

Type Theory (15-814) Fall 2006

Homework 5: Classical Logic and Monadic Effects

William Lovas (wlovas@cs)

Out: Friday, December 1, 2006

Due: Thursday, December 7, 2006 (before 1:30 pm)

In this homework, you'll review proof terms for classical logic and explore the relation between implicit effects and monadic effects. This assignment is written-only—there is no programming component. As usual, you should submit your solution as `solution.pdf` by copying it to the directory

`/afs/cs.cmu.edu/academic/class/15814/handin/<userid>/hw5`

1 Classical logic

Classical logic represents a logical view of control effects, via the Curry-Howard isomorphism. Recall that in classical logic, we have propositions A , expressions M , continuations K , and states S .

$$\begin{aligned} A &::= \top \mid A \wedge B \mid \perp \mid A \vee B \mid A \supset B \mid \neg A \\ M &::= x \mid \langle \rangle \mid \langle M, N \rangle \mid \text{inl } M \mid \text{inr } M \mid \lambda x:A. M \mid \text{not } K \mid \mathbf{ccr}(u \div A. S) \\ K &::= u \mid \text{fst}; K \mid \text{snd}; K \mid \text{abort} \mid \{K, L\} \mid [M, K] \mid \text{not } M \mid \mathbf{ccp}(x:A. S) \\ S &::= K \# M \end{aligned}$$

Expressions M represent proofs of the truth of propositions A via the judgement $\Gamma; \Delta \vdash M : A$ (we also called expressions “assertions” or simply “proofs”) Continuations represent proofs of the falsehood of propositions A via the judgement $\Gamma; \Delta \vdash K \div A$ (we also called continuations “refutations”). States represent contradictions between a proof M and a refutation K of the same proposition A via the judgement $\Gamma; \Delta \vdash S$. Figure 1 shows the typing/proof rules for classical logic.

Computationally, proof terms for classical logic can be viewed as an abstract machine similar in spirit to the C machine from homework 4.

1.1 Elimination forms

As a type system, this language is different from ones we've seen before in that it doesn't have any elimination forms, just expressions and continuations. The elimination forms we're familiar with are all definable though, using the computation form $\mathbf{ccr}(u \div A. S)$, which corresponds roughly to `letcc` in the C machine, and logically to proof by contradiction. For example, we may define:

$$\mathbf{fst}(M : A \wedge B) : A := \mathbf{ccr}(u \div A. \text{fst}; u \# M)$$

Exercise 1. Define the following elimination forms using `ccr`:

1. $\mathbf{snd}(M : A \wedge B) : B := \dots$
2. $(M : A \supset B) (N : A) : B := \dots$
3. $\mathbf{case}(M : A \vee B) \{ \text{inl}(x:A) \Rightarrow N, \text{inr}(y:B) \Rightarrow P \} : C := \dots$

$\Gamma; \Delta \vdash M : A$		
$\frac{}{(\Gamma, x:A); \Delta \vdash x : A} \text{ (HYPT)}$	$\frac{\Gamma; (\Delta, u \div A) \vdash S}{\Gamma; \Delta \vdash \mathbf{ccr}(u \div A. S) : A} \text{ (#E-T)}$	$\frac{}{\Gamma; \Delta \vdash \langle \rangle : \top} \text{ (\top T)}$
$\frac{\Gamma; \Delta \vdash M : A \quad \Gamma; \Delta \vdash N : B}{\Gamma; \Delta \vdash \langle M, N \rangle : A \wedge B} \text{ (\wedge T)}$	$\frac{\Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash \mathbf{inl} M : A \vee B} \text{ (\vee T}_1\text{)}$	$\frac{\Gamma; \Delta \vdash M : B}{\Gamma; \Delta \vdash \mathbf{inr} M : A \vee B} \text{ (\vee T}_2\text{)}$
$\frac{\Gamma; \Delta \vdash K \div A}{\Gamma; \Delta \vdash \mathbf{not} K : \neg A} \text{ (\neg T)}$	$\frac{(\Gamma, x:A); \Delta \vdash M : B}{\Gamma; \Delta \vdash \lambda x:A. M : A \supset B} \text{ (\supset T)}$	
$\Gamma; \Delta \vdash K \div A$		
$\frac{}{\Gamma; (\Delta, u \div A) \vdash u \div A} \text{ (HYPF)}$	$\frac{(\Gamma, x:A); \Delta \vdash S}{\Gamma; \Delta \vdash \mathbf{ccp}(x:A. S) \div A} \text{ (#E-F)}$	$\frac{}{\Gamma; \Delta \vdash \mathbf{abort} \div \perp} \text{ (\perp F)}$
$\frac{\Gamma; \Delta \vdash K \div A}{\Gamma; \Delta \vdash \mathbf{fst}; K \div A \wedge B} \text{ (\wedge F}_1\text{)}$	$\frac{\Gamma; \Delta \vdash K \div B}{\Gamma; \Delta \vdash \mathbf{snd}; K \div A \wedge B} \text{ (\wedge F}_2\text{)}$	$\frac{\Gamma; \Delta \vdash K \div A \quad \Gamma; \Delta \vdash L \div B}{\Gamma; \Delta \vdash \{K, L\} \div A \vee B} \text{ (\vee F)}$
$\frac{\Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash \mathbf{not} M \div \neg A} \text{ (\neg F)}$	$\frac{\Gamma; \Delta \vdash M : A \quad \Gamma; \Delta \vdash K \div B}{\Gamma; \Delta \vdash [M, K] \div A \supset B} \text{ (\supset F)}$	
$\Gamma; \Delta \vdash S$		
$\frac{\Gamma; \Delta \vdash K \div A \quad \Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash K \# M} \text{ (#I)}$		

