Name: Hemos ID:

CSE-321 Programming Languages 2010
Final — Sample Solution

	Prob 1	Prob 2	Prob 3	Prob 4	Prob 5	Prob 6	Total
Score							
Max	18	28	16	12	36	40	150

- There are six problems on 16 pages, including two work sheets, in this exam.
- The maximum score for this exam is 150 points, and there is an extracredit problem.
- Be sure to write your name and Hemos ID.
- You have three hours for this exam.

Instructor-Thank-Students-Problem [Extracredit]

State "Yes" if you attended all the lectures in this course, without missing a single lecture.

____

`PL 2010`$^\lambda$ [Extracredit]

State "Yes" if you wear the `PL 2010`$^\lambda$ T-shirt. ______________________

PL 2010 Tekken Match [Extracredit]

State "Yes" if you played in PL 2010 Tekken Match. ______________________

Who did you beat in PL 2010 Tekken Match? ______________________

1 Mutable references [18 pts]

Consider the following simply-typed λ-calculus extended with mutable references.

$$
\begin{array}{rrcl}
\textsf{type} & A & ::= & P \mid A \rightarrow A \mid \textsf{int} \mid \textsf{ref}\ A \\
\textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\,e \mid e\ e \mid \textsf{let}\ x = e\ \textsf{in}\ e \mid \textsf{ref}\ e \mid\ !e \mid e := e \mid 0 \mid 1 \mid \cdots \\
\textsf{value} & v & ::= & \lambda x\!:\!A.\,e \mid l \mid 0 \mid 1 \mid \cdots \\
\textsf{store} & \psi & ::= & \cdot \mid \psi, l \mapsto v \\
\textsf{typing context} & \Gamma & :;= & \cdot \mid \Gamma, x : A \\
\textsf{store typing context} & \Psi & ::= & \cdot \mid \Psi, l \mapsto A
\end{array}
$$

Question 1. **[8 pts]** We want to represent an array of integers as a function taking an index (of type int) and returning a corresponding elements of the array. We choose a functional representation of arrays by defining type iarray for arrays of integers as follows:

$$\textsf{iarray} = \textsf{ref}\ (\textsf{int} \rightarrow \textsf{int})$$

We need the following constructs for arrays:

- $\textsf{new} : \textsf{unit} \rightarrow \textsf{iarray}$ for creating a new array.
  $\textsf{new}\ ()$ returns a new array of indefinite size; all elements are initialized as 0.
- $\textsf{access} : \textsf{iarray} \rightarrow \textsf{int} \rightarrow \textsf{int}$ for accessing an array.
  $\textsf{access}\ a\ i$ returns the i-th element of array a.
- $\textsf{update} : \textsf{iarray} \rightarrow \textsf{int} \rightarrow \textsf{int} \rightarrow \textsf{unit}$ for updating an array.
  $\textsf{update}\ a\ i\ n$ updates the i-the element of array a with integer n.

Exploit the constructs for mutable references to implement new, access and update. Fill in the blank:

$$
\begin{array}{rcl}
\textsf{new} & = & \lambda\_\!:\!\textsf{unit}.\ \textsf{ref}\ \lambda i\!:\!\textsf{int}.\,0 \\[1em]
\textsf{access} & = & \lambda a\!:\!\textsf{iarray}.\,\lambda i\!:\!\textsf{int}.\,(!a)\ i \\[1em]
\textsf{update} & = & \lambda a\!:\!\textsf{iarray}.\,\lambda i\!:\!\textsf{int}.\,\lambda n\!:\!\textsf{int}.
\end{array}
$$

$\underline{\textsf{let}\ old =\ !a\ \textsf{in}\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$

$\underline{a := \lambda j\!:\!\textsf{int}.\,\textsf{if}\ i = j\ \textsf{then}\ n\ \textsf{else}\ old\ j\qquad\qquad\qquad\qquad}$

Question 2. [10 pts] State progress and type preservation theorems. In your statements, use the following judgments:

- A typing judgment $\Gamma \mid \Psi \vdash e : A$ means that expression e has type A under typing context Γ and store typing context Ψ.
- A reduction judgment $e \mid \psi \mapsto e' \mid \psi'$ means that e with store ψ reduces to e' with ψ'.
- A store judgment $\psi :: \Psi$ means that Ψ corresponds to ψ.

