Type Abstraction Rules for References: A Comparison of Four Which Have Achieved Notoriety

James William O'Toole Jr. *

Abstract

I present four type abstraction rules which have been introduced by various authors to permit polymorphic type safety in the presence of mutable data. Each of the type abstraction rules is discussed in the context of the language in which is was introduced, and the various abstraction rules are compared.

Categories and Subject Descriptions: D3.1 [**Programming Languages**] – Formal Definitions and Theory; D3.3 [**Programming Languages**] – Language Constructs: *Implicit Typing*; D1.m [**Programming Techniques**] – Miscellaneous: *Polymorphic References*;

General Terms: Languages, Type Theory, Polymorphism, Reference Values.
Additional Key Words and Phrases: type systems, polymorphic references, mutation, effect systems, type inference, type reconstruction, imperative types, weak polymorphism, Standard ML, FX-89.

*National Science Foundation Graduate Fellow.
This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-83-K-0125.
Authors' address: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139.
E-mail: james@zermatt.lcs.mit.edu

1 Type Abstraction Rules

The type abstraction rules I consider here are:

1. FX-89-pure: expression abstracted must be pure
2. Tofte-applicative: one-level store types
3. Damas-III: two-level store types
4. MacQueen-weak: type variables have strength

2 FX-89-pure

Attaches specific side-effect information to all function arrows and enforces the correctness of these effect specifications. The expression which is abstracted with respect to a type variable must have no (immediate) side-effects.

This ought to make it a very restrictive rule, as compared to the others. (Aside from the fact that I expect the checking of the side-effect specifications to disallow more programs.) However, inserting an explicit type abstraction at the appropriate point within the expression might alleviate the problem.

2.1 FX-89 Language Syntax

ι	: I	::=	Identifiers		
π	: P	::=	Primitive types		
υ	: U	::=	P	primitive type	
			I	type identifier	
			U $\rightarrow$ U	function	
e	: E	::=	I		variable
			(lambda (I) E)		lambda
			(E E)		application
			(let (I E) E)		generic-let

The type domain U contains the types which are supplied by the programmer in explicit type declarations. The type of a function encodes the type of its argument and its result. If the type of the argument is monomorphic, then it may be omitted. The type $\forall t.v$ represents the type of polymorphic values abstracted over the type parameter t.

In the expression domain, `lambda` abstracts `E` over the ordinary variable `I`.

2.2 Deductive System

I present the typing system of IFX as as formal deduction system consisting of a set of type reconstruction rules. The type system contains *generic* (i.e. general) type variables, and distinguishes between these generic type variables and the type identifiers which appear in user-supplied types. The type system also distinguishes between monomorphic and polymorphic types:

α	: `G`	::=	General type variables	
μ	: `M`	::=	`P`	primitive type
			`I`	type identifier
			`G`	general type variable
			`M` $\rightarrow$ `M`	function
τ	: `T`	::=	`P`	primitive type
			`I`	type identifier
			`G`	general type variable
			`T` $\rightarrow$ `T`	function
			$\forall$ `I`$_1 \ldots$ `I`$_n$ `T`	polymorphic type

The IFX typing rules make use of an important distinction between the M and T type domains. The rules are designed so that M types may be omitted from formal argument type declarations, but T types may not. Thus, the different levels in the type syntax specify the restrictions on the input programs. The use of syntactically-specified restrictions is intended to communicate clearly to the programmer the limitations of the type reconstruction system.

Type Schemes

The IFX type system supports the generic polymorphism found in ML, as well as the explicit polymorphism found in Reynolds' second-order polymorphic lambda calculus. In order to provide generic polymorphism, *type schemes* are defined which represent the *generic* (i.e. general) type of a variable which is permitted multiple *instantiations*:

Definition (Type Scheme). A *type scheme* η is a term of the form

$$\tilde{\forall}\alpha_1 \ldots \alpha_n.\ \tau,$$

where $\alpha_1 \ldots \alpha_n$ are the *generic variables* of $\tau \in$ T.