Figure 1: Classical logic proof terms

1.2 Classical Curry-Howard

The Curry-Howard isomorphism lets us interpret a program with a type as a proof of a proposition. For example, in the simply-typed λ -calculus we can prove one direction of the law of the contrapositive, $(A \supset B) \supset (B \supset \perp) \supset (A \supset \perp)$, as follows:

$$\lambda f:A \supset B. \lambda g:B \supset \perp. \lambda x:A. g (f x)$$

(Note: this proof uses the intuitionistic meaning of negation, defined as $A \supset \perp$; classical negation $\neg A$ is a primitive notion with its own rules; see Figure 1.)

Exercise 2. Prove the following classical theorems using classical proof terms:

1. Double-negation elimination: $\neg\neg A \supset A$.
2. Peirce's law: $((A \supset B) \supset A) \supset A$.

Cultural note: these theorems are equivalent to the law of the excluded middle, $A \vee \neg A$.

Exercise 3 (optional). Peirce's law, $((A \supset B) \supset A) \supset A$, is the type of **call/cc**, a primitive analogous to **letcc** in many languages with control effects like Scheme and SML/NJ. Can you explain your proof of Peirce's law in terms of its computational behavior?

2 Monadic effects

In class we saw two languages with storage effects: in one language effects were pervasive, and in the other, effects were segregated into a monad, τ comp, representing computations returning a value of type τ .

The language with implicit effects includes a type τ ref of references to values of type τ along with primitives **new** e , **get** e , and **set** (e_1, e_2) .

$$\begin{aligned} \tau ::= & \text{nat} \mid \tau_1 \rightarrow \tau_2 \mid \tau \text{ ref} \\ e ::= & x \mid \text{zero} \mid \text{succ } e \mid \mathbf{natcase } e \text{ of } \{ \text{zero} \Rightarrow e_0, \text{succ } x \Rightarrow e_1 \} \mid \lambda x:\tau. e \mid e_1 e_2 \\ & \mid \ell \mid \mathbf{new } e \mid \mathbf{get } e \mid \mathbf{set } (e_1, e_2) \end{aligned}$$

The typing rules for this language are shown in Figure 2. Its dynamic semantics consists of two judgements: $e@_\mu \mapsto e'@_\mu'$ and e value (not shown).