Theorem (Progress). *Suppose that expression e satisfies $\cdot \mid \Psi \vdash e : A$ for some store typing context Ψ and type A. Then either:*

(1) e is a value, or

(2) for any store ψ such that $\psi :: \Psi$,

there exist some expression e' and store ψ' such that $e \mid \psi \mapsto e' \mid \psi'$.

Theorem (Type preservation). *Suppose* $\begin{cases} \Gamma \mid \Psi \vdash e : A \\ \psi :: \Psi \\ e \mid \psi \mapsto e' \mid \psi' \end{cases}$.

Then there exists a store typing context Ψ' such that $\begin{cases} \Gamma \mid \Psi' \vdash e' : A \\ \Psi \subset \Psi' \\ \psi' :: \Psi' \end{cases}$.

2 Evaluation context and environment [28 pts]

Consider the following fragment of the simply-typed λ-calculus.

$$
\begin{array}{rrcl}
\textsf{type} & A & ::= & P \mid A \to A \\
\textsf{base type} & P & ::= & \textsf{bool} \\
\textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\ e \mid \textsf{true} \mid \textsf{false} \mid \textsf{if}\ e\ \textsf{then}\ e\ \textsf{else}\ e
\end{array}
$$

Question 1. [5 pts] Give the definition of evaluation contexts for the call-by-value strategy.

$$\textsf{evaluation context} \qquad \kappa \ ::= \ \square \mid \kappa\ e \mid (\lambda x\!:\!A.\, e)\ \kappa \mid \textsf{if}\ \kappa\ \textsf{then}\ e\ \textsf{else}\ e$$

Question 2. [5 pts] Give the definition of evaluation contexts for the call-by-name strategy.

$$\textsf{evaluation context} \qquad \kappa \ ::= \ \square \mid \kappa\ e \mid \textsf{if}\ \kappa\ \textsf{then}\ e\ \textsf{else}\ e$$

Question 3. [5 pts] Under the call-by-value strategy, give an expression e such that

- $e = \kappa[\![e']\!]$ where e' is the redex, and
- e reduces to e_0 that is decomposed to $\kappa[\![e'']\!]$ where e'' is the redex for the next reduction.

$$(\lambda y\!:\!\textsf{bool}.\, (\lambda x\!:\!\textsf{bool}.\, x)\ y)\ \textsf{true}$$

Question 4. [5 pts] Under the call-by-value strategy, give an expression e such that

- $e = \kappa[\![e']\!]$ where e' is the redex, and
- e reduces to e_0 that is decomposed to $\kappa'[\![e'']\!]$ where e'' is the redex for the next reduction and $\kappa \neq \kappa'$.

$$(\lambda x\!:\!\textsf{bool}\to\textsf{bool}.\, x)\ (\lambda y\!:\!\textsf{bool}.\, y)\ \textsf{true}$$

Question 5. [8 pts] The key idea behind the environment semantics is to postpone a substitution $[v/x]e$ by storing a pair of value v and variable x in an *environment*. We use the following definition of environment:

$$\textsf{environment} \qquad \eta \quad ::= \quad \cdot \mid \eta, x \hookrightarrow v$$

$\cdot$ denotes an empty environment, and $x \hookrightarrow v$ means that variable x is to be replaced by value v. We use an *environment evaluation judgment* of the form $\eta \vdash e \hookrightarrow v$:

$$\eta \vdash e \hookrightarrow v \qquad \Leftrightarrow \qquad e \textit{ evaluates to } v \textit{ under environment } \eta$$

Give the definition of values for the simply-typed λ-calculus given in the beginning of this section.

$$\textsf{value} \qquad v \quad ::= \quad [\eta, \lambda x\!:\!A.\, e] \mid \textsf{true} \mid \textsf{false}$$

Complete the following three rules for the environment evaluation judgment $\eta \vdash e \hookrightarrow v$ corresponding to the call-by-value strategy.

