J. Functional Programming **4** (1): 117–123, January 1994 © 1994 Cambridge University Press

FUNCTIONAL PEARL
On generating unique names

LENNART AUGUSTSSON, MIKAEL RITTRI AND DAN SYNEK

Department of Computing Science, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden
(*E-mail:* {augustss,rittri,synek}@cs.chalmers.se)

1 Introduction

And Joktan begat Almodad, and Sheleph, and Hazarmaveth, and Jerah, and Hadoram, and Uzal, and Diklah, and Obal, and Abimael, and Sheba, and Ophir, and Havilah, and Jobab: all these were the sons of Joktan.

— *Genesis 10:26–29*

In a lazy, purely functional language, it is awkward to generate new and unique names. The *gensym* function (figure 1) is impure, since its result depends on a hidden counter. We can instead pass the counter as an argument, and regard it as representing the infinite supply of unused names. But the usual implementation (figure 2) has drawbacks: the access to the names is sequential, and to evaluate the *n*th name, you must first evaluate all previous names. This can cause

lost laziness: a large datastructure with unique names may have to be evaluated only because the next unused name is needed, and

lost parallelism: one evaluation may have to wait for another one to return the next unused name.

We will implement counter passing by hidden calls to *gensym*, so that the visible effect is referentially transparent. The generation of names will then not destroy laziness or parallelism.

The sequential access is still awkward, though. Peter Hancock (1987) has suggested a more flexible interface: since a name supply is an infinite set, it can be split into two disjoint and infinite subsets (figure 3). His implementation, which represents names by lists of integers, is inefficient and seldom used. An implementation with integers is possible (figure 4), but they overflow quickly unless you use integers of arbitrary size, which are inefficient, too. Fortunately, our hidden calls to *gensym* will make Hancock's interface as efficient as counter passing, or better.

Except figure 1, our examples will be in the Haskell language (Hudak et al., 1992).

2 Hiding *gensym* in a tree

In the *Doubling* module (Fig. 4), *gen* generated an infinite binary tree with distinct names in each node. We replace this *gen* by one which places the expression

```
(* gensym : 'a → int *)
local val counter = ref 0
in fun gensym(_) = (counter := !counter + 1;      !counter)
end
```

Fig. 1. A *gensym* function with a private counter, written in Standard ML. The function ignores its argument, increments the counter and returns its new value.

```
module CounterPassing(
    Name, NameSupply, initialNameSupply, getNameDeplete)
where
    data Name = MkName Int deriving (Eq)
    data NameSupply = MkNameSupply Int
    initialNameSupply :: NameSupply
    getNameDeplete :: NameSupply → (Name, NameSupply)
    initialNameSupply = MkNameSupply 0
    getNameDeplete (MkNameSupply i) = (MkName i, MkNameSupply(i+1))
```

Fig. 2. A simple implementation of counter passing, written in Haskell. The counter i represents the supply of integers $\geq i$. Overflow can occur after 2^{32} steps.

```
interface Hancock
where
    data Name
    instance Eq Name
    data NameSupply
    initialNameSupply  :: NameSupply
    getNameDeplete     :: NameSupply → (Name, NameSupply)
    splitNameSupply    :: NameSupply → (NameSupply, NameSupply)
```

(a) Every name supply represents an infinite set of names.
(b) *getNameDeplete*(s) returns an element n of s and an infinite subset of $s - \{n\}$.
(c) *splitNameSupply*(s) returns two infinite disjoint subsets of s.

Fig. 3. Hancock's specification.

```
module Doubling(
    Name, NameSupply, initialNameSupply, getNameDeplete, splitNameSupply)
where
    data Name = MkName Integer deriving (Eq)
    data NameSupply = MkNameSupply Name NameSupply NameSupply
    initialNameSupply = gen 1
        where gen i = MkNameSupply (MkName i) (gen(2*i)) (gen(2*i+1))
    getNameDeplete(MkNameSupply n s1 _) = (n, s1)
    splitNameSupply(MkNameSupply _ s1 s2) = (s1, s2)
```

Fig. 4. A simple implementation of Hancock's specification. The integer whose binary notation is b represents the infinite supply of integers whose binary notations begin with b. The integers become larger than 2^{32} after 32 consecutive splits, hence we use the Haskell type *Integer* (arbitrarily sized integers) rather than *Int* (machine integers).