The symbols $\tilde{\forall}$ and $\forall$ are distinguished deliberately: $\tilde{\forall}$ binds the generic type variables of a type scheme, and $\forall$ binds type variables within a type.

Definition (Alpha-renaming). Types τ and τ' are alpha-renamable (written $\tau \simeq \tau'$) iff some renaming of type variables bound in τ produces τ'.

Definition (Instantiation). The type τ' is an instance of the scheme $\eta = \tilde{\forall}\alpha_1 \ldots \alpha_n.\ \tau$ (written $\eta \succeq \tau'$) iff there are monomorphic $\mu_1 \ldots \mu_n$ such that $\tau[\mu_i/\alpha_i] \simeq \tau'$. (The $\succeq$ relation extends to type schemes by $\eta \succeq \eta'$ iff $\forall\ \tau : \eta' \succeq \tau \Rightarrow \eta \succeq \tau$.)

Note that only M types may be substituted to produce instantiations, and that it is assumed that substitution takes place with renaming of any bound type variables to avoid capture. The result of substituting μ for t in τ will be written $\tau[\mu/t]$. The type scheme $\eta = \tilde{\forall}.\tau$, having no generic type variables, will occasionally be abbreviated as τ.

The inference rules for explicitly typed terms are presented first. A type assignment A maps each variable in its domain to a type scheme. The notation A_x refers to the type assignment A with the assignment for variable x removed.

The notation FGV(τ) refers to the free *general type variables* of τ, and FTV(τ) to refer to the free *type identifiers* of τ. Similarly, FGV(A) refers to the free *general type variables* of the type schemes in the assignment A. Also, Gen(A, τ) is defined as follows:

Definition (Generalization). The *generalization* of τ with respect to A (written Gen(A, τ)), is the type scheme $\eta = \tilde{\forall}\alpha_i.\ \tau$, where $\{\alpha_i\}$ = FGV(τ) $-$ FGV(A).

Typing Rules

The type reconstruction rules of IFX are as follows:

ILAMBDA

$$\frac{A_x + (x : \mu) \vdash e : \tau}{A \vdash \texttt{(lambda (}x\texttt{)}\ e\texttt{)} : \mu \rightarrow \tau}$$

APPL

$$\frac{A \vdash e : \tau_a \rightarrow \tau_r \qquad A \vdash e_a : \tau_a}{A \vdash (e\ e_a) : \tau_r}$$

The above rules describe the typing requirements of value abstraction and value application.

The following rules describe the typing requirements of variables and the ML-style generic `let` construct.

VARINST

$$\frac{(x : \tilde{\forall}\alpha_i.\,\tau) \in A}{A \vdash x : \tau[\mu_i/\alpha_i]}$$

ILET

$$\frac{A \vdash e_b : \tau_b \qquad A \vdash e_b : \tau_b' \Rightarrow \mathrm{Gen}(A, \tau_b) \succeq \mathrm{Gen}(A, \tau_b') \qquad A_x + (x : \mathrm{Gen}(A, \tau_b)) \vdash e : \tau}{A \vdash (\mathtt{let}\ (x\ e_b)\ e) : \tau}$$

Generic `let`

The ILET and VARINST rules provide the ML-style generic `let`. ILET associates a generic type scheme with the `let`-bound variable, and VARINST permits each occurence of the variable to be independently assigned any instance of its generic type scheme. The convenience of automatic generalization and instantiation are provided by these two rules. In IFX, the typing rules permit this convenience with the caveat that the automatically deduced type parameters be M-types.

The typing power of the ILET rule is equivalent to that provided by rewriting the `let` expression in the usual way, while making use of `open` and `close`:

$$(\mathtt{(lambda}\ (x : \tau)\ e[\mathtt{(open}\ x)/x])\ \mathtt{(close}\ e_b)).$$

However, this transformation is not pure syntactic sugar, because it requires τ, the explicitly polymorphic type of the bound variable.