The monadic language distinguishes pure expressions e from effectful commands m .

$$\begin{aligned} \tau ::= & \text{nat} \mid \tau_1 \rightarrow \tau_2 \mid \tau \text{ ref} \mid \tau \text{ comp} \\ e ::= & x \mid \text{zero} \mid \text{succ } e \mid \mathbf{natcase } e \text{ of } \{ \text{zero} \Rightarrow e_0, \text{succ } x \Rightarrow e_1 \} \mid \lambda x:\tau. e \mid e_1 e_2 \mid \ell \mid \text{val } m \\ m ::= & \mathbf{return } e \mid i; x:\tau. m \mid \mathbf{let val } x : \tau = e \mathbf{ in } m \\ i ::= & \mathbf{new } e \mid \mathbf{get } e \mid \mathbf{set } (e_1, e_2) \end{aligned}$$

Any pure expression can be regarded as a trivial effectful command by **return**'ing it. Any effectful command m may be suspended as a pure expression $\text{val } m$. The command $i; x:\tau. m$ represents the sequential composition of instruction i with command m ; inside m , the result of executing i is bound to x . The command $\mathbf{let val } x : \tau = e \mathbf{ in } m$ runs a command that was suspended as an expression e ; as with instruction sequencing, the result of executing the suspended command is bound to x in the body m .

This language's complete typing rules are shown in Figure 3. Note that although we use the same metavariable e for expressions of the two languages, in the monadic language, we subscript the turnstile on the typing judgements with an m so that no confusion can result.

Its dynamic semantics consist of six judgements:

$$\begin{array}{lll} m@_\mu \mapsto m'@_\mu' & \mu \vdash e \mapsto e' & \mu \vdash i \mapsto i' \\ m@_\mu \text{ final} & \mu \vdash e \text{ value} & \mu \vdash i \text{ ready} \end{array}$$

$\Lambda; \Gamma \vdash e : \tau$

$$\begin{array}{c}
\frac{}{\Lambda; (\Gamma, x:\tau) \vdash x : \tau} \text{(VAR)} \qquad \frac{}{(\Lambda, x:\tau); \Gamma \vdash \ell : \tau \text{ ref}} \text{(LOC)} \\
\\
\frac{}{\Lambda; \Gamma \vdash \text{zero} : \tau} \text{(NAT-I-ZERO)} \qquad \frac{\Lambda; \Gamma \vdash e : \text{nat}}{\Lambda; \Gamma \vdash \text{succ } e : \text{nat}} \text{(NAT-I-SUCC)} \\
\\
\frac{\Lambda; \Gamma \vdash e : \text{nat} \quad \Lambda; \Gamma \vdash e_0 : \tau \quad \Lambda; (\Gamma, x:\text{nat}) \vdash e_1 : \tau}{\Lambda; \Gamma \vdash \text{natcase } e \text{ of } \{ \text{zero} \Rightarrow e_0, \text{succ } x \Rightarrow e_1 \} : \tau} \text{(NAT-E)} \\
\\
\frac{\Lambda; (\Gamma, x:\tau_1) \vdash e : \tau_2}{\Lambda; \Gamma \vdash \lambda x:\tau_1. e : \tau_1 \rightarrow \tau_2} (\rightarrow\text{-I}) \qquad \frac{\Lambda; \Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Lambda; \Gamma \vdash e_2 : \tau_2}{\Lambda; \Gamma \vdash e_1 e_2 : \tau} (\rightarrow\text{-E}) \\
\\
\frac{\Lambda; \Gamma \vdash e : \tau}{\Lambda; \Gamma \vdash \text{new } e : \tau \text{ ref}} \text{(REF-I)} \qquad \frac{\Lambda; \Gamma \vdash e : \tau \text{ ref}}{\Lambda; \Gamma \vdash \text{get } e : \tau} \text{(REF-E-GET)} \qquad \frac{\Lambda; \Gamma \vdash e_1 : \tau \text{ ref} \quad \Lambda; \Gamma \vdash e_2 : \tau}{\Lambda; \Gamma \vdash \text{set}(e_1, e_2) : \tau} \text{(REF-E-SET)}
\end{array}$$

Figure 2: Language with implicit effects

The expression judgements are standard, with the exception of suspended computations. These are judged to be values, since they do not dynamically evolve as pure expressions; they only run once invoked via a **let val** command.

$$\frac{}{\mu \vdash \text{val } m \text{ value}}$$

Recall also that locations are values provided they appear in the memory:

$$\frac{\ell \in \text{dom}(\mu)}{\mu \vdash \ell \text{ value}}$$

The other four judgements are summarized in Figure 4. They make use of a “leftist substitution”, $\langle m_1/x \rangle m_2$ which sequentially executes the command m_1 and substitutes its final value into m_2 ; leftist substitution is defined by induction over its left argument, the command being substituted:

$$\begin{aligned}
\langle i; y:\tau. m_1/x \rangle m_2 &= i; y:\tau. \langle m_1/x \rangle m_2 \\
\langle \text{let val } y : \tau = e \text{ in } m_1/x \rangle m_2 &= \text{let val } y : \tau = e \text{ in } \langle m_1/x \rangle m_2 \\
\langle \text{return } e/x \rangle m_2 &= [e/x] m_2
\end{aligned}$$

Safety for this language consists of six theorems relating the various static and dynamic semantics, three of which are Progress theorems and three of which are Preservation theorems.

