LAMBDA CALCULUS FOR ENGINEERS

PIETER H. HARTEL AND WILLEM G. VREE

Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente
e-mail address: pieter.hartel@utwente.nl

Faculty of Technology, Policy and Management, Technical University of Delft
e-mail address: w.g.vree@tudelft.nl

Abstract. In pure functional programming it is awkward to use a stateful sub-computation in a predominantly stateless computation. The problem is that the state of the sub-computation has to be passed around using ugly plumbing. Classical examples of the plumbing problem are: providing a supply of fresh names, and providing a supply of random numbers. We propose to use (deterministic) inductive definitions rather than recursion equations as a basic paradigm and show how this makes it easier to add the plumbing.

Introduction

In the λ-calculus, the *variable convention* [2, 2.1.13] states that "If $\mathsf{M}_1, \ldots \mathsf{M}_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables". One such mathematical context is the defining equation of substitution for a lambda abstraction:

$$(\lambda \mathsf{y}.\mathsf{M})[\mathsf{x} := \mathsf{N}] = \lambda \mathsf{y}.\mathsf{M}[\mathsf{x} := \mathsf{N}]$$

To adhere to the variable convention, it may be necessary to rename bound variables, which in turn requires a supply of fresh names. In the substitution example below z is a name that appears only here:

$$(\lambda \mathsf{y}.\mathsf{x}\ \mathsf{y})[\mathsf{x} := \mathsf{y}] \equiv (\lambda \mathsf{z}.\mathsf{y}\ \mathsf{z})$$

The authors are engineers, and like other engineers, we are concerned with this seemingly trivial issue of providing freshness: over the past 20 years a number of papers have been wholly or partly devoted to solutions of providing fresh names, including Peyton Jones [11, Chapter 9], Augustsson, Rittri and Synek [1], Launchbury [10], Boulton [3], Peyton Jones and Marlow [12], Shinwell, Pitts and Gabbay [13], Cheney and Urban [5], and Cheney [4]. The problem is: "how to get a supply of fresh names inconspicuously from the source to the destination". This question arises not only in an abstract setting, such as when defining substitution for the λ-calculus, the same problem frequently crops up in concrete settings

2000 ACM Subject Classification: D.3.3 Language Constructs and Features, F.4.1 Mathematical Logic.
Key words and phrases: binding, substitution, random numbers.

too, for example in compilers and program transformation, and also in other areas such as scientific computing when a fresh random number is needed. Interestingly, in our paper we discuss a program transformation for providing a fresh name supply that itself, i.e. at the meta level requires fresh names!

In an impure language such as SML the problem of supplying fresh names can be solved by a function such as gensym below with a side effect [1]. The function gensym ignores its argument, increments the counter and returns its new value:

```
local val counter  =  ref 0
in fun gensym(_)   =  (count :=!counter + 1; !counter)
end
```

We feel that a logician would not be enthusiastic about sacrificing referential transparency, and he would thus favour a purely declarative approach. In the declarative approach the name supply must be passed around as an argument to calls of functions and/or predicates, and, depending on the details of the approach, a name supply may also have to be returned as a function result. This is known as the plumbing problem, because it is reminiscent of the problem that arises when a plumber installs central heating in a house that was built with fireplaces only. It is illustrative to explore this metaphor in a little more depth.

Firstly, the central heating engineer knows where the hot water pipes should begin and end: the boiler is the (unique) source and the radiators are the (many) destinations. Similarly, in our running example the software engineer knows where the source of the fresh names should be (most likely in the main expression) and where the destination is (in the body of the substitute function or predicate if we are talking about the λ-calculus).

Secondly, the water for the radiators must be hot, so it is best to feed each radiator directly from the boiler rather than indirectly, via other radiators. Similarly, to ensure that each name in our running example is fresh, the names are best produced by a single source and fed directly to the destination, so that accidental reuse of names is avoided.

Thirdly, the software engineer must connect the source to the destination via intermediate function and/or predicate definitions, just like the piping laid by the central heating engineer works its way under floors and up the walls to connect the source and destination of hot water. To find the right spot for the pipes, such that the amount of work and the damage to the house are minimal requires skill and experience. Similarly, optimal plumbing of the name supply, which minimises the amount of restructuring of the code, requires skill and experience.