3 Tofte-applicative

Contaminates all type variables appearing in any type expression at which the `ref` constructor is instantiated. The contaminated type variables are imperative, the others are applicative. This distinction is maintained by type abstraction, and is enforced at function call boundaries, etc. The abstraction rule does not permit abstractions of expansive expressions with respect to imperative type variables; expansive expressions are let and application expressions.

3.1 Definitions

The typing system distinguishes between *imperative* and *applicative* type variables:

$$t \in AppTyVar$$
$$u \in ImpTyVar$$
$$\alpha \in TyVar = AppTyVar \cup ImpTyVar$$

τ : M	::=	P	primitive type
		G	type variable
		M $\rightarrow$ M	function
		$\tau\ ref$	reference type

Definition (Type Closure). The *type closure* of τ with respect to A (written $\mathrm{Clos}_A\,\tau$), is the type scheme $\eta = \tilde{\forall}\alpha_i.\,\tau$, where $\{\alpha_i\} = \mathrm{tyvars}\ \tau - \mathrm{tyvars}\ A$.

Definition (Applicative Type Closure). The *applicative type closure* of τ with respect to A (written $\mathrm{AppClos}_A\,\tau$), is the type scheme $\eta = \tilde{\forall}\alpha_i.\,\tau$, where $\{\alpha_i\} = \mathrm{apptyvars}\ \tau - \mathrm{apptyvars}\ A$.

When a type scheme is instantiated, only imperative types may be substituted for imperative type variables. An expression is considered to be *expansive* if its evaluation might expand the domain of the store (i.e., allocate mutable data). The classification adopted in [Tofte87] is that `let` expressions and applications are expansive, but lambda abstractions and variable accesses are not.

3.2 Typing rules

The reference creation operator `ref` is assigned the imperative type $\forall u.\, u \rightarrow u\ ref$. The rules which provide type abstraction of expansive and non-expansive expressions in the imperative/applicative system are as follows:

VARINST

$$\frac{(x : \tilde{\forall}\alpha_i.\,\tau) \in A}{A \vdash x : \tau[\mu_i/\alpha_i]}$$

LET-Expansive

$$\frac{A \vdash e_b : \tau_b \qquad e_b \text{ is expansive.} \qquad A_x + (x : \mathrm{AppClos}_A\,\tau_b) \vdash e : \tau}{A \vdash (\mathtt{let}\ (x\ e_b)\ e) : \tau}$$

LET-Non-expansive

$$\frac{A \vdash e_b : \tau_b \quad e_b \text{ is non-expansive.} \quad A_x + (x : \text{Clos}_A\, \tau_b) \vdash e : \tau}{A \vdash \texttt{(let (}x\ e_b\texttt{)}\ e\texttt{)} : \tau}$$

3.3 Applicative Types and FX-89

In FX-89, type abstraction is permitted only when the side-effect specifications ensure that the polymorphic expression is referentially transparent. [Tofte87] takes a different approach, based on the concept of *applicative* types. Tofte classifies certain expressions as *expansive*, and permits type abstraction of these expressions only with respect to *applicative* type variables. This type abstraction rule permits different type abstractions than does the FX-89 `pure-type-abstraction` rule, as I will show later. Perhaps the imperative typing discipline can be combined with the type reconstruction system of FX-89.

4 Damas-III

Maintains a two level version of imperative types, distinguishing those type variables which have been contaminated already from those which will become contaminated by further application. The deductions carry a set of type variables, and so also do any type schemes which are arrows. Types, however, do not carry sets of type variables, nor is there more than a single top-level such set in a type scheme.