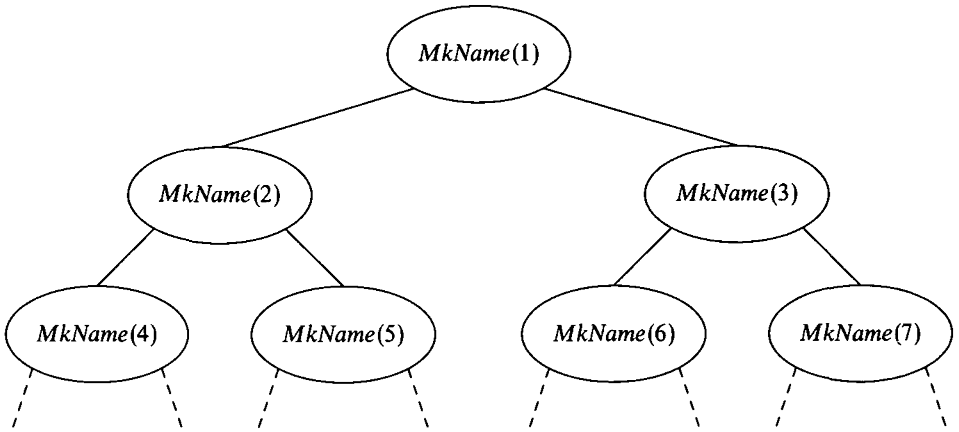

Fig. 5. The tree represented by *initialNameSupply* in Fig. 4.

MkName(*gensym*()) in each node (Fig. 6). These expressions are evaluated only when a name is used, each evaluates to *MkName*(i) where i is a distinct integer, and as usual, the expressions are replaced by their values once they have been evaluated. Since the only visible operation on names is equality, users cannot distinguish between the *Doubling* and the *HideGensym* module. But with the *Doubling* module, the integers would become larger than 2^{32} after 32 consecutive splits, whereas with *gensym*, they do not become that large until 2^{32} names have been used. This is why we dare to use short integers in *HideGensym* but not in *Doubling*.

The unevaluated *gensym* applications in the nodes are a simple variation of the *decisions* or *oracles* which have been suggested for handling nondeterminism in functional languages (Burton, 1988; Augustsson, 1989). The concrete names are nondeterministic, since they depend on the evaluation order, hence only equality of names is visible (but see section 3).

A word of warning: a too clever compiler might recognize the repeated subexpression *gen x* in Fig. 6, and implement *gen* by code which creates sharing (or even cycles). This would not destroy the referential transparency of the interface, but the binary tree would no longer appear to have distinct names in each node. If such compiler optimizations cannot be turned off or fooled, one must generate code for the *gen* function by hand.

If a compile-time analysis of a program can guarantee that every name supply is used at most once, either to do *getNameDeplete* or *splitNameSupply*, the tree becomes unnecessary and we can save some work (figure 7).

3 Making names visible

Often we need to print names, not just compare them for equality. It is possible to assign unique strings to names without revealing their secret representation, but it is awkward. It is much simpler not to hide names at all, that is, to make *Name* a synonym for *Int*. Since the concrete integers in the nodes depend on the evaluation order, *initialNameSupply* no longer denotes a fixed value. Rather, each

```
module HideGensym(
    Name, NameSupply, initialNameSupply, getNameDeplete, splitNameSupply)
where
    gensym :: a → Int   -- implemented in assembler
    data Name = MkName Int deriving (Eq)
    data NameSupply = MkNameSupply Name NameSupply NameSupply
    initialNameSupply = gen ()
        where gen x = MkNameSupply (MkName(gensym x)) (gen x) (gen x)
    getNameDeplete (MkNameSupply n s1 _) = (n, s1)
    splitNameSupply (MkNameSupply _ s1 s2) = (s1, s2)
```

Fig. 6. An implementation of Hancock's specification, using an infinite binary tree with the expression *MkName*(*gensym*()) in each node. The *gensym* must be coded in assembler, and possibly also the *gen* function.

```
module OneTimeSupplies(
    Name, NameSupply, initialNameSupply, getNameDeplete, splitNameSupply)
where
    gensym :: a → Int   -- implemented in assembler
    data Name = MkName Int deriving (Eq)
    data NameSupply = MkNameSupply
    initialNameSupply = MkNameSupply
    getNameDeplete s = (MkName(gensym(s)), MkNameSupply)
    splitNameSupply MkNameSupply = (MkNameSupply, MkNameSupply)
```

Fig. 7. An unsafe implementation of Hancock's specification. It is referentially transparent only if each supply is used at most once.

time a program is run, it looks as if a new value is provided, and we know only that it is a tree with distinct integers in the nodes. To fit this into Haskell, we can simply add a new request *GetInitialNameSupply* to the I/O system, alongside *GetArgs*, *GetEnv* and the others (Hudak et al., 1992). The same solution was adopted by Burton (1988) and Augustsson (1989).

4 Benchmarks

We have tested the various implementations using the Haskell B. compiler of Chalmers. Our test programs renamed the bound variables of a large lambda expression so that they got unique names. We also run two base tests, one which renamed all variables to the *same* name, and one which used an unhidden *gensym*. The results can be found in Table 1. The tests were made so that the implementations which used Hancock's operations did not gain anything from increased laziness or parallelism.

The implementation in Fig. 6 is now available in the Chalmers Haskell compiler — just import a module called *NameSupply*.