We observe that the structure of the house provides significant guidance to the central heating engineer; for example a pipe through the middle of a room is less appreciated than a pipe along the wall, neatly tucked away in the corner of a room. The contribution of this paper is to identify the corners and the rooms of a program, and thus to identify a similar structural support for the software engineer. We believe that a logical approach provides better structural guidance for plumbing than a functional approach. We illustrate this by showing how a specification of the λ-calculus with substitution in a natural deduction style, and the same specification in a logic programming style are superior to the functional style. Then we illustrate the ideas using a program for composing serial music that uses many random numbers.

The plan of the paper is as follows. The logical approach of Section 1 presents the reduction relation of the λ-calculus, showing that substitution when expressed as a relation

facilitates the plumbing of fresh names. The functional approach of Section 2 presents substitution in the conventional way as a set of recursive equations, showing that plumbing is a real issue. Using monads hides the plumbing, but this comes at the cost of more restructuring at the source and the destination. Section 3 shows a musical example, where the need for random numbers presents similar problems as the need for fresh names, and where the logical approach again offers relief. The final section concludes.

1. The logical approach

This section introduces our running example: the syntax and semantics of the untyped λ-calculus. The abstract syntax of the untyped λ-calculus is given by variables v, and expressions e as shown below. Strictly speaking, the third syntactic category σ is not part of the λ-calculus; it is needed to represent a single substitution.

$$\begin{aligned}\mathsf{v} &\equiv \mathsf{a} \mid \mathsf{b} \mid \mathsf{c} \ \ldots \\ \mathsf{e} &\equiv \mathsf{v} \mid \lambda\ \mathsf{v}\ \cdot\ \mathsf{e} \mid \mathsf{e}_1\ \mathsf{e}_2 \\ \sigma &\equiv [\ \mathsf{v}\ :=\ \mathsf{e}\ ]\end{aligned}$$

The semantics of the λ-calculus can be given by a reduction relation $\overset{r}{\Rightarrow}$.

$$\overset{r}{\Rightarrow} :: \mathsf{e} \leftrightarrow \mathsf{e}$$

The reduction relation $\overset{r}{\Rightarrow}$ is usually specified in a natural deduction style. The β-reduction rule (below) would be the most interesting in this definition. In our example we assume an applicative order reduction strategy to normal form:

$$[\mathsf{beta}]\ \frac{\mathsf{e}_1 \overset{r}{\Rightarrow} \lambda\ \mathsf{v}\ \cdot\ \mathsf{e}'_1,\ \mathsf{e}_2 \overset{r}{\Rightarrow} \mathsf{e}'_2,\ \mathsf{e}'_1\ [\ \mathsf{v}\ :=\ \mathsf{e}'_2\ ] \overset{1}{\Rightarrow} \mathsf{e}''_1}{\mathsf{e}_1\ \mathsf{e}_2 \overset{r}{\Rightarrow} \mathsf{e}''_1}$$

The third premiss of the rule states that all free occurrences of v should be replaced by e'_2 in the expression e'_1 by the substitution relation $\overset{1}{\Rightarrow}$ introduced below.

It is customary to specify substitution using recursive equations (See Section 2). However, considering that we are in the process of specifying a reduction *relation* we find it more natural to also specify substitution using a relation. This avoids a paradigm shift.

As usual, substitution is written by a juxtaposition $\mathsf{e}\ \sigma$, where e is the expression and σ is the substitution (for typing purposes the term $\mathsf{e}\ \sigma$ may be interpreted as a tuple):

$$\overset{1}{\Rightarrow} :: \mathsf{e}\ \sigma \leftrightarrow \mathsf{e}$$

The two variable axioms below specify when to perform the actual substitutions. The main purpose of the abstraction and application rules is to traverse the expression e and to offer every variable occurrence to the substitution σ.

$$[\mathsf{var}^1]\ \ \mathsf{v}_1\ [\ \mathsf{v}_2\ :=\ \mathsf{e}_2\ ] \stackrel{1}{\Rightarrow} \mathsf{e}_2,\ \textbf{if}\ \mathsf{v}_1 = \mathsf{v}_2$$

$$[\mathsf{var}^2]\ \ \mathsf{v}_1\ [\ \mathsf{v}_2\ :=\ \mathsf{e}_2\ ] \stackrel{1}{\Rightarrow} \mathsf{v}_1,\ \textbf{if}\ \mathsf{v}_1 \neq \mathsf{v}_2$$