4.1 Definitions

The typing system defines schemes to include a set of type variables:

$\alpha \in TyVar$

τ	: T	::=	P	primitive type
			G	type variable
			T $\rightarrow$ T	function
			$\tau\ ref$	reference type
η	: S	::=	T	type
			T $\rightarrow$ T $* \Delta$	impure function
			$\forall\ \alpha . \eta$	polymorphic type

Definition (Type Closure). The *type closure* of η with respect to type assignment A and type variables Δ (written $\text{DClos}_{A\,\Delta}\,\eta$), is the type scheme $\eta' = \forall\ \alpha_i . \eta$, where $\{\alpha_i\} = \text{tyvars}\ \eta - (\text{tyvars}\ A\ \text{tyvars}\ \Delta$

When a type scheme is instantiated, the substitution is used to expand the set of type variables, and the set of type variables may be spuriously expanded as well.

4.2 Typing rules

The reference creation operator `ref` is assigned the imperative type $\forall\ t . t \rightarrow t\ ref * \{t\}$. The rules which provide type abstraction and instantiation are as follows:

DVAR-INST

$$\frac{(x : \forall\ \alpha_i\, \tau) \in A}{A \vdash x : \tau[\mu_i/\alpha_i] * \phi}$$

DLET

$$\frac{A \vdash e_b : \eta_b * \Delta \quad A_x + (x : \text{DClos}_A\, \eta_b) \vdash e : \eta * \Delta}{A \vdash \texttt{(let (}x\ e_b\texttt{)}\ e\texttt{)} : \eta * \Delta}$$

The rules which describe the typing requirements of value abstraction and value application are as follows:

DLAMBDA

$$\frac{A_x + (x : \tau_a) \vdash e : \tau * \Delta}{A \vdash \texttt{(lambda (}x\texttt{)}\ e\texttt{)} : (\tau_a \rightarrow \tau * \Delta) * \phi}$$

DAPPL

$$\frac{A \vdash e : (\tau_a \rightarrow \tau_r) * \Delta \quad A \vdash e_a : \tau_a * \Delta}{A \vdash (e\ e_a) : \tau_r * \Delta}$$

5 MacQueen-weak

Attaches numbers to type variables which measure their "weakness" (strength). The numbers indicate how many applications must take place before a reference to the type variable might have been created. Abstraction is permitted only with respect to type variables whose weakness remains positive. Weakness is downward contaminating, and the reference constructor is the source of contamination. A further restriction is not yet well understood: An instantiation of a let-bound variable is strength-limited somehow, related to the outermost abstraction level at which the expression of which it is part appears as an operand. Better figure this out.

5.1 Definitions

The typing system distinguishes between *imperative* and *applicative* type variables:

$$
\begin{aligned}
w &\in Strength = \{\ldots, -1, 0, 1, \ldots, \infty\} \\
\alpha &\in Tyvar \\
\alpha^w &\in WeakTyVar = TyVar \times Strength
\end{aligned}
$$

τ : M	::=	P	primitive type
		G	type variable
		$\mathtt{M} \rightarrow \mathtt{M}$	function
		$\tau\ ref$	reference type

Definition (Strength Limit). The type τ is *w-strength limited*, written $[\tau]^w$, iff all type variables in τ with non-infinite strength have strength less than or equal to w.

Definition (Strengthening). The *strengthening* of τ, written $[\tau]^{++}$, is the type in which all type variables with non-infinite strength have incrementally larger strength. So $[\tau_a \rightarrow \tau_b]^{++} \equiv [\tau_a]^{++} \rightarrow [\tau_b]^{++}$, and $[\alpha^\infty]^{++} \equiv \alpha^\infty$, but $[\alpha^w]^{++} \equiv \alpha^{w+1}$.

Definition (Weak Type Closure). The *weak type closure* of τ with respect to A (written $\mathrm{WeakClos}_A\, \tau$), is the type scheme $\eta = \tilde{\forall}\alpha_i^{w_i}.\,\tau$, where $\{\alpha_i\} = \mathrm{tyvars}\ \tau - \mathrm{tyvars}\ A$, and $w_i = min\{w \mid \alpha_i^w \in \mathrm{tyvars}\ \tau\}$.