$$\frac{x \hookrightarrow v \in \eta}{\eta \vdash x \hookrightarrow v}$$

$$\frac{}{\eta \vdash \lambda x\!:\!A.\, e \hookrightarrow [\eta, \lambda x\!:\!A.\, e]}$$

$$\frac{\eta \vdash e_1 \hookrightarrow [\eta', \lambda x\!:\!A.\, e] \quad \eta \vdash e_2 \hookrightarrow v' \quad \eta', x \hookrightarrow v' \vdash e \hookrightarrow v}{\eta \vdash e_1\ e_2 \hookrightarrow v}$$

3 Subtyping [16 pts]

Question 1. **[6 pts]** Complete subtyping rules for function and reference types.

$$\frac{A' \leq A \quad B \leq B'}{A \to B \leq A' \to B'}\ \mathit{Fun}_{\leq}$$

$$\frac{A \leq B \quad B \leq A}{\mathsf{ref}\ A \leq \mathsf{ref}\ B}\ \mathit{Ref}_{\leq}$$

Question 2. **[10 pts]** The Java language adopts the following subtyping rule for array types:

$$\frac{A \leq B}{\mathsf{array}\ A \leq \mathsf{array}\ B}\ \mathit{Array}_{\leq}{}'$$

While it is controversial whether the rule $\mathit{Array}_{\leq}{}'$ is a flaw in the design of the Java language, using the rule $\mathit{Array}_{\leq}{}'$ for subtyping on array types incurs a runtime overhead which would otherwise be unnecessary. State specifically when and why such runtime overhead occurs in terms of dynamic tag-checks which inspect type information of each object at runtime. You may write in Korean.

Answer: Whenever a value of type B is written to an array of type $\mathsf{array}\ A$, the runtime system must verify a subtyping relation $B \leq A$, which incurs a runtime overhead of dynamic tag-checks.

4 Recursive types [12 pts]

Consider the following simply-typed λ-calculus extended with recursive types:

$$\begin{array}{rrcl} \textsf{type} & A & ::= & \textsf{unit} \mid A \rightarrow A \mid A + A \mid \alpha \mid \mu\alpha.A \\ \textsf{expression} & e & ::= & x \mid \lambda x : A.\, e \mid e\ e \mid \\ & & & \textsf{inl}_A\ e \mid \textsf{inr}_A\ e \mid \textsf{case}\ e\ \textsf{of inl}\ x.\, e \mid \textsf{inr}\ y.\, e \mid \\ & & & \textsf{fold}_C\ e \mid \textsf{unfold}_C\ e \\ \textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, \alpha\ \textsf{type} \end{array}$$

Question 1. [**6 pts**] Give typing rules for $\textsf{fold}_C\ e$ and $\textsf{unfold}_C\ e$.

$$\frac{C = \mu\alpha.A \quad \Gamma \vdash e : [C/\alpha]A \quad \Gamma \vdash C\ \textsf{type}}{\Gamma \vdash \textsf{fold}_C\ e : C}\ \textsf{Fold}$$

$$\frac{C = \mu\alpha.A \quad \Gamma \vdash e : C}{\Gamma \vdash \textsf{unfold}_C\ e : [C/\alpha]A}\ \textsf{Unfold}$$

Question 2. [**6 pts**] Consider the following recursive datatype for natural numbers:

$$\textsf{datatype}\ \texttt{nat} \quad = \quad \texttt{Zero} \mid \texttt{Succ}\ \textsf{of}\ \texttt{nat}$$

Using a recursive type, we encode type `nat` as $\mu\alpha.\textsf{unit}+\alpha$. Encode `Zero` and `Succ` e.

$$\texttt{Zero} \quad = \quad \textsf{fold}_{\texttt{nat}}\ \textsf{inl}_{\texttt{nat}}\ ()$$

$$\texttt{Succ}\ e \quad = \quad \textsf{fold}_{\texttt{nat}}\ \textsf{inr}_{\textsf{unit}}\ e$$

Question 3. [Extracredit] We want to translate an expression e in the untyped λ-calculus into an expression e° in the simply typed λ-calculus extended with recursive types. We treat all expressions in the untyped λ-calculus alike by assigning a unique type Ω (*i.e.*, e° is to have type Ω). If every expression is assigned type Ω, we may think that $\lambda x.\, e$ is assigned type $\Omega \rightarrow \Omega$ *as well as* type Ω. Or, in order for $e_1\ e_2$ to be assigned type Ω, e_1 must be assigned *not only* type Ω but also type $\Omega \rightarrow \Omega$ because e_2 is assigned type Ω. Thus Ω must be identified with $\Omega \rightarrow \Omega$.

