEven(u)\}$ (by (AIH$_{A\exists}$))
7. $\{\top\}$`safeEven` $:_u \{ValEven(u)\}@\emptyset$

C.4 Derivation for Meyer-Seiber

For the derivation of (5.6) we use (ε is the empty string): $I = \mathsf{Inv}(f, Even(!x), x, \varepsilon, \varepsilon)$, $G_0 = \{Even(!x) \wedge x\#g\}g \bullet f\{Even(!x)\}$, and $G_1 = \{\top\}g \bullet f\{\top\}$. The derivation follows. Below $M_{1,2}$ is the body of the first/second lets, respectively.

1.$\{Even(!x) \wedge G_0\}\ gf\ \{Even(!x)\}$	(App)
2.$\{Even(!x) \wedge I \wedge G_1\}\ gf\ \{Even(!x)\}$	(1, Conseq)
3.$\{E \wedge [!x]C \wedge I \wedge x\#g\}\ gf\ \{C'\}@\tilde{w}x$	(App)
4.$\{E \wedge [!x]C \wedge I \wedge x\#g\}\ gf\ \{Even(!x) \wedge C'\}@\tilde{w}x$	(2, 3, Conj)
5.$\{Even(!x) \wedge C'\}$ $\texttt{if}$ $even(!x)$ $\texttt{then}$ $()$ $\texttt{else}$ $\Omega()$ $\{C'\}@\emptyset$	(If)
6.$\{E \wedge [!x]C \wedge I \wedge x\#g\}M_2\{C'\}@\tilde{w}x$	(4, 5, Seq)
7.$\{Even(!x)\}\lambda().x := !x+2 :_f \{I\}@\emptyset$	(Abs etc.)
8.$\{E \wedge [!x]C \wedge Even(!x) \wedge x\#g\}\ M_1\ \{C'\}@\tilde{w}x$	(7, 6, LetRef)
9.$\{E \wedge C\}\ 0 :_m\ \{E \wedge C \wedge Even(m)\}@\emptyset$	(Const)
10.$\{E \wedge C\}$ $\texttt{MeyerSieber}$ $\{C'\}@\tilde{w}$	(9, LetRef)

Line 2 uses the axiom in Proposition 9. Line 4 uses the standard structural rule. Line 10 cancels $[!x]$ from $[!x]C$ which is possible since m does not occur in C.

C.5 Derivation for Object

We need the following generalisation: The procedure u in (AIH) is of a function type $\alpha \Rightarrow \beta$: when values of other types such as $\alpha \times \beta$ or $\alpha + \beta$ are returned, we can make use of a generalisation. For simplicity we restrict our attention to the case when types do not contain recursive or reference types.

$$\begin{aligned}
\mathsf{Inv}(u^{\alpha\times\beta}, C_0, \tilde{x}, \tilde{r}, \tilde{w}) &= \wedge_{i=1,2}\mathsf{Inv}(\pi_i(u), C_0, \tilde{x}, \tilde{r}, \tilde{w}) \\
\mathsf{Inv}(u^{\alpha+\beta}, C_0, \tilde{x}, \tilde{r}, \tilde{w}) &= \wedge_{i=1,2}\forall y_i.(u = \mathsf{inj}_i(y_i) \supset \mathsf{Inv}(y_i, C_0, \tilde{x}, \tilde{r}, \tilde{w})) \\
\mathsf{Inv}(u^{\alpha}, C_0, \tilde{x}, \tilde{r}, \tilde{w}) &= \mathsf{T} \qquad (\alpha \in \{\mathsf{Unit}, \mathsf{Nat}, \mathsf{Bool}\})
\end{aligned}$$

Using this extension, we can generalise (AIH) so that the cancelling of C_0 is possible for all components of u. For example, if u is a pair of functions, those two functions need to satisfy the same condition as in (AIH). This is what we shall use for `cellGen`. We call the resulting generalised axiom ($\mathsf{AIH_c}$).

Let `cell` be the internal λ-abstraction of `cellGen`. First, it is easy to obtain:

$$\{\mathsf{T}\}\ \texttt{cell} :_o\ \{I_0\ \wedge\ G_1\ \wedge\ G_2\ \wedge\ E'\} \qquad \text{(C.1)}$$

where, with $I_0 = !x_0 = !x_1$ and $E' = !x_0 = z$.

$$\begin{aligned}
G_1 &= \{I_0\}\pi_1(o)\bullet() = v\{v = !x_0 \wedge I_0\}@\emptyset \\
G_2 &= \forall w.\{I_0\}\pi_1(o)\bullet w\{!x_0 = w \wedge I_0\}@x_0x_1
\end{aligned}$$

which will become, after taking off the invariant I_0:

$$\begin{aligned}
G_1' &= \{\mathsf{T}\}\pi_1(o)\bullet() = v\{v = !x_1\}@\emptyset \\
G_2' &= \forall w.\{\mathsf{T}\}\pi_1(o)\bullet w\{!x_0 = w\}@x_0.
\end{aligned}$$

Note I_0 is stateless except x_0. In G_1, notice the empty write set means $!x_1$ does not change from the pre to the postcondition. We now present the inference. We set $\texttt{cell}' \stackrel{\text{def}}{=} \texttt{let}\ y = \texttt{ref}(0)\ \texttt{in}\ \texttt{cell}$ below.