When a type scheme is instantiated, the type substituted for a type variable must not be stronger than the type variable.

Weak Typing Rules

The type reconstruction rules of McQueen-weak are as follows:

WLAMBDA

$$\frac{A_x + (x : \mu) \vdash e : \tau}{A \vdash \mathtt{(lambda}\ (x)\ e\mathtt{)} : [\mu \rightarrow \tau]^{++}}$$

WAPPL

$$\frac{A \vdash e : [\tau_a \rightarrow \tau_r]^{++} \quad A \vdash e_a : \tau_a \quad [\tau_a]^0}{A \vdash (e\ e_a) : \tau_r}$$

The above rules describe the typing requirements of value abstraction and value application.

The following rules describe the typing requirements of variables and the ML-style generic `let` construct.

WINST

$$\frac{(x : \tilde{\forall}\alpha_i^{w_i}.\,\tau) \in A \quad [\tau_i]^{w_i}}{A \vdash x : \tau[\tau_i/\alpha_i]}$$

WLET

$$\frac{A \vdash e_b : \tau_b \quad A_x + (x : \mathrm{WeakClos}_A\, \tau_b) \vdash e : \tau}{A \vdash \mathtt{(let}\ (x\ e_b)\ e\mathtt{)} : \tau}$$

The reference value constructor `ref` is assigned the type $\forall\,\alpha^1.\,\alpha^1 \rightarrow \alpha^1\ ref$.

6 Comparison of Abstraction Rules

6.1 Damas-III > Tofte-applicative

```
(1)    let f = let x = (fn x => x) 1
                in (fn y => !(ref y))
                end
        in (f 1; f true)
        end
```

[Tofte87] provides this example on page 73.

Damas-III can type this system because the `let` expression defining `f` is abstractable with respect to the type of `y`. This is the case because the two-level analysis of the allocated types of the `let` expression reveals that none are already allocated, although the type of `y` will be allocated by further application.

Tofte-applicative cannot type this system because the one-level analysis reveals merely that the type of `y` is-or-will-be allocated, and the `let` expression is considered expansive, so the type abstraction is not permitted.

6.2 Tofte-applicative > Damas-III

(2) No known example.

[Tofte87] states on page 73 that an embedding exists.

6.3 MacQueen-weak > Tofte-applicative, Damas-III

```
(3)
let fold = fn f => fn i => fn l =>
            let data = ref l
                result = ref i
             in (while (!data <> []) do
                  (result := f(hd(!data))(!result);
                   data := tl(!data));
                 !result)
             end
 in let fast_reverse = fold cons []
     in (fast_reverse [3,5,7];
         fast_reverse [true,true,false])
     end
 end
```

[Tofte87], Example 4.5, mentioned on page 74.

MacQueen-weak can type this example, because the counting methods used by the typing algorithm deduce that `fold` must be applied three times before any allocation occurs, and since `fast_reverse` is defined by applying `fold` only twice, `fast_reverse` still has a type of strength one, and so may be generalized with respect to the type of the elements of the list.

Tofte-applicative cannot type this example because the one-level store typing analysis considers the expression `(fold cons [])` to be expansive, and therefore does not permit the type abstraction.

Damas-III cannot type this example because the two-level method also considers all type variables to have been allocated by the evaluation of `(fold cons [])`, and therefore does not permit the necessary type abstraction.

6.4 Damas-III, Tofte-applicative > MacQueen-weak

```
(4)     let rid = fn x => !(ref x)
         in let f = fn y => rid (fn a => a) y
             in (f 0;
                 f true)
             end
         end
```

Tofte-applicative can type this example, because the defining expression for `f` is a lambda abstraction, which is considered non-expansive, and so the type abstraction with respect to the type of `y` is permitted even though it is an imperative type. (Damas-III can also type this example, because the two-level analysis also reveals that the lambda expression defining `f` has not yet allocated at any types.)

