

# Final Examination Solutions

15-312: Principles of Programming Languages

May 10, 2007

Name:

Andrew ID:

## Instructions

- This exam is open-notes and open-book, but no computers are allowed.
- There are five problems, each worth 30 points for a total of 150 points. You have 3 hours in which to complete the exam.

	Ques 1	Ques 2	Ques 3	Ques 4	Ques 5	Total
Score						
Max	30	30	30	30	30	150
Grader						

# 1 Short Answer

**Problem 1** (5 points): Consider the following syntax of untyped  $\lambda$ -terms given as abstract binding trees:

$$u ::= x \mid \lambda(x.u) \mid \text{ap}(u_1, u_2) \quad (1)$$

Given an inductive definition of the judgement  $x$  free in  $u$  stating that the variable  $x$  occurs free in the term  $u$ . Do not assume that terms are identified up to  $\alpha$ -equivalence! Be sure to state any apartness conditions you may need.

$$\frac{}{x \text{ free in } x} \quad \frac{x \# y \quad x \text{ free in } u}{x \text{ free in } \lambda(y.u)} \quad \frac{x \text{ free in } u_1}{x \text{ free in } \text{ap}(u_1, u_2)} \quad \frac{x \text{ free in } u_2}{x \text{ free in } \text{ap}(u_1, u_2)}$$

**Problem 2** (5 points): If mutation is not required, the type of binary trees with values of type `nat` at each of the nodes is given by the recursive type

$$\mu(t.\mathbf{1} + (t \times \text{nat} \times t)).$$

The empty tree of this type is given by `fold(inl(⟨⟩))`, and a non-empty tree with children  $a$  and  $b$  and value  $v$  is given by `fold(inr(a, v, b))` (omitting type annotations for the sake of concision).

Suppose that we wish to support *in-place mutation* of trees, including the operation of deleting a subtree rooted at a node a given value,  $v$ , from the tree. Which of the following types are suitable for implementing this operation on trees? Please answer “yes” or “no” for each choice; it may or may not be appropriate to answer “yes” for more than one choice.

1.  $\mu(t.\mathbf{1} + (t \times \text{nat} \times t)) \text{ ref}$ :

Yes. We could functionally delete from the tree, then re-assign to the reference cell.

2.  $\mu(t.\mathbf{1} + ((t \times \text{nat} \times t) \text{ ref}))$ :

No. Does not permit a tree to become empty after deletion.

3.  $\mu(t.\mathbf{1} \text{ ref} + ((t \times \text{nat} \times t) \text{ ref}))$ :

No. Same problem as previous.

4.  $\mu(t.(\mathbf{1} + (t \times \text{nat} \times t)) \text{ ref})$ :

Yes. Can locally delete a node from the tree.

**Problem 3** (10 points): Many languages have *enumeration types* such as  $\{A, B, C\}$ , consisting of the given letters of the alphabet viewed as atomic symbols. The introduction forms for this type are the symbols `A`, `B`, and `C`. The elimination form is the expression

$$\text{case } e \{ A \Rightarrow e_A \mid B \Rightarrow e_B \mid C \Rightarrow e_C \}$$

with the obvious static and dynamic semantics.

Give an encoding of this type in System F (the polymorphic  $\lambda$ -calculus) by filling in the following equations:

1. Type definition:  $\{A, B, C\} =$

$$\forall t. t \rightarrow t \rightarrow t \rightarrow t$$

2. Introduction forms:

- (a)  $A =$

$$\Lambda t. \lambda a: t. \lambda b: t. \lambda c: t. a$$

- (b)  $B =$

$$\Lambda t. \lambda a: t. \lambda b: t. \lambda c: t. b$$

- (c)  $C =$

$$\Lambda t. \lambda a: t. \lambda b: t. \lambda c: t. c$$

3. Elimination form (case expression as above, of type  $\tau$ ):

$$e [\tau] e_A e_B e_C$$

**Problem 4** (10 points): Consider a *call-by-need* semantics for recursive types in which the recursive `fold` operator is to be evaluated lazily, so that (a) its argument is only evaluated when needed, and (b) its argument is evaluated at most once in any computation. Give the appropriate dynamic semantics rules for call-by-need evaluation of `fold` and `unfold`. You will require one rule for the judgement  $e$  value, and three transition rules.