Theorem (Progress for States). If $\mu : \Lambda$ and $\Lambda; \cdot \vdash_m m \sim \tau$, then either $m@_\mu$ final or $m@_\mu \mapsto m'@_{\mu'}$.

Theorem (Preservation for States). If $\mu : \Lambda$ and $\Lambda; \cdot \vdash_m m \sim \tau$ and $m@_\mu \mapsto m'@_{\mu'}$, then there exists a $\Lambda' \supseteq \Lambda$ such that $\mu' : \Lambda'$ and $\Lambda'; \cdot \vdash_m m' \sim \tau$.

Exercise 4. Complete the statement of safety for this language by stating Progress and Preservation for expressions and instructions.

$\Lambda; \Gamma \vdash_m e : \tau$	
$\frac{}{\Lambda; (\Gamma, x:\tau) \vdash_m x : \tau} \text{(VAR)}$	$\frac{}{(\Lambda, x:\tau); \Gamma \vdash_m \ell : \tau \text{ ref}} \text{(LOC)}$
$\frac{}{\Lambda; \Gamma \vdash_m \text{zero} : \tau} \text{(NAT-I-ZERO)}$	$\frac{\Lambda; \Gamma \vdash_m e : \text{nat}}{\Lambda; \Gamma \vdash_m \text{succ } e : \text{nat}} \text{(NAT-I-SUCC)}$
$\frac{\Lambda; \Gamma \vdash_m e : \text{nat} \quad \Lambda; \Gamma \vdash_m e_0 : \tau \quad \Lambda; \Gamma, x:\text{nat} \vdash_m e_1 : \tau}{\Lambda; \Gamma \vdash_m \text{natcase } e \text{ of } \{ \text{zero} \Rightarrow e_0, \text{succ } x \Rightarrow e_1 \} : \tau} \text{(NAT-E)}$	
$\frac{\Lambda; \Gamma, x:\tau_1 \vdash_m e : \tau_2}{\Lambda; \Gamma \vdash_m \lambda x:\tau_1. e : \tau_1 \rightarrow \tau_2} \text{(}\rightarrow\text{-I)}$	$\frac{\Lambda; \Gamma \vdash_m e_1 : \tau_2 \rightarrow \tau \quad \Lambda; \Gamma \vdash_m e_2 : \tau_2}{\Lambda; \Gamma \vdash_m e_1 e_2 : \tau} \text{(}\rightarrow\text{-E)}$
$\frac{\Lambda; \Gamma \vdash_m m \sim \tau}{\Lambda; \Gamma \vdash_m \text{val } m : \tau \text{ comp}} \text{(COMP-I)}$	

$\Lambda; \Gamma \vdash_m m \sim \tau$	
$\frac{\Lambda; \Gamma \vdash_m e : \tau}{\Lambda; \Gamma \vdash_m \text{return } e \sim \tau} \text{(RETURN)}$	$\frac{\Lambda; \Gamma \vdash_m i \sim \sigma \quad \Lambda; \Gamma, x:\sigma \vdash_m m \sim \tau}{\Lambda; \Gamma \vdash_m i; x:\sigma. m \sim \tau} \text{(SEQ)}$
$\frac{\Lambda; \Gamma \vdash_m e : \sigma \text{ comp} \quad \Lambda; \Gamma, x:\sigma \vdash_m m \sim \tau}{\Lambda; \Gamma \vdash_m \text{let val } x : \sigma = e \text{ in } m \sim \tau} \text{(COMP-E)}$	