$$[\mathsf{abs}]\ \ \frac{\textbf{fresh}\ \mathsf{w},\ \mathsf{e}\ [\ \mathsf{v}\ :=\ \mathsf{w}\ ] \stackrel{1}{\Rightarrow} \mathsf{e}',\ \mathsf{e}'\ \sigma \stackrel{1}{\Rightarrow} \mathsf{e}''}{(\lambda\ \mathsf{v}\ \cdot\ \mathsf{e})\sigma \stackrel{1}{\Rightarrow} \lambda\ \mathsf{w}\ \cdot\ \mathsf{e}''}$$

$$[\mathsf{app}]\ \ \frac{\mathsf{e}_1\ \sigma \stackrel{1}{\Rightarrow} \mathsf{e}'_1,\ \mathsf{e}_2\ \sigma \stackrel{1}{\Rightarrow} \mathsf{e}'_2}{(\mathsf{e}_1\ \mathsf{e}_2)\sigma \stackrel{1}{\Rightarrow} \mathsf{e}'_1\ \mathsf{e}'_2}$$

The abstraction rule [abs] exposes an important technical detail. The keyword **fresh** declares our intention that w is a fresh variable, to make sure that free occurrences of v in the substitution σ are not accidentally captured by the bound variable v of the λ abstraction. We will make our intention precise in the next section.

The [abs] rule is hugely inefficient in the sense that all bound variables are renamed (and more than once), whether this is necessary or not. In such a case Henk would say "Tegen exponentiële domheid is het slecht op pico-en".

1.1. **Data refinement.** The generation of fresh variables can be made explicit by "plumbing" a name supply ns into the reduction relation. We will refine our existing relation $\stackrel{1}{\Rightarrow}$ to a new relation $\stackrel{2}{\Rightarrow}$ between pairs of expressions and name supplies as follows:

$$\stackrel{2}{\Rightarrow} :: (\mathsf{e}\ \sigma,\ \mathsf{ns}) \leftrightarrow (\mathsf{e},\ \mathsf{ns})$$

Assuming that the name supply is represented by a list of names (the constructor : separates the head from the tail of the list), the refinement of the axioms and rules defining the relation $\stackrel{1}{\Rightarrow}$ to the axioms and rules of $\stackrel{2}{\Rightarrow}$ is:

$$[\mathsf{var}^1]\ \ (\mathsf{v}_1\ [\ \mathsf{v}_2\ :=\ \mathsf{e}_2\ ],\ \mathsf{ns}) \stackrel{2}{\Rightarrow} (\mathsf{e}_2,\ \mathsf{ns}),\ \textbf{if}\ \mathsf{v}_1 = \mathsf{v}_2$$

$$[\mathsf{var}^2]\ \ (\mathsf{v}_1\ [\ \mathsf{v}_2\ :=\ \mathsf{e}_2\ ],\ \mathsf{ns}) \stackrel{2}{\Rightarrow} (\mathsf{v}_1,\ \mathsf{ns}),\ \textbf{if}\ \mathsf{v}_1 \neq \mathsf{v}_2$$

$$[\mathsf{abs}]\ \ \frac{\mathsf{ns}_0 \stackrel{id}{\Rightarrow} \mathsf{w} : \mathsf{ns}_1,\ (\mathsf{e}\ [\ \mathsf{v}\ :=\ \mathsf{w}\ ],\ \mathsf{ns}_1) \stackrel{2}{\Rightarrow} (\mathsf{e}',\ \mathsf{ns}_2),\ (\mathsf{e}'\ \sigma,\ \mathsf{ns}_2) \stackrel{2}{\Rightarrow} (\mathsf{e}'',\ \mathsf{ns}_3)}{((\lambda\ \mathsf{v}\ \cdot\ \mathsf{e})\sigma,\ \mathsf{ns}_0) \stackrel{2}{\Rightarrow} (\lambda\ \mathsf{w}\ \cdot\ \mathsf{e}'',\ \mathsf{ns}_3)}$$

$$[\mathsf{app}]\ \ \frac{(\mathsf{e}_1\ \sigma,\ \mathsf{ns}_0) \stackrel{2}{\Rightarrow} (\mathsf{e}'_1,\ \mathsf{ns}_1),\ (\mathsf{e}_2\ \sigma,\ \mathsf{ns}_1) \stackrel{2}{\Rightarrow} (\mathsf{e}'_2,\ \mathsf{ns}_2)}{((\mathsf{e}_1\ \mathsf{e}_2)\sigma,\ \mathsf{ns}_0) \stackrel{2}{\Rightarrow} (\mathsf{e}'_1\ \mathsf{e}'_2,\ \mathsf{ns}_2)}$$

The definition of the new relation $\stackrel{2}{\Rightarrow}$ is less clear than the original definition $\stackrel{1}{\Rightarrow}$, because of the clutter introduced by the plumbing of the name supply. However, the structure of the rules and axioms is unchanged. In fact, the addition of the name supply is a trivial mechanical process.