$$\frac{}{\text{fold}(x) \text{ value}} \qquad \frac{x \# M}{(M, \text{fold}(e)) \mapsto (M[x = e], \text{fold}(x))}$$

$$\frac{}{(M, \text{unfold}(\text{fold}(x))) \mapsto (M, x)} \qquad \frac{(M, e) \mapsto (M', e')}{(M, \text{unfold}(e)) \mapsto (M', \text{unfold}(e'))}$$

## 2 Safety

The progress and preservation properties of a general transition system govern the individual steps of evaluation:

**Theorem 1** (Progress).

$$s \text{ ok} \supset s \text{ final} \vee (\exists s'. s \mapsto s')$$

**Theorem 2** (Preservation).

$$s \text{ ok} \wedge s \mapsto s' \supset s' \text{ ok}$$

In this question you are to extend these theorems to govern multiple steps of evaluation. Specifically, as a consequence of progress and preservation, one may prove that for any number of steps,  $n$ , and any state,  $s$ , either  $s$  steps to a state  $s'$  in  $n$  steps, or  $s$  steps to a value in fewer than  $n$  steps. This is stated formally by the following theorem:

**Theorem 3.**

$$\forall n. \forall s. s \text{ ok} \supset \exists e'. s \mapsto^n s' \vee (\exists n'. n' < n \wedge s \mapsto^{n'} s' \wedge s' \text{ final})$$

Here we have used the  $n$ -fold iteration of transition as defined in Chapter 5 of the textbook, which we repeat here for reference:

$$\frac{}{s \mapsto^0 s} \quad \frac{s \mapsto s'' \quad s'' \mapsto^n s'}{s \mapsto^{n+1} s'}$$

You are to give a proof of Theorem 3 by induction on  $n$ .

**Problem 1** (5 points): State the property,  $P$ , that is to be proved for each natural number  $n$  so that  $\forall n. P(n)$  is the safety theorem stated above. Hint: be sure to *explicitly state* any quantifiers that are required.

$$P(n) \equiv \forall s. s \text{ ok} \supset \exists e'. s \mapsto^n s' \vee (\exists n'. n' < n \wedge s \mapsto^{n'} s' \wedge s' \text{ final})$$

**Problem 2** (10 points): Prove  $P(0)$ , where  $P$  is the predicate you defined above

Given an arbitrary  $s$  assume  $s \text{ ok}$ . By the reflexive rule  $s \mapsto^0 s$ , thus  $\exists s'. s \mapsto^0 s' \vee (\exists n'. n' < n \wedge s \mapsto^{n'} s' \wedge s' \text{ final})$ .

**Problem 3** (15 points): Prove that  $P(n)$  implies  $P(n + 1)$  for the predicate  $P$  you defined above. To get you started, we have provided an outline as follows:

Let  $n$  be any natural number. Assuming  $P(n)$ , we are to deduce  $P(n + 1)$ . That is, given  $P(n)$  as inductive hypothesis, we are to show:

$$\forall s. s \text{ ok} \supset \exists e'. s \mapsto^{n+1} s' \vee (\exists n'. n' < n + 1 \wedge s \mapsto^{n'} s' \wedge s' \text{ final})$$

Assume \_\_\_\_\_ . By the progress theorem we have:

either  $s$  final or there exists an  $s'$  such that  $s \mapsto s'$ .

There are then two cases to consider, arising from the appeal to the progress theorem.

The first case is (fill in and complete the proof):

If  $s$  final then  $s \mapsto^0 s$ , which is enough for the result.

The second case states that (fill in):

$s \mapsto s'$  for some  $s'$ .

By the preservation theorem we have that

$s' \text{ ok}$

Therefore, by the inductive hypothesis,

$s' \mapsto^n s'' \vee (\exists n'. n' < n \wedge s' \mapsto^{n'} s'' \wedge s'' \text{ final})$  for some  $s''$ .

(Continued on the next page)

Proceed by cases based on preceding appeal to the inductive hypothesis to complete the proof.

In the first case we have  $s' \mapsto^n s''$ , and hence it follows from our assumptions that  $s \mapsto^{n+1} s''$ , which is enough for the result.