Use recursive types and their constructs to complete the definition of Ω and e°. Fill in the blank:

$$\begin{array}{rcl}
\Omega & = & \mu\alpha.\alpha \rightarrow \alpha \\
\\
x^\circ & = & x \\
\\
(\lambda x.\, e)^\circ & = & \mathsf{fold}_\Omega\ \lambda x : \Omega.\, e^\circ \\
\\
(e_1\ e_2)^\circ & = & (\mathsf{unfold}_\Omega\ e_1{}^\circ)\ e_2{}^\circ
\end{array}$$

5 Polymorphism (36 pts)

The following shows the abstract syntax for System F:

$$
\begin{array}{rrcl}
\textsf{type} & A & ::= & A \rightarrow A \mid \alpha \mid \forall\alpha.A \\
\textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\,e \mid e\ e \mid \Lambda\alpha.\,e \mid e\,[\![A]\!]
\end{array}
$$

Below we define an *erasure* function $erase(\cdot)$ which takes an expression in System F and erases all type annotations in it to produce a corresponding expression in untyped λ-calculus:

$$
\begin{array}{lcl}
erase(x) & = & x \\
erase(\lambda x\!:\!A.\,e) & = & \lambda x.\,erase(e) \\
erase(e_1\ e_2) & = & erase(e_1)\ erase(e_2) \\
erase(\Lambda\alpha.\,e) & = & erase(e) \\
erase(e\,[\![A]\!]) & = & erase(e)
\end{array}
$$

Question 1. **[5 pts]** Give a well-typed closed expression e in System F such that $erase(e) = \lambda x.\,x\ x$. If there is no such expression, state so.

$$\lambda x\!:\!\forall\alpha.\alpha\rightarrow\alpha.\,x\,[\![\forall\alpha.\alpha\rightarrow\alpha]\!]\ x$$

Question 2. **[5 pts]** Give a well-typed closed expression e in System F such that $erase(e) = (\lambda x.\,x\ x)\ (\lambda x.\,x\ x)$. If there is no such expression, state so.

Answer: There is no such expression.

Question 3. **[6 pts]** A Church numeral $\widehat{n}$ takes a function f and returns another function f^n which applies f exactly n times. In order for f^n to be well-typed, its argument type and return type must be identical. Hence we define the base type $\textsf{nat}$ in System F as follows:

$$\textsf{nat} \quad = \quad \forall\alpha.(\alpha\rightarrow\alpha)\rightarrow(\alpha\rightarrow\alpha)$$

Encode a zero $\textsf{zero}$ of type $\textsf{nat}$ and a successor function $\textsf{succ}$ of type $\textsf{nat}\rightarrow\textsf{nat}$:

$$\textsf{zero} \quad = \quad \Lambda\alpha.\,\lambda f\!:\!\alpha\rightarrow\alpha.\,\lambda x\!:\!\alpha.\,x$$

$$\textsf{succ} \quad = \quad \lambda n\!:\!\textsf{nat}.\,\Lambda\alpha.\,\lambda f\!:\!\alpha\rightarrow\alpha.\,\lambda x\!:\!\alpha.\,(n\,[\![\alpha]\!]\ f)\ (f\ x)$$

The following shows the abstract syntax for the let-polymorphism system:

$$\begin{array}{rrcl} \textsf{monotype} & A & ::= & A \rightarrow A \mid \alpha \\ \textsf{polytype} & U & ::= & A \mid \forall \alpha.U \\ \textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\ e \mid \Lambda \alpha.\, e \mid e\, [\![A]\!] \mid \textsf{let}\ x\!:\!U = e\ \textsf{in}\ e \end{array}$$

Below we define an erasure function $erase(\cdot)$ which takes an expression in the let-polymorphism system and erases all type annotations in it to produce a corresponding expression in the implicit let-polymorphism system:

$$\begin{array}{lcl} erase(x) & = & x \\ erase(\lambda x\!:\!A.\, e) & = & \lambda x.\, erase(e) \\ erase(e_1\ e_2) & = & erase(e_1)\ erase(e_2) \\ erase(\Lambda \alpha.\, e) & = & erase(e) \\ erase(e\, [\![A]\!]) & = & erase(e) \\ erase(\textsf{let}\ x\!:\!U = e\ \textsf{in}\ e') & = & \textsf{let}\ x = erase(e)\ \textsf{in}\ erase(e') \end{array}$$