MacQueen-weak cannot type this example, because `rid` must be given a type with strength one, and yet `rid`, after instantiation, is applied twice. This lowers the strength so that the strength of the type of `y` becomes zero. Therefore, the type abstraction is not permitted.

6.5 FX-89-pure > Damas-III, Tofte-applicative, MacQueen-weak

```
(5)     let nop = fn f => fn x =>
                   let g = fn y => f x
                    in x
                    end
         in let h = nop (fn a => !(ref a))
                        (fn b => b)
             in (h 0;
                 h true)
             end
         end
```

FX-89-pure can type this example because the side-effect analysis system correctly determines that `nop` has no latent side-effects, because the evaluation of `nop` applied to any arguments `f` and `x` will merely return `x`. If `f` were applied by `nop`, then the latent effect of the type of `nop` would include the latent effect of its argument type, the type of `f`. Therefore, the defining expression for `h` is pure, and may be abstracted with respect to the type of `b`.

Damas-III and Tofte-applicative cannot type this example because the store-typing analysis methods assume that allocation has occured at the type of `a` during the evaluation of the defining expression for `h` (the application of `nop`). The binding of `g` in the definition of `nop` constrains the type of `a` to be the same as the type of `b`. Therefore, type abstraction with respect to the type of `b` is not permitted.

MacQueen-weak cannot type this example because the maximum weakness permitted for the type of `a` is zero, because it is an operand of `nop`. Therefore, the type of `b` is forced to strength zero and the type abstraction is not permitted. My intuition for this is that `nop` is presumed to apply its arguments completely.

6.6 Damas-III, Tofte-applicative, MacQueen-weak > FX-89-pure

```
(6)     let k = fn a =>
```

```
            let r = ref a
             in fn b => !r
             end
    in let f = k []
        in (f 0;
            f true;
            false)
        end
    end
```

Damas-III, Tofte-applicative, and MacQueen-weak can type this example because the defining expression for `f` can be abstracted with respect to the type of `b`. All three systems will not permit abstraction with respect to the type of `a`, because an allocation at that type will have occured. However, this does not prevent the other abstraction, because the types of `a` and `b` are not related.

FX-89-pure cannot type this example because the abstraction rule requires that no allocations have taken place, and does not distinguish between the type variables at which allocations have taken place and the type variables at which no allocation have (or will) occur.

6.7 Further Speculation

However, the above example will be typed by FX-89-pure if an explicit abstraction is inserted within the definition of `k`. It is also interesting to observe that even with an explicit type abstraction in the definition of `k`, it will not be possible to give `f` a generic type, because of the pure-abstraction rule. Yet `f` will be automatically projected as required in this example, and `f` can be opened explicitly as well. Special casing the abstraction rule in `let` to permit generalization by opening would circumvent this peculiarity, although it will not eliminate the need for explicit abstractions.

Also, I do not expect explicit abstractions to solve this problem in general. Including types in allocation (and perhaps other) effects, and relaxing the pure-abstraction rule to examine the side effects and selectively permit type abstractions should provide a much more general treatment of this problem. I call this the "`Alloc@T`" typing system. This system is essentially the system which is mentioned in [Damas85] on pages 90–91, where he observes that attaching sets of types to type arrows will complicate the unification algorithm for types. Damas-III therefore attaches a set of types to type scheme arrows and also a set of types to typing assertions. Tofte-applicative may be viewed as attaching a single set of types to type schemes.

7 Summary

Damas-III is strictly superior to Tofte-applicative, but MacQueen-weak and FX-89-pure are incomparable to either of the above. Tofte has suggested in [Tofte89] that MacQueen-weak is strictly superior to Tofte-applicative, but this is not the case (see example (4)).