Otherwise we have  $n' < n$ ,  $s' \mapsto s''$  and  $s''$  final. It follows that  $n' + 1 < n + 1$  and  $s \mapsto^{n'+1} s''$  and  $s''$  final. Thus we have  $\exists s''. s \mapsto^{n+1} s'' \vee (\exists n'. n' < n + 1 \wedge s \mapsto^{n'} s'' \wedge s'' \text{ final})$  ie.  $P(n + 1)$ .

### 3 $\pi$ -calculus

The  $\pi$ -calculus is used to model interaction between concurrent processes (eg. protocols).

Here are two user scenarios for an ATM. In the next question you will model the interaction using  $\pi$ -calculus. Each user performs a sequence of actions:

- User 1, inserts her debit card, realizes that she has to go, pushes the cancel button, waits for her card and disappears.
- User 2, inserts his card, pushes the \$60 button, waits for the money and for his card and then disappears.

In the following questions you should use the following syntax for the synchronous  $\pi$ -calculus.

$$P ::= P|P \mid \text{await}(E) \mid * P \mid \nu c.P$$
$$E ::= E + E \mid ?\text{name}.P \mid !\text{name}.P \mid 0$$

**Problem 1** (10 points): Write three processes in the synchronous  $\pi$ -calculus representing User 1, User 2 and the ATM such that both User 1 and User 2 are able to interact with the ATM as described above. Make sure that the ATM process will work for any number of users, but may only have transaction at a time. This can be achieved with the  $*$  construct and a channel. You will need the channels

\$60 : To model the event of a push on the \$60 button.

cancel : To model the event of a push on the cancel button.

money : To model the event of receiving money.

card : To model the event of giving and receiving a card.

on : To model that there is precisely one ATM.

**User 1**(on, \$60, cancel, money, card) =

**User 2**(on, \$60, cancel, money, card) =

**ATM'** Hint: You will need the \* to model the ATM. It is helpful to name a process  $ATM'$  such that  $ATM = *ATM' \mid P$ , where you define  $P$ .

$ATM' = (\text{on}, \$60, \text{cancel}, \text{money}, \text{card}) =$

ATM (on, \$60, cancel, money, card) =

User 1:

$\text{await}(!\text{card}.\text{await}(!\text{cancel}.\text{await}(\text{?card}.\text{await}(0))))$

User 2:

$\text{await}(!\text{card}.\text{await}(!\$60.\text{await}(\text{?money}.\text{await}(\text{?card}.\text{await}(0))))))$

ATM' :

$\text{await}(\text{?on}.\text{await}(\text{?card}.\text{await}(\text{?cancel}.\text{await}(!\text{card}.\text{await}(!\text{on}.\text{await}(0))))+\text{?\$60}.\text{await}(!\text{money}.\text{await}(!\text{card}.\text{await}(!\text{on}.\text{await}(0))))))$

ATM :

$*\text{ATM}'|\text{await}(!\text{on}.\text{await}(0))$

System:

$\text{User 1}|\text{User 2}|\text{ATM}$

**Problem 2** (10 points): Give a process with the two users from above and four ATMs such that a user can interact with any of the four copies of the ATM.

System:

$\text{User 1}|\text{User 2}|\text{* ATM1}|\text{await}(!\text{on}.\text{await}(!\text{on}.\text{await}(!\text{on}.\text{await}(!\text{on}.\text{await}(0))))))$

In the following question you may use the *polyadic  $\pi$ -calculus*, in which we can send multiple channels in a single message.

$$P ::= P|P \mid \text{await}(E) \mid *P \mid \nu c.P$$
$$E ::= E + E \mid \text{?name}(c_1, \dots, c_n).P \mid !\text{name}\langle \text{name}, \dots, \text{name} \rangle.P \mid 0$$

The processes in the previous tasks may not be secure, in the sense that if we put a thief ( $\text{await}(\text{?money}.\text{await}(0))$ ) in parallel with the system then the thief may end up with the money and User 2 may end up stuck, waiting for some money that never arrives.

**Problem 3** (10 points): Create a new ATM and two new user processes behaving almost as above, but with a protocol that ensures that no intruder can disrupt a transaction with the ATM once it has begun. To ensure this, we introduce an additional, preliminary step in which the user obtains suitable credentials from the ATM with which to conduct the remainder of the transaction. The credentials consist of four channels that are transmitted simultaneously on a dedicated channel by the ATM so that a user process either obtains all or none of the channels. This ensures that no thief can disturb the integrity of a transaction.