Returning once more to our metaphor, the keyword **fresh** in the abstraction clause of the substitution function indicates the destination of the name supply, and the main

expression below indicates the source, where $[\ldots]$ represents a potentially infinite list of fresh names:

$$\mathsf{main}\ \mathsf{e} = (\mathsf{e},\ [\ldots]) \overset{r}{\Rightarrow}$$

1.2. **Prolog.** A name supply can also be added by the same purely mechanical process to a Prolog version of relation $\overset{1}{\Rightarrow}$, yielding a Prolog version of relation $\overset{2}{\Rightarrow}$. We present the result of this transformation below as predicate subst, which works on a concrete syntax of the λ-calculus:

$$\mathsf{subst}(\mathsf{v}(\mathsf{V2}), \mathsf{sigma}(\mathsf{V2}, \mathsf{E2}), \mathsf{NS}, \mathsf{E2}, \mathsf{NS}).$$
$$\mathsf{subst}(\mathsf{v}(\mathsf{V1}), \mathsf{sigma}(\mathsf{V2}, \mathsf{E2}), \mathsf{NS}, \mathsf{v}(\mathsf{V1}), \mathsf{NS}).$$

$$\mathsf{subst}(\mathsf{ap}(\mathsf{E1}, \mathsf{E2}), \mathsf{Sigma}, \mathsf{NS0}, \mathsf{ap}(\mathsf{E1}', \mathsf{E2}'), \mathsf{NS2})$$
$$\leftarrow \mathsf{subst}(\mathsf{E1}, \mathsf{Sigma}, \mathsf{NS0}, \mathsf{E1}', \mathsf{NS1}), \mathsf{subst}(\mathsf{E2}, \mathsf{Sigma}, \mathsf{NS1}, \mathsf{E2}', \mathsf{NS2}).$$

$$\mathsf{subst}(\mathsf{lamb}(\mathsf{V}, \mathsf{E}), \mathsf{Sigma}, [\mathsf{W}|\mathsf{NS1}], \mathsf{lamb}(\mathsf{W}, \mathsf{E}''), \mathsf{NS3})$$
$$\leftarrow \mathsf{subst}(\mathsf{E}, \mathsf{sigma}(\mathsf{V}, \mathsf{v}(\mathsf{W})), \mathsf{NS1}, \mathsf{E}', \mathsf{NS2}), \mathsf{subst}(\mathsf{E}', \mathsf{Sigma}, \mathsf{NS2}, \mathsf{E}'', \mathsf{NS3}).$$

A Prolog version of the reduction relation can also be plumbed in a straightforward fashion:

$$\mathsf{reduce}(\mathsf{ap}(\mathsf{E1}, \mathsf{E2}), \mathsf{NS0}, \mathsf{E1}'', \mathsf{NS3})$$
$$\leftarrow \mathsf{reduce}(\mathsf{E1}, \mathsf{NS0}, \mathsf{lamb}(\mathsf{V}, \mathsf{E1}'), \mathsf{NS1}), \mathsf{reduce}(\mathsf{E2}, \mathsf{NS1}, \mathsf{E2}', \mathsf{NS2}),$$
$$\mathsf{subst}(\mathsf{E1}', \mathsf{sigma}(\mathsf{V}, \mathsf{E2}'), \mathsf{NS2}, \mathsf{E1}'', \mathsf{NS3}).$$

$$\mathsf{reduce}(\mathsf{E}, \mathsf{NS}, \mathsf{E}, \mathsf{NS}).$$

The ease at which plumbing is added leads to the conclusion that Prolog shares the advantages of the logical style. Again the destination of the name supply is in the abstraction clause of the subst predicate, whereas the source is in the main query:

$$\leftarrow \mathsf{reduce}(\mathsf{E}, [\ldots], \mathsf{E}', \mathsf{NS}').$$

Note: In Prolog the problem can also be solved directly by just using a new variable W as shown below. This works as long as one does not (accidentally) bind the variables in the top-level query.

$$\mathsf{subst}(\mathsf{lamb}(\mathsf{V}, \mathsf{E}), \mathsf{Sigma}, \mathsf{lamb}(\mathsf{W}, \mathsf{E}''))$$
$$\leftarrow \mathsf{subst}(\mathsf{E}, \mathsf{sigma}(\mathsf{V}, \mathsf{v}(\mathsf{W})), \mathsf{E}'), \mathsf{subst}(\mathsf{E}', \mathsf{Sigma}, \mathsf{E}'').$$

As we shall see in Section 3 this solution is not able to create random numbers.