1.$\{\mathsf{T}\}$ $\texttt{cell} :_o$ $\{I_0 \wedge G_1 \wedge G_2 \wedge E'\}$	
2.$\{\mathsf{T}\}$ $\texttt{cell}' :_o$ $\{I_0 \wedge G_1 \wedge G_2 \wedge E'\}$	(LetRef)
3.$\{\mathsf{T}\}$ $\texttt{let}\ x_1 = z\ \texttt{in}\ \texttt{cell}' :_o$ $\{\nu\#x_1.(I_0 \wedge G_1 \wedge G_2) \wedge E'\}$	(LetRef)
4.$\{\mathsf{T}\}$ $\texttt{let}\ x_1 = z\ \texttt{in}\ \texttt{cell}' :_o$ $\{G_1' \wedge G_2' \wedge E'\}$	($\mathsf{AIH_c}$, ConsEval)
5.$\{\mathsf{T}\}$ $\texttt{let}\ x_{0,1} = z\ \texttt{in}\ \texttt{cell}' :_o$ $\{\nu\#x.(\ G_1' \wedge G_2' \wedge E'\ )\}$	(LetRef)
6.$\{\mathsf{T}\}$ $\texttt{cellGen} :_u$ $\{CellGen(u)\}$.	(Abs)

D. Algorithms for Dag and Graph

This appendix lists the programs for the dag copy and graph copy. The detailed derivation can be found in [2]. First we show the algorithm for the dag copy.

$$\begin{aligned}
\texttt{dagCopy}^{\alpha} &\stackrel{\text{def}}{=} \lambda g^{Tree(\alpha)}\texttt{let}\ x = \texttt{ref}(\emptyset)\ \texttt{in}\ \texttt{Main}\ g \\
\texttt{Main} &\stackrel{\text{def}}{=} \mu f.\lambda g.\texttt{if}\ \texttt{dom}(!x, g)\ \texttt{then}\ \texttt{get}(!x, g)\ \texttt{else} \\
&\qquad \texttt{case}\ !g\ \texttt{of} \\
&\qquad\quad \texttt{in}_1(n) : \texttt{new}(\texttt{inj}_1(n), g) \\
&\qquad\quad \texttt{in}_2(y_1, y_2) : \texttt{new}(\texttt{inj}_2(\langle fy_1, fy_2\rangle), g) \\
\texttt{new} &\stackrel{\text{def}}{=} \lambda(y, g).\texttt{let}\ g' = \texttt{ref}(y)\ \texttt{in}\ (x := \texttt{put}(!x, \langle g, g'\rangle); g')
\end{aligned}$$

When the program is called with the root of a dag, it first creates an empty table stored in a local variable x. The table remembers those nodes in the original dag which have already been processed, associating them with the corresponding nodes in the fresh dag. Before creating a new node, the program checks if the original node (say g) already exists in the table. If not, a new node (say g') is created, and x now stores the new table which adds a tuple $\langle g, g'\rangle$ to the original. The program assumes, for brevity, a predefined data type for a table (which in fact is realisable as, say, lists), with associated procedures. $\texttt{get}(t, g)$ to get the image of g in t; $\texttt{put}(t, \langle g, g'\rangle)$ to add a new tuple when g is not in the domain; $\texttt{dom}(t, g)$ and $\texttt{cod}(t, g)$ to judge if g is in the pre/post-image of t, as well as the constant $\emptyset$ for the empty table.

Next we present a copying algorithm which works with any graph of *Tree*-type, including those with circular edges.

$$\begin{aligned}
\texttt{graphCopy}^{\alpha} &\stackrel{\text{def}}{=} \lambda g^{Tree(\alpha)}.\texttt{let}\ x = \texttt{ref}(\emptyset)\ \texttt{in}\ \texttt{Main}\ g \\
\texttt{Main} &\stackrel{\text{def}}{=} \mu f.\lambda g.\texttt{if}\ \texttt{dom}(!x, g)\ \texttt{then}\ \texttt{get}(!x, g)\ \texttt{else} \\
&\qquad \texttt{case}\ !g\ \texttt{of} \\
&\qquad\quad \texttt{in}_1(n) : \texttt{new}(\texttt{inj}_1(n), g) \\
&\qquad\quad \texttt{in}_2(y_1, y_2) : \\
&\qquad\quad\quad \texttt{let}\ g' = \texttt{new}(\texttt{tmp}, g) \\
&\qquad\quad\quad \texttt{in}\ g' := \texttt{inj}_2(\langle fy_1, fy_2\rangle); g'
\end{aligned}$$

where $\texttt{tmp} = \texttt{inj}_1(0)$. $\texttt{graphCopy}^{\alpha}$ is essentially identical with $\texttt{dagCopy}^{\alpha}$ except when it processes a branch node, say g. Since its subgraphs can have a circular link to g or above, we should first register g and its corresponding fresh node, say g' (the latter with a temporary content), before processing two subgraphs.

Finally the polymorphic version of $\texttt{graphCopy}^{\alpha}$ is simply given by $\Lambda X.\texttt{graphCopy}^{X}$, using the standard universal type abstraction.