ATM:

```
await(?on.vcard.vmoney.vcancel.v$60.!atm(card, money, cancel, $60).
      ATM'(on, card, money, cancel, $60))|await(!on.await(0))
```

User 1:

```
await(?atm(money, card, cancel, $60).User 1(on, $60, cancel, money, card))
```

User 2:

```
await(?atm(money, card, cancel, $60).User 2(on, $60, cancel, money, card))
```

## 4 Implicit Parallelism

In this question you are to consider an extension of PCF with natural numbers, products, and a type  $\tau \text{ vec}$  of vectors of type  $\tau$ . You may take as given arithmetic operations such as addition, multiplication, division, and remainder on natural numbers. However, the *only* operations on vectors are those given by the following chart:

$$\begin{aligned} \text{idx} &: \text{nat} \rightarrow \text{nat vec} \\ \text{rep} &: \text{nat} \rightarrow \tau \rightarrow \tau \text{ vec} \\ \text{sub} &: \tau \text{ vec} \rightarrow \text{nat} \rightarrow \tau \\ \text{permute} &: \text{nat vec} \rightarrow \tau \text{ vec} \rightarrow \tau \text{ vec} \\ \text{sum} &: \text{nat vec} \rightarrow \text{nat} \\ \text{map} &: (\sigma \rightarrow \tau) \rightarrow \sigma \text{ vec} \rightarrow \tau \text{ vec} \end{aligned}$$

These operations have the following meanings:

$\text{idx}(n)$	the vector $[0, \dots, n - 1]$
$\text{rep}(n)(x)$	the vector $[x, \dots, x]$ of length $n$
$\text{sub}(v)(n)$	$v[n]$ , the $n$ th element of $v$ , starting from 0
$\text{permute}(u)(v)$	the vector whose $i$ th element is $v[u[i]]$
$\text{sum}(v)$	the sum of the elements of the vector $v$
$\text{map}(f)(v)$	apply $f$ to each element of $v$

Any out of bounds accesses abort the program.

The *depth complexity* of each of these operations is given as follows. Each of the operations, except for  $\text{map}$ , is assigned a depth of 1; the depth of  $\text{map}(e_1)(e_2)$  is one more than the maximum of the depth of  $e_1$  and the depth of  $e_2$ . The depth of  $\text{map}(f)(v)$  is one more than the maximum of the depths of  $f$  and  $v$ .

An  $m \times n$  matrix consisting of  $m$  rows and  $n$  columns may be represented as a value of type  $\text{nat vec vec}$  consisting of an  $m$ -vector of  $n$ -vectors, a vector containing one vector for each

row of the matrix. For example, the matrix  $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$  is represented by the vector of vectors  $[[1, 2, 3], [4, 5, 6]]$ .

The *transpose* of an  $m \times n$  matrix may be defined using the operations above as follows:

$$\lambda X. \text{map } (\lambda j. \text{map } (\lambda r. \text{sub } (r, j)) X) (\text{idx } n)$$

This code computes the index vector  $[0, 1, \dots, n - 1]$ , one entry per column, over which we map the function that, when applied to column number  $j$ , maps another function over the rows of the matrix to extract the  $j$ th element of each row.

**Problem 1** (5 points): Determine the *asymptotic work complexity* of this algorithm as a function of  $m$  and  $n$ , assuming that the work complexity of `map` is linear in the size of the vector, `idx` is linear in its input, and that `sub` is of unit work complexity. You may use “big-O” notation in your answer.

The work is  $O(m \cdot n)$  since there is one unit of work for each element of the matrix.

**Problem 2** (5 points): Determine the *exact depth complexity* of this algorithm according to the cost assignments for the primitives given above. *Do not* use “big-O” notation.

The depth is 3, independently of  $m$  and  $n$ : the subscript has depth 1, the inner map depth 2, the index operation depth 1, the outer map depth 3.