2. The functional approach

We should now like to turn our attention to the problems of plumbing in the functional style. The function subst_1 below is the functional equivalent of the relation $\overset{1}{\Rightarrow}$. Again, the keyword **fresh** declares our intention that w is a fresh variable, which will be made precise below.

$$
\begin{array}{lll}
\mathsf{subst}_1 & :: & \mathsf{e}{\rightarrow}\sigma{\rightarrow}\mathsf{e} \\
\mathsf{subst}_1\ \mathsf{v}_1\ [\ \mathsf{v}_2\ :=\ \mathsf{e}_2\ ] & = & \mathsf{e}_2,\ \textbf{if}\ \mathsf{v}_1 = \mathsf{v}_2 \\
 & = & \mathsf{v}_1,\ \textbf{otherwise} \\
\mathsf{subst}_1(\lambda\ \mathsf{v}\ \cdot\ \mathsf{e})\sigma & = & \lambda\ \mathsf{w}\ \cdot\ \mathsf{subst}_1(\mathsf{subst}_1\ \mathsf{e}\ [\ \mathsf{v}\ :=\ \mathsf{w}\ ])\sigma \\
 & & \quad \textbf{where} \\
 & & \quad \textbf{fresh}\ \mathsf{w} \\
 & & \\
\mathsf{subst}_1(\mathsf{e}_1\ \mathsf{e}_2)\sigma & = & (\mathsf{subst}_1\ \mathsf{e}_1\ \sigma)(\mathsf{subst}_1\ \mathsf{e}_2\ \sigma)
\end{array}
$$

To develop the functional equivalent of $\stackrel{2}{\Rightarrow}$ we must plumb the name supply as before. However, in the functional style this requires a significant restructuring of the specification basically because of the use of nested expressions. (In the world of our central heating engineer this would be equivalent to encountering an unusual building with rooms within rooms).

The transformation to which the function subst_1 must be subjected consists of the following steps: (1) identify each relevant sub expression, (2) lift a relevant sub expression out of the containing expression, (3) replace the sub expression by a (fresh!) name, and (4) add an appropriate defining equation to the **where** clause. In the logical style, nested sub expression are possible, but in no case did we have to lift, name and move a relevant sub expression. Interestingly, we need fresh names in step (3), which suggests that a mechanised version of the transformations should probably be written in a logical style!

The result of the lift, name and move steps is presented below, showing that four new defining equations have been added (for $\mathsf{e}', \mathsf{e}'', \mathsf{e}'_1$, and e'_2).

$$
\begin{array}{lll}
\mathsf{subst}_2 & :: & \mathsf{e}{\rightarrow}\sigma{\rightarrow}\mathsf{e} \\
\mathsf{subst}_2\ \mathsf{v}_1\ [\ \mathsf{v}_2\ :=\ \mathsf{e}_2\ ] & = & \mathsf{e}_2,\ \textbf{if}\ \mathsf{v}_1 = \mathsf{v}_2 \\
 & = & \mathsf{v}_1,\ \textbf{otherwise} \\
\mathsf{subst}_2(\lambda\ \mathsf{v}\ \cdot\ \mathsf{e})\sigma & = & \lambda\ \mathsf{w}\ \cdot\ \mathsf{e}'' \\
 & & \quad \textbf{where} \\
 & & \quad \textbf{fresh}\ \mathsf{w} \\
 & & \quad \mathsf{e}' = \mathsf{subst}_2\ \mathsf{e}\ [\ \mathsf{v}\ :=\ \mathsf{w}\ ] \\
 & & \quad \mathsf{e}'' = \mathsf{subst}_2\ \mathsf{e}'\ \sigma \\
 & & \\
\mathsf{subst}_2(\mathsf{e}_1\ \mathsf{e}_2)\sigma & = & \mathsf{e}'_1\ \mathsf{e}'_2 \\
 & & \quad \textbf{where} \\
 & & \quad \mathsf{e}'_1 = \mathsf{subst}_2\ \mathsf{e}_1\ \sigma \\
 & & \quad \mathsf{e}'_2 = \mathsf{subst}_2\ \mathsf{e}_2\ \sigma
\end{array}
$$