Appendix (sml)

Examples provided in sml syntax.

```
(1)     let val f = let val x = (fn x => x) 1
                     in (fn y => !(ref y))
                     end
         in (f 1; f true)
         end;
```

(2) No known example.

```
(3)
let fun fold f i l =
         let val data = ref l
             and result = ref i
          in while (!data <> []) do
              (result := f(hd (!data))(!result);
               data := tl(!data)
               );
             !result
          end
    and cons a b = a::b
    val fast_reverse = fold cons []
 in fast_reverse [3,5,7];
    fast_reverse [true,true,false]
 end;
```

```
(4)     let fun id x = x
            and rid x = !(ref x)
            fun f y = rid id y
         in (f 0;
             f true)
         end;
```

```
(5)     let fun id b = b
            and rid a = !(ref a)
            and nop f x =
                 let fun g y = f x
                  in x
                  end
            val h = nop rid id
         in (h 0;
```

```
          h true)
      end;

(6)     let fun ka =
              let val r = ref a
               in fn b => !r
               end
          val f = k []
      in (f 0;
          f true;
          false)
      end;
```

Appendix (fx)

Examples provided in FX-89 syntax.

```
(1)
(let ((f (let ((x ((lambda (x) x) 1)))
           (lambda (y) (get (new y))))))
  (begin
    (f 1)
    (f #t)))
```

(2) No known example.

```
(3)
(let ((fold (lambda (f i) (lambda (l)
        (let ((data (new l))
              (result (new i)))
          (begin
            (while (not (null? (get data)))
              (begin
                (set result (f (car (get data))
                               (get result)))
                (set data (cdr (get data)))))
            (get result)))))))
  (let ((fast_reverse (fold cons nil)))
    (begin
      (fast_reverse (list 3 5 7))
      (fast_reverse (list #t #t #f)))))

(4)     (let ((id (lambda (x) x))
              (rid (lambda (x) (get (new x)))))
          (let ((f (lambda (y) ((rid id) y))))
            (begin
              (f 0)
              (f #t))))

(5)
(let ((id (lambda (b) b))
      (rid (lambda (a) (get (new a))))
      (nop (lambda (f)
             (lambda (x)
               (let ((g (lambda (y) (f x))))
                 x)))))
  (let ((h ((nop rid) id)))
    (begin
      (h 0)
      (h #t))))

(6)     (let ((k (lambda (a)
                   (let ((r (new a)))
                     (lambda (b) (get r))))))
          (let ((f (k nil)))
            (begin
              (f 0)
              (f #t)
              #f)))
```

References

[Damas82] Damas, L., Milner, R., "Principal type-schemes for functional programs", *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, January 1982, pages 207–212.

[Damas85] Damas, L., "Type Assignment in Programming Lanuages", Ph.D Thesis CST-33-85, University of Edinburgh, April 1985.

[Gifford87] Gifford, D K, Jouvelot, P., Lucassen, J. M, Sheldon, M A, *The FX-87 Reference Manual*, MIT/LCS/TR-407, October 1987.

[Hindley69] Hindley, R., "The principal type-scheme of an object in combinatory logic", *Transactions of the American Mathematical Society*, vol. 146, 1969, pages 29–60.

[Lucassen87] Lucassen, J. M, *Types and Effects: Towards the Integration of Functional and Imperative Programming*, Ph.D Thesis MIT/LCS/TR-408, Massachusetts Institute of Technology, September 1987.

[MacQueen84] MacQueen, D, "Modules for Standard ML", *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, 1984, pages 198–207.

[Milner78] Milner, R., "A Theory of Type Polymorphism in Programming", *Journal of Computer and System Sciences*, vol. 17, 1978, pages 349–375.

[Morris68] Morris, J. H., *Lambda-Calculus Models of Programming Languages*, Massachusetts Institute of Technology, MAC-TR-57, 1968.

[O'Toole89] O'Toole, James William, Jr., *Type Reconstruction with First Class Polymorphic Values*, MIT/LCS/TM-380, 1989.

[Tofte87] Tofte, Mads, *Operational Semantics and Polymorphic Type Inference*, Ph.D Thesis, University of Edinburgh, 1987.