**Problem 3** (10 points): Given the functions `dotp` to compute the dot product of two vectors, and the function `transpose` to compute the transpose of a matrix, define the product of an  $m \times n$  and an  $n \times p$  matrix. The result will, of course, have  $m$  rows and  $p$  columns. Your solution must be of *constant depth*.

```
λ X. λ Y. map (λ r. map (λ c. dotp x y) (transpose Y)) X
```

An alternative representation of an  $m \times n$  matrix is in *flattened* form consisting of an  $m \cdot n$ -vector of numbers, laid out row-after-row. Thus, the matrix  $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$  would be represented by

the vector  $[1, 2, 3, 4, 5, 6]$ , together with the dimensions  $m$  and  $n$  that determine the number of rows and columns. The  $j$ th element of the  $i$ th row lies at position  $i \cdot n + j$  in the flattened form. Put the other way around, the  $k$ th element of the flattened form represents the matrix element in row  $k \text{ div } n$  and column  $k \text{ mod } n$ .

**Problem 4** (10 points): Compute the *transpose* of the flattened form of an  $m \times n$  matrix. *Hint*: First compute a suitable permutation of the numbers  $[0, \dots, m \cdot n - 1]$  using the above calculations, then compute the flattened representation of the transpose by applying this permutation to the original. Remember that the transpose swaps the roles of the rows and columns of the matrix!

```
λ xs.  
  let f = (λ i. (i mod m) * n + (i div m)) in  
    permute (map f (idx (m*n))) xs
```

Alternative, using vectorized arithmetic:

```
λ xs.  
  let k = m * n in  
  let m' = repl k m in  
  let n' = repl k n in  
  let i' = idx k in  
  let p = add (mul (mod i' m') n') (i' m') in  
    permute p xs
```

## 5 Monads and Exceptions

In a call-by-name setting, reasoning about programs with exceptions has many of the same problems as reasoning about mutable storage. Recall the exception handling constructs from class,  $\text{handle}(e_1, x.e_2)$  and  $\text{raise}[\tau](e)$ . As a reminder we include typing rules for each below:

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{\text{exn}} \vdash e_2 : \tau}{\Gamma \vdash \text{handle}(e_1, x.e_2) : \tau} \quad \frac{\Gamma \vdash e : \tau_{\text{exn}}}{\Gamma \vdash \text{raise}[\tau](e) : \tau}$$

In the remainder of this problem, you may assume that there is some type  $\tau_{\text{exn}}$  along with two constructors,  $\text{toExn} : \text{nat} \rightarrow \tau_{\text{exn}}$  and  $\text{fromExn} : \tau_{\text{exn}} \rightarrow \text{nat}$  for injecting and extracting nats from that type.

**Problem 1** (5 points): Use  $\text{handle}$  and  $\text{raise}$  to give a program that gives a different result in call-by-name and call-by value languages. You may use any of the constructs from PCF, that is, functions,  $\text{fix}$ , and natural numbers, along with the just described exception operations.

$\text{handle}((\lambda x : \text{unit}.0) (\text{raise}[\text{unit}](\text{toExn}((1))))), x.e \text{ fromExn } x)$

The problem is that we must sequentialize effects in a call-by-name setting by treating exceptions as a form of effect using monads to enforce this. In the remainder of this problem you will work out the consequences of this.

In a monadic setting, we have two classes of terms — pure expressions,  $e$ , and computations  $m$ , along with a typing judgment for each form of term. The first was our usual expression typing judgment,  $\Gamma \vdash e : \tau$ , the second was a typing judgment for effectful computations:  $\Gamma \vdash m \sim \tau$ . Note that in this problem we won't be considering storage effects and hence do not need a store typing context as in the presentation of monads from class. Similarly there will be no store passed around in the dynamic semantics.

Consider the PCF language extended with the following syntax:

Types  $\tau ::= \dots \mid \tau_{\text{exn}} \mid \bigcirc \tau$

Expressions  $e ::= \dots \mid \text{comp}(m) \mid \text{toExn} \mid \text{fromExn}$

Computations  $m ::= \text{return}(e) \mid \text{raise}[\tau](e) \mid \text{letcomp}(e, x.m_1, y.m_2)$

Notice that  $\text{letcomp}$ , the monadic sequencing operation, has changed. It now has two possible branches. The first, which we will refer to as the success handler, is exactly what it was before: When  $e$ , gets down to  $\text{comp}(\text{return}(e))$  where  $e$  is a value, it substitutes  $e$  for  $x$  in the body of  $m_1$ . However, some computations can fail, that is evaluate down to  $\text{comp}(\text{raise}[\tau](e))$ . In these cases, we need to substitute the result of failure into the second handler, the failure handler.