$\Lambda; \Gamma \vdash_m i \sim \tau$	
$\frac{\Lambda; \Gamma \vdash_m e : \tau}{\Lambda; \Gamma \vdash_m \text{new } e \sim \tau \text{ ref}} \text{(REF-I)}$	$\frac{\Lambda; \Gamma \vdash_m e : \tau \text{ ref}}{\Lambda; \Gamma \vdash_m \text{get } e \sim \tau} \text{(REF-E-GET)}$
$\frac{\Lambda; \Gamma \vdash_m e_1 : \tau \text{ ref} \quad \Lambda; \Gamma \vdash_m e_2 : \tau}{\Lambda; \Gamma \vdash_m \text{set}(e_1, e_2) \sim \tau} \text{(REF-E-SET)}$	

Figure 3: Language with monadic effects: static semantics

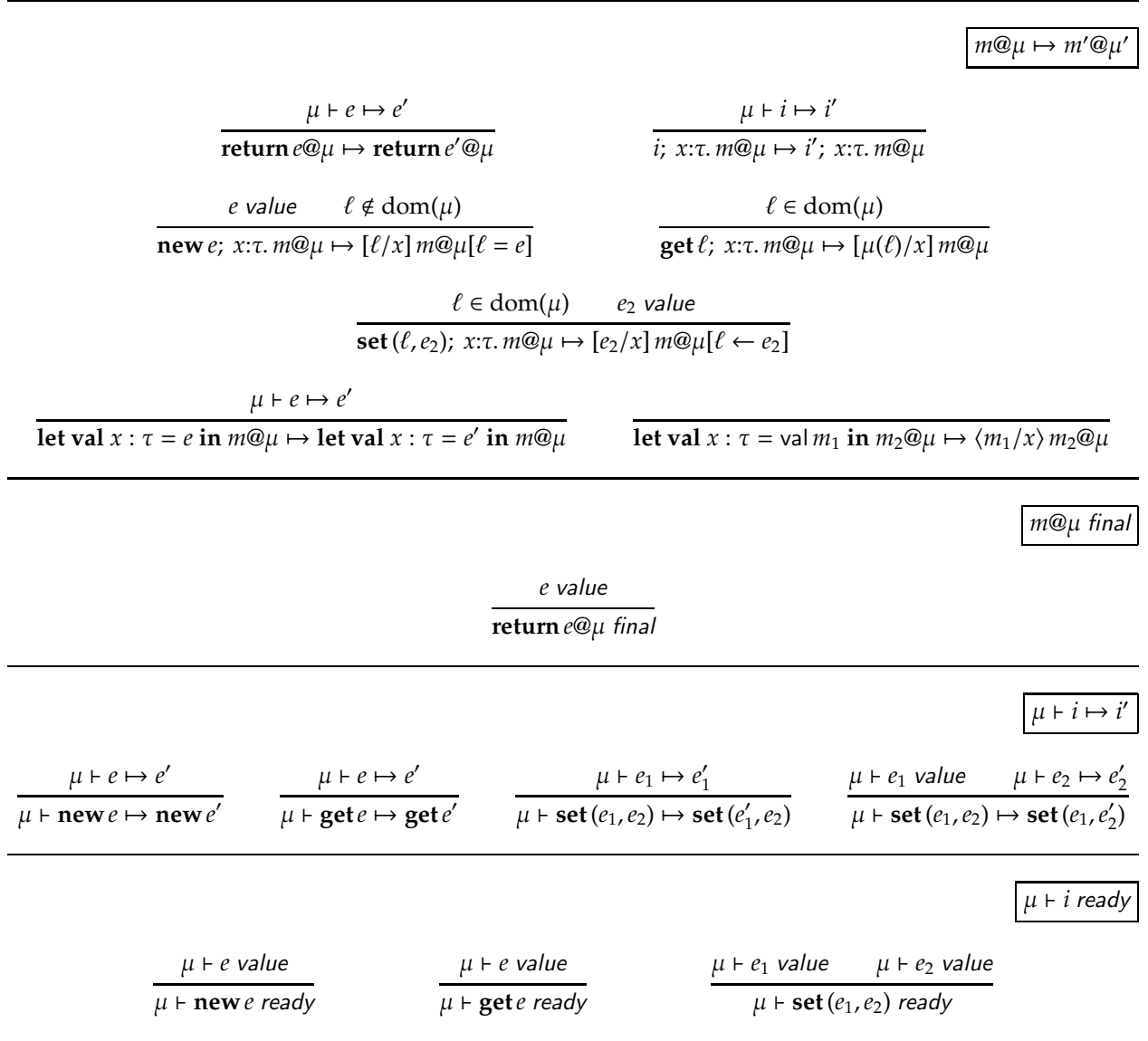


Figure 4: Language with monadic effects: dynamic semantics

Using the monadic language, we can write effectful programs, but instead of having type τ they have type $\tau \text{ comp}$. Consider the following program from the implicitly effectful language:

$$\begin{aligned} \text{incr} &: \text{nat ref} \rightarrow \text{nat} \\ &:= \lambda r:\text{nat ref}. \mathbf{set}(r, \text{succ}(\mathbf{get} r)) \end{aligned}$$

(Note: the return type is nat , since by convention we chose to let the return value of a \mathbf{set} expression be the value written to the reference.)