The actual plumbing can be provided as follows:

$$
\begin{array}{lll}
\mathsf{subst}_3 & :: \mathsf{e}{\rightarrow}\sigma{\rightarrow}\mathsf{ns}{\rightarrow}(\mathsf{e},\ \mathsf{ns}) \\
\mathsf{subst}_3\ \mathsf{v}_1\ [\ \mathsf{v}_2\ :=\ \mathsf{e}_2\ ]\ \mathsf{ns} & = (\mathsf{e}_2,\ \mathsf{ns}),\ \textbf{if}\ \mathsf{v}_1 = \mathsf{v}_2 \\
 & = (\mathsf{v}_1,\ \mathsf{ns}),\ \textbf{otherwise} \\
\mathsf{subst}_3(\lambda\ \mathsf{v}\ \cdot\ \mathsf{e})\sigma\ \mathsf{ns}_0 & = (\lambda\ \mathsf{w}\ \cdot\ \mathsf{e}'',\ \mathsf{ns}_3) \\
 & \quad \textbf{where} \\
 & \quad \mathsf{w} : \mathsf{ns}_1 \quad = \mathsf{ns}_0 \\
 & \quad (\mathsf{e}',\ \mathsf{ns}_2) = \mathsf{subst}_3\ \mathsf{e}\ [\ \mathsf{v}\ :=\ \mathsf{w}\ ]\ \mathsf{ns}_1 \\
 & \quad (\mathsf{e}'',\ \mathsf{ns}_3) = \mathsf{subst}_3\ \mathsf{e}'\ \sigma\ \mathsf{ns}_2 \\
 & \\
\mathsf{subst}_3(\mathsf{e}_1\ \mathsf{e}_2)\sigma\ \mathsf{ns}_0 & = (\mathsf{e}'_1\ \mathsf{e}'_2,\ \mathsf{ns}_2) \\
 & \quad \textbf{where} \\
 & \quad (\mathsf{e}'_1,\ \mathsf{ns}_1) = \mathsf{subst}_3\ \mathsf{e}_1\ \sigma\ \mathsf{ns}_0 \\
 & \quad (\mathsf{e}'_2,\ \mathsf{ns}_2) = \mathsf{subst}_3\ \mathsf{e}_2\ \sigma\ \mathsf{ns}_1
\end{array}
$$

2.1. **Monads.** Monads [14] can be used to hide plumbing, so it is illustrative to look at a monadic version of the substitution function, subst_4 below, which uses a state monad SM. Here we assume that all the hard work of restructuring the specification has already been done as in function subst_3: All relevant sub expressions have been lifted, named and moved as before, although in this case instead of generating defining equations for the **where** clause, we generate binding expressions for the monad.

$$
\begin{array}{lll}
\mathsf{subst}_4 & :: \mathsf{e}{\rightarrow}\sigma{\rightarrow}\mathsf{SM\ e} \\
\mathsf{subst}_4\ \mathsf{v}_1\ [\ \mathsf{v}_2\ :=\ \mathsf{e}_2\ ] & = \mathsf{return}\ \mathsf{e}_2,\ \textbf{if}\ \mathsf{v}_1 = \mathsf{v}_2 \\
 & = \mathsf{return}\ \mathsf{v}_1,\ \textbf{otherwise} \\
\mathsf{subst}_4(\lambda\ \mathsf{v}\ \cdot\ \mathsf{e})\sigma & = \mathsf{do}\ \mathsf{w}\ \leftarrow\ \textbf{fresh} \\
 & \quad \mathsf{e}'\ \leftarrow\ \mathsf{subst}_4\ \mathsf{e}\ [\ \mathsf{v}\ :=\ \mathsf{w}\ ] \\
 & \quad \mathsf{e}''\ \leftarrow\ \mathsf{subst}_4\ \mathsf{e}'\ \sigma \\
 & \quad \mathsf{return}(\lambda\ \mathsf{w}\ \cdot\ \mathsf{e}'') \\
\mathsf{subst}_4(\mathsf{e}_1\ \mathsf{e}_2)\sigma & = \mathsf{do}\ \mathsf{e}'_1\ \leftarrow\ \mathsf{subst}_4\ \mathsf{e}_1\ \sigma \\
 & \quad \mathsf{e}'_2\ \leftarrow\ \mathsf{subst}_4\ \mathsf{e}_2\ \sigma \\
 & \quad \mathsf{return}(\mathsf{e}'_1\ \mathsf{e}'_2)
\end{array}
$$