Assume the following static semantics:

$$\frac{\Gamma \vdash e : \tau_{\text{exn}}}{\Gamma \vdash \text{raise}[\tau](e) \sim \tau} \quad \frac{\Gamma \vdash e : \bigcirc \tau' \quad \Gamma, x : \tau' \vdash m_1 \sim \tau \quad \Gamma, y : \tau_{\text{exn}} \vdash m_2 \sim \tau}{\Gamma \vdash \text{letcomp}(e, x.m_1, y.m_2) \sim \tau}$$

**Problem 2** (5 points): Give the definition of a judgment  $m$  final which defines when a computation is considered to be complete. This is roughly analogous to the definition of the value judgment for expressions.

$$\frac{}{\text{return}(e) \text{ final}} \quad \frac{}{\text{raise}[\tau](e) \text{ final}}$$

The dynamic semantics for this language are characterized by two judgments,  $e \mapsto e'$  and  $m \mapsto m'$ . The following are some of the cases of the dynamic semantics for `letcomp`, those corresponding to the dynamic semantics rules from the previous version `letcomp`.

$$\frac{e \mapsto e'}{\text{letcomp}(e, x.m_1, y.m_2) \mapsto \text{letcomp}(e', x.m_1, y.m_2)}$$

$$\frac{m \mapsto m'}{\text{letcomp}(\text{comp}(m), x.m_1, y.m_2) \mapsto \text{letcomp}(\text{comp}(m'), x.m_1, y.m_2)}$$

$$\frac{e \text{ value}}{\text{letcomp}(\text{comp}(\text{return}(e)), x.m_1, y.m_2) \mapsto [e/x]m_1}$$

**Problem 3** (10 points): Our new form of `letcomp` has more cases to consider than the old version. Give the remaining dynamic semantics rule for `letcomp` as informally described previously and the dynamic semantic rule for `raise` expressions. You should be guided by your definition of a final state.

$$\frac{e \text{ value}}{\text{letcomp}(\text{comp}(\text{raise}[\tau](e)), x.m_1, y.m_2) \mapsto [e/y]m_2}$$

$$\frac{e \mapsto e'}{\text{raise}[\tau](e) \mapsto \text{raise}[\tau](e')}$$

Encapsulating exceptions using monads results in similar costs and benefits as encapsulating state. In particular, previous functions that used exceptions must now be written in the monadic style.

Consider the following specification of the `half` function, which returns half of its input:

$$\begin{aligned} \text{half}(\text{zero}) &= \text{zero} \\ \text{half}(\text{succ}(\text{succ}(n))) &= \text{succ}(\text{half}(n)) \end{aligned}$$

Notice that this function is only defined on even numbers. An implementation of `half` should raise an exception when the input is odd. The content of that exception should be the input value. Thus `half(7)` should result in 7 being raised as an exception and `half(4)` is 2.

**Problem 4** (10 points): Fill in the code in the `half` function using monadic exceptions as described. The type of this expression will need to be  $\text{nat} \rightarrow \bigcirc \text{nat}$  as the input is a `nat` and it uses monadic operations to compute a result `nat`. You will use the constructs from PCF, `ifz` and `fix`, to handle the recursion. As a refresher, the typing rules for `ifz` and `fix` follow. Also remember that the variable

bound in the third part of the ifz expression represents the predecessor of the expression being examined.

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \text{fix}[\tau](x.e) : \tau} \quad \frac{\Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, x : \text{nat} \vdash e_2 : \tau}{\Gamma \vdash \text{ifz}(e, e_1, x.e_2) : \tau}$$

$\text{fix}[\text{nat} \rightarrow \bigcirc \text{nat}] \text{half}$  is

$\lambda x : \text{nat}.$

ifz  $x$  of

zero  $\Rightarrow$   $\text{comp}(\text{return}(\text{zero}))$

| succ( $x'$ )  $\Rightarrow$

ifz  $x'$  of

zero  $\Rightarrow$   $\text{comp}(\text{raise}[\text{nat}](\text{toExn}(x)))$

| succ( $x''$ )  $\Rightarrow$

comp(

letcomp  $r = \text{half } x''$

in  $\text{comp}(\text{return}(\text{succ}(r)))$

handle  $r \Rightarrow \text{comp}(\text{raise}[\text{nat}](\text{toExn}(\text{succ}(\text{succ}(\text{fromExn}(r))))))$ )