Table 1. *The times to rename the variables of a large lambda expression, scaled to make the first base time 1.*

Base tests	1	renaming all bound variables to the same name
	1.13	unhidden *gensym*
Counter passing	2.18	(Fig. 2)
Hancock's operations	6.34	lists of short integers (Hancock, 1987)
	4.89	doubling of long integers (Fig. 4)
	1.57	hidden *gensym* (Fig. 6)
	1.26	hidden *gensym*, one-time use of supplies (Fig. 7)

Related work

Wadler (1990) tidies up counter passing by wrapping up the "plumbing" into a monad. His monad could be implemented by a hidden counter. One could also access Hancock's split operation in a more monadic style. Paulson (1991) describes other methods to hide assignments behind a referentially transparent interface.

Acknowledgements

We thank John Hughes, Thomas Johnsson, Mark Jones and Simon Jones for valuable comments.

References

Augustsson, L. (1989) Functional non-deterministic programming, or How to make your own oracle. PMG memo 66, Dept. of Comput. Sci., Chalmers Univ. of Tech., Göteborg, Sweden (augustss@cs.chalmers.se).

Burton, F. W. (1988) Nondeterminism with referential transparency in functional programming languages. *Computer Journal*, **31**(3):243–247.

Hancock, P. (1987) A type-checker. Chapter 9 (pp. 163–182) of Peyton Jones, S. L., *The Implementation of Functional Programming Languages*, Prentice-Hall.

Hudak, P. et al. (1992) Haskell special issue. *ACM SIGPLAN Notices*, **27**(5).

Paulson, L. C. (1991) Imperative programming in ML. Chapter 8 (pp. 279–313) of *ML for the Working Programmer*, Cambridge University Press.

Wadler, P. (1990) Comprehending monads. In *ACM Conf. on LISP and Functional Programming*, Nice, pp. 61–78, ACM Press.

Appendix

Here are two of the test programs, and some auxiliary operations on name supplies that can be useful. The test was to rename a large lambda expression, and then comparing the result to itself to force full evaluation. The test was thus

r==*r* **where** *r* = *rename* (*expr*(15)) *initialNameSupply*,

```
data (Eq n) =>    Term n = Var n
                         | Lam n (Term n)
                         | App (Term n) (Term n)
                  deriving (Eq)

replace nNew nOld (Var n)       | nOld==n    = Var nNew
replace nNew nOld (Lam n t)     | nOld/=n    = Lam n (replace nNew nOld t)
replace nNew nOld (App t1 t2)                = App  (replace nNew nOld t1)
                                                    (replace nNew nOld t2)
replace nNew nOld t             | otherwise  = t
```

Fig. 8. The definition of lambda expressions, plus a naïve replacement of free occurrences of *nOld* by *nNew*.

```
renamer :: Term Name → NameSupply → (Term Name, NameSupply)
renamer (Var n) s        =  (Var n, s)
renamer (Lam n t) s      =  (Lam n' (replace n' n t'), s'')
                            where   (n', s') = getNameDeplete s
                                    (t', s'') = renamer t s'
renamer (App t1 t2) s    =  (App t1' t2', s'')
                            where   (t1', s') = renamer t1 s
                                    (t2', s'') = renamer t2 s'

rename :: Term Name → NameSupply → Term Name
rename t s = fst(renamer t s)
```

Fig. 9. Renaming the bound variables in a lambda expression by counter passing. The program in Fig. 2 was used, except that both *Name* and *NameSupply* were made synonyms for *Int*.

```
rename :: Term Name → NameSupply → Term Name
rename (Var n) s = Var n
rename (Lam n t) s =
      case getNameDeplete s of
            (n',s') →    Lam n' (replace n' n t')
                         where   t' = rename t s'
rename (App t1 t2) s =
      case splitNameSupply s of
            (s1, s2) →   App t1' t2'
                         where   t1' = rename t1 s1
                                 t2' = rename t2 s2
```

Fig. 10. Renaming the bound variables in a lambda expression by splitting infinite name supplies.

with

$expr(0) = Lam\ 0\ (Var\ 0)$
$expr(k) = App\ t\ t$ **where** $t = expr(k-1)$

and a datatype of lambda expressions as in Fig. 8.

Note: in an application *rename t s*, the expression *t* should not contain any names that are in the name supply *s* — otherwise the names in the supply are not “new”, of course.

```
listName :: NameSupply → [Name]
listNameDeplete :: NameSupply → ([Name], NameSupply)
listNameSupply :: NameSupply → [NameSupply]

listName (MkNameSupply n _ s2) = n : listName s2
listNameDeplete s@(MkNameSupply _ s1 _) = (listName s, s1)
listNameSupply (MkNameSupply _ s1 s2) = s1 : listNameSupply s2
```

Fig. 11. Some auxiliary operations that can be added to the name supply module.