Although there is no clutter of numbered variables for the name supply, the original specification still has to be restructured in much the same way as the previous example. In addition, the specification of "fresh" is not trivial and requires knowledge of the monad at hand (SM):

$$
\begin{array}{l}
\textbf{fresh} :: \mathsf{SM\ ns} \\
\textbf{fresh} = \mathsf{SM}(\lambda\ \mathsf{s}\ \cdot(\mathsf{head\ s},\ \mathsf{tail\ s}))
\end{array}
$$

The invocation of the substitute equation also has to be changed. Assume that the original equation is instantiated as follows:

$$\mathsf{ex}_2 = \mathsf{subst}_4\ \mathsf{ex}_1\ \sigma$$

The corresponding monadic invocation would then look like this:

$$
\begin{aligned}
(\mathsf{ex}_2,\ \_) = \mathsf{c}[&\texttt{"v1"},\ \texttt{"v2"},\ \ldots] \\
&\textbf{where} \\
&\mathsf{SM}\ \mathsf{c} = \mathsf{subst}_4\ \mathsf{ex}_1\ \sigma
\end{aligned}
$$

The example shows that using monads does not simplify the transformation compared to the plumbing of the name supply. Therefore, we suggest that the logical style is superior to the functional style since it makes the plumbing straightforward.

3. Plumbing of a real world program: Project One

To investigate the impact of the presented plumbing techniques on a larger program we study a well known piece of software, called *Project One*, written by the German composer Gottfried Michael Koenig [8, 9]. Although Project One (abbreviated PR-1) was originally written in Fortran, it is almost purely functional, mapping a set of input parameters to a list of characters: the composition. The only non-functional aspect of the program is the extensive use of random numbers.

PR-1 composes music according to the principles of *Total Serialism* [7]. This is music in which sets are used to derive all aspects of the composition, like pitch, duration and loudness. The philosophical background is the preference of the composer for an equal treatment of all musical elements in his work. Because a set does not contain repeated elements, no musical elements can be stressed by occurring more often than others. Integral serial music can be viewed as a "socialist" attitude towards music.

The PR-1 program has inspired a follow up effort in the late eighties, when Koenig assembled a large team of composers and advisers to create Project Three (PR-3). The list of participants[1] of the kick-off conference shows a well known expert in the area of λ-calculus as a prominent member of the team. It was during this effort and influenced by the said expert that a functional version of the old PR-1 program was created. That is, ... an almost functional version, written in Scheme, using the built-in random function of Scheme (which is not a pure function because it is using a hidden state). The program has been used and approved by the composers in the team.

As a moderately complex real-world program, this almost functional Scheme version of PR-1 is well suited to bring our proposed plumbing methods to the test. We have translated PR-1 into Haskell and into Prolog, measuring the practical impact of plumbing "real functional" random numbers. An added benefit of our translation effort is that the PR-1 program is now accessible to a larger audience. We have also provided a midi backend, which allows for the ready audition of compositions and variations thereof.

We have implemented three versions of PR-1. All three follow the original Scheme version as closely as possible. The first translation results in a naive Haskell version in which a function random is called that always returns the same value. Replacing the 21 invocations of this random function by several fixed choices we were able to verify that the output of the translated version is identical to the original. This gives confidence in the correctness of the translation.

[1]Henk Barendregt, Aad te Bokkel, Gerhard Eckel, KarlHeinz Essl, Michael Fares, Ramon Gonzalez-Arroyo, Gerhard Koenig, Markus Lepper, Thomas Neuhaus, Albert Nijhof, Dirk Reith, Rutger Teunissen, Luuk Trip, Serge Verstockt, and Wim Vree.

```
mk_row_str par_list = perm par_list : mk_row_str par_list
mk_row_str par_list rg = perm par_list rg' : mk_row_str par_list rg''
                         where
                         (rg', rg'') = split rg
```

Figure 1: Plumbing with splitting generators in Haskell (plumbing in italic font)

```
mk_row_str par_list = perm par_list : mk_row_str par_list
mk_row_str par_list rs = res' : mk_row_str par_list rs'
                         where
                         (res', rs') = perm par_list rs
```

Figure 2: Plumbing with random streams in Haskell (plumbing in italic font)

Next, real random numbers were introduced in the naive version. We present two approaches, one with generators and one with streams.

3.1. **Plumbing with generators.** Haskell has a type class called RandomGen, which provides two functions next, and split:

```
class RandomGen g where
  next :: g → (Int, g)
  split :: g → (g, g)
```

The first function generates the next random number and a new generator to be used when more random numbers are needed. The second function split is supposed to return two new independent generators. Repeated splitting would create an arbitrary number of independent generators.

An example function from the PR-1 program is shown in Figure 1. The first row shows the naive version, and the second row shows the version with plumbing.

The function mk_row_str computes a stream of permutations of a musical parameter: par_list. To compute a permutation, a certain number of random numbers are needed. Therefore the function perm gets a fresh generator rg'. The recursive invocation of mk_row_str continues to use the other generator rg''. The plumbing in Figure 1 shows that only input parameters have been added. The output of all functions remains unchanged. Still, a complete new **where** clause had to be added to unpack the tuple returned by split.

3.2. **Plumbing with streams.** Another possibility to insert random numbers is to use a stream of random numbers as an extra parameter. Figure 2 shows the same example as above, but now with stream plumbing.

We have added a stream of random numbers, rs, to the input parameter of perm and also require that perm returns the rest of the random stream in addition to its old result. This is the sub-expression lifting discussed in Section 2.

```
mk_row_str(Par_list, [Row|Row_str])
 ← perm(Par_list, Row),
    freeze(Row_str, mk_row_str(Par_list, Row_str)).
mk_row_str(Par_list, [Row|Row_str], Rs′)
 ← perm(Par_list, Row, Rs′, Rs″),
    freeze(Row_str, mk_row_str(Par_list, Row_str, Rs″)).
```

Figure 3: Plumbing with streams in Prolog (plumbing in italic font)

3.3. **Plumbing in Prolog.** To support our claim that the logical style is superior to the functional style when it comes to plumbing, we have repeated the plumbing of PR-1 in Prolog. Again we follow the Scheme original as closely as possible, which results in a naive Prolog version where the predicate random always returns the same value. Next we insert stream plumbing and while doing so we note another advantage of logic programming: A plumbed predicate, with extra in- and output parameters can coexist with its original version because in Prolog two predicates with the same name but different arity are considered distinct. This means that a small part of the program can already use the new plumbed version while the rest of the program still uses the old one. This allows for incremental development of the plumbing. In Haskell we had to invent new names for the plumbed functions while the old functions were still in use.

The main advantage of Prolog is the symmetry of in- and output parameters. One can just as easily insert extra output parameters as one can insert extra input parameters. Our running example is shown in Figure 3, again with the naive version in first row and the plumbed version in the second row:

A striking difference between the Prolog and Haskell versions is the predicate freeze. This predicate is used to avoid the eager evaluation of the recursive call to mk_row_str. Evaluation is suspended until the variable Row_str is instantiated somewhere else. This is called "coroutining" in Prolog. Although it is a distinct disadvantage of Prolog compared to the ease of programming that lazy evaluation offers, it is beside the point that we are making. It is not about plumbing, it's about evaluation order.

What we do have to observe is the regular and symmetric addition of the in- and output parameters: Rs', and Rs''. Compared to the functional version no expression lifting is needed and no tuples are packed or unpacked. All other predicates were plumbed in precisely the same regular way.

3.4. **Comparing the plumbing.** To quantify our plumbing effort in the three PR-1 versions we have counted the number of lines of code that had to be changed. Table 1 summarizes these numbers.

The first entry in the table reports the total number of lines of code without comments. The Prolog version is longer than the Haskell version. The difference is, however, mainly in the Prolog specific IO part of the code, not in the part that we have changed by plumbing.

The second row in Table 1 shows the number of lines of the core of the program, i.e. without code concerned with pretty printing the composition and without small utility functions (predicates). This is the part of the code in which we have counted the plumbing. The difference in length between Prolog and Haskell is much less pronounced.

The third row in Table 1 reports the total number of lines that have been changed or added by plumbing. The numbers indicate that there is not much difference between the

	Haskell generator	Haskell stream	Prolog stream
total lines of code	378	378	436
lines without IO and utilities	271	271	297
lines plumbed	84	90	86
of which: tuples added	22	16	-
tuples extended	0	37	-
random numbers used	21	21	21

Table 1: Plumbing comparison

three approaches. However, we can further refine the data by distinguishing between three types of plumbing:

- the addition of a parameter to an existing list of parameters
- the addition of a new line of code to deal with the unpacking of a new tuple (subexpression lifting)
- the extension of an existing tuple (adding output to an already lifted expression)

The following rows of Table 1 show the considerable amount of restructuring of the functional program that is caused by packing and unpacking tuples, either new or existing. These changes are more complex and error prone than