Exercise 5. Write $\text{incr} : \text{nat ref} \rightarrow \text{nat comp}$ in the monadic language by explicitly sequencing the effects.

Using references, we can simulate general recursion using backpatching. In the implicit language, we might write the factorial function as follows:

$$\begin{aligned} \text{fact} &:= \mathbf{let} fref = \mathbf{new}(\lambda x:\text{nat}. x) \mathbf{in} \\ &\quad \mathbf{set}(fref, \lambda x:\text{nat}. \mathbf{natcase} x \mathbf{of} \{ \text{zero} \Rightarrow 1, \text{succ } x' \Rightarrow \text{times } x((\mathbf{get} fref) x') \}) \end{aligned}$$

Exercise 6 (optional). Write a backpatching version of fact using the monadic language. (You may find this exercise useful in completing the following one.)

In general, we can translate expressions from the implicitly effectful language into commands of the monadic language in a type-preserving way. We must translate them to commands because in general, expressions in the implicitly effectful language might have effects. Suppose e is an expression from the effectful language; then e^* is its translation to the monadic language. The translation should satisfy the following theorem:

Theorem (Static correctness of $(\cdot)^*$). If $\Lambda; \Gamma \vdash e : \tau$, then $\Lambda^*; \Gamma^* \vdash_m e^* \sim \tau^*$.

The translation of contexts and store typings is just pointwise translation of the types contained therein. The translation of types is nearly the identity; it must however account for the fact that functions may have suspended effects in their bodies, so the codomain type is forced to be a computation:

$$\begin{aligned} \text{nat}^* &= \text{nat} \\ (\tau_1 \rightarrow \tau_2)^* &= \tau_1^* \rightarrow \tau_2^* \text{ comp} \\ (\tau \text{ ref})^* &= \tau^* \text{ ref} \end{aligned}$$

Exercise 7. Complete the following translation from implicit expressions e to monadic commands m . You should ensure—but need not prove—that each case of your translation satisfies the static correctness theorem above.

$$(e : \tau)^* = (m \sim \tau)$$

$$\begin{aligned}
x^* &= \mathbf{return} \ x \\
\mathbf{zero}^* &= \\
(\mathbf{succ} \ e)^* &= \\
(\mathbf{natcase} \ e \ \mathbf{of} \ \{ \mathbf{zero} \Rightarrow e_0, \mathbf{succ} \ x \Rightarrow e_1 \})^* &= \mathbf{let} \ \mathbf{val} \ n : \mathbf{nat} = \mathbf{val} \ (e^*) \ \mathbf{in} \\
&\quad \mathbf{let} \ \mathbf{val} \ res : \tau = \mathbf{natcase} \ n \ \mathbf{of} \ \{ \mathbf{zero} \Rightarrow \mathbf{val} \ (e_0^*), \mathbf{succ} \ x \Rightarrow \mathbf{val} \ (e_1^*) \} \ \mathbf{in} \\
&\quad \mathbf{return} \ res \\
(\lambda x : \tau. e)^* &= \\
(e_1 \ e_2)^* &= \\
\ell^* &= \\
(\mathbf{new} \ e)^* &= \\
(\mathbf{get} \ e)^* &= \\
(\mathbf{set} \ (e_1, e_2))^* &= \mathbf{let} \ \mathbf{val} \ r : \tau \ \mathbf{ref} = \mathbf{val} \ (e_1^*) \ \mathbf{in} \\
&\quad \mathbf{let} \ \mathbf{val} \ v : \tau = \mathbf{val} \ (e_2^*) \ \mathbf{in} \\
&\quad \mathbf{set} \ (r, v); \ v' : \tau. \\
&\quad \mathbf{return} \ v'
\end{aligned}$$