Question 4. [5 pts] Give a well-typed closed expression e in the let-polymorphism system such that $erase(e) = \textsf{let}\ f = \lambda x.\, x\ \textsf{in}\ (f\ \textsf{true}, f\ 0)$. Assume two monotypes $\textsf{bool}$ for boolean values and $\textsf{int}$ for integers.

$$\textsf{let}\ f\!:\!\forall \alpha.\alpha \rightarrow \alpha = \Lambda \alpha.\, \lambda x\!:\!\alpha.\, x\ \textsf{in}\ (f\, [\![\textsf{bool}]\!]\ \textsf{true}, f\, [\![\textsf{int}]\!]\ 0))$$

Question 5. [10 pts] Explain value restriction. You may write in Korean.

Answer: Value restriction allows variable x in a let-binding $\textsf{let}\ x = e\ \textsf{in}\ e'$ to be assigned a polytype only if expression e is a value.

Question 6. [5 pts] Give a well-typed closed expression e in the let-polymorphism system with value restriction such that $erase(e) = \textsf{let}\ f = (\lambda x.\, x)\ (\lambda y.\, y)\ \textsf{in}\ (f\ \textsf{true}, f\ 1)$. If there is no such expression, explain why. You may write in Korean.

Answer: Due to the value restriction, f cannot have a polytype such as $\forall \alpha.\alpha \rightarrow \alpha$. Therefore f cannot be applied to two values $\textsf{true}$ and 1 of different types at the same time.

6 Type reconstruction [40 pts]

Consider the implicit let-polymorphic type system given in the Course Notes.

$$\begin{array}{rrcl}
\textsf{monotype} & A & ::= & A \rightarrow A \mid \alpha \\
\textsf{polytype} & U & ::= & A \mid \forall \alpha . U \\
\textsf{expression} & e & ::= & x \mid \lambda x.\, e \mid e\ e \mid \textsf{let}\ x = e\ \textsf{in}\ e \\
\textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : U \\
\textsf{type substitution} & S & ::= & \mathrm{id} \mid \{A/\alpha\} \mid S \circ S \\
\textsf{type equations} & E & ::= & \cdot \mid E, A = A
\end{array}$$

- $S \cdot U$ and $S \cdot \Gamma$ denote applications of S to U and Γ, respectively.
- $\mathit{ftv}(\Gamma)$ denotes the set of free type variables in Γ; $\mathit{ftv}(U)$ denotes the set of free type variables in U.
- We write $\Gamma + x : U$ for $\Gamma - \{x : U'\}, x : U$ if $x : U' \in \Gamma$, and for $\Gamma, x : U$ if Γ contains no type binding for variable x.

Question 1. [6 pts] The type reconstruction algorithm, called $\mathcal{W}$, takes a typing context Γ and an expression e as input, and returns a pair of a type substitution S and a monotype A as output. State the soundness theorem of the algorithm $\mathcal{W}$. Use the typing judgment of the form $\Gamma \triangleright x : U$ in your statement.

(Soundness of $\mathcal{W}$). If $\mathcal{W}(\Gamma, e) = (S, A)$,

then $S \cdot \Gamma \triangleright e : A$

Question 2. [14 pts] Assume that you are given the unification algorithm $\mathsf{Unify}(E)$ and an auxiliary function $\mathsf{Gen}_\Gamma(A)$ which generalizes monotype A to a polytype after taking into account free type variables in typing context Γ. The following specifies the algorithm $\mathcal{W}$. Complete the specification:

$$
\begin{array}{rcll}
\mathcal{W}(\Gamma, x) & = & (\mathrm{id}, \{\vec{\beta}/\vec{\alpha}\} \cdot A) & x : \forall\vec{\alpha}.A \in \Gamma \text{ and fresh } \vec{\beta} \\
\mathcal{W}(\Gamma, \lambda x.\, e) & = & \text{let } (S, A) = \mathcal{W}(\Gamma + x : \alpha, e) \text{ in} & \text{fresh } \alpha \\
 & & (S, (S \cdot \alpha) \rightarrow A) & \\
\mathcal{W}(\Gamma, e_1\ e_2) & = & \text{let } (S_1, A_1) = \mathcal{W}(\Gamma, e_1) \text{ in} & \\
 & & \text{let } (S_2, A_2) = \mathcal{W}(S_1 \cdot \Gamma, e_2) \text{ in} & \\
 & & \text{let } S_3 = \mathsf{Unify}(S_2 \cdot A_1 = A_2 \rightarrow \alpha) \text{ in} & \text{fresh } \alpha \\
 & & (S_3 \circ S_2 \circ S_1, S_3 \cdot \alpha) & \\
\mathcal{W}(\Gamma, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) & = & \text{let } (S_1, A_1) = \mathcal{W}(\Gamma, e_1) \text{ in} & \\
 & & \text{let } (S_2, A_2) = \mathcal{W}(S_1 \cdot \Gamma + x : \mathsf{Gen}_{S_1 \cdot \Gamma}(A_1), e_2) \text{ in} & \\
 & & (S_2 \circ S_1, A_2) &
\end{array}
$$

Question 3. **[6 pts]** Given an application $e_1\ e_2$, the algorithm $\mathcal{W}$ reconstructs first the type of e_1 and then the type of e_2. Modify the algorithm $\mathcal{W}$ so that it reconstructs first the type of e_2 and then the type of e_1.

$$
\begin{array}{rcl}
\mathcal{W}(\Gamma, e_1\ e_2) & = & \text{let } (S_2, A_2) = \mathcal{W}(\Gamma, e_2) \text{ in} \\
 & & \text{let } (S_1, A_1) = \mathcal{W}(S_2 \cdot \Gamma, e_1) \text{ in} \\
 & & \text{let } S_3 = \mathsf{Unify}(A_1 = (S_1 \cdot A_2) \rightarrow \alpha) \text{ in} \quad \text{fresh } \alpha \\
 & & (S_3 \circ S_1 \circ S_2, S_3 \cdot \alpha)
\end{array}
$$

Question 4. **[6 pts]** Now we add a product type $A_1 \times A_2$ and an untyped pair construct (e_1, e_2). The typing rule for (e_1, e_2) is as follows:

$$
\frac{\Gamma \triangleright e_1 : A_1 \quad \Gamma \triangleright e_2 : A_2}{\Gamma \triangleright (e_1, e_2) : A_1 \times A_2} \times\mathsf{I}
$$

Complete the case for (e_1, e_2) in the algorithm $\mathcal{W}$:

$$
\begin{array}{rcl}
\mathcal{W}(\Gamma, (e_1, e_2)) & = & \text{let } (S_1, A_1) = \mathcal{W}(\Gamma, e_1) \text{ in} \\
 & & \text{let } (S_2, A_2) = \mathcal{W}(S_1 \cdot \Gamma, e_2) \text{ in} \\
 & & (S_2 \circ S_1, (S_2 \cdot A_1) \times A_2)
\end{array}
$$

Question 5. **[8 pts]** What is the result of $\mathcal{W}(\cdot, \mathsf{let}\ f = \lambda x.\, x\ \mathsf{in}\ (f\ 0, f\ \mathsf{true}))$? Assume two monotypes bool for boolean values and int for integers.

$$\begin{array}{rcl} \text{Substitution} & = & \{\mathsf{int}/\alpha_4\} \circ \{\mathsf{int}/\alpha_3\} \circ \{\mathsf{bool}/\alpha_2\} \circ \{\mathsf{bool}/\alpha_1\} \\ \\ \text{Type} & = & \mathsf{int} \times \mathsf{bool} \end{array}$$

For each type variable in the resultant type substitution, indicate where it is produced.

1. α_1 is generated when the algorithm $\mathcal{W}$ specializes the polymorphic type of f for $f\ 0$;
2. α_2 when $\mathcal{W}$ reconstructs the type of $f\ 0$;
3. α_3 when $\mathcal{W}$ specializes the polymorphic type of f for $f\ \mathsf{true}$; and
4. α_4 when $\mathcal{W}$ reconstructs the type of $f\ \mathsf{true}$.

Work sheet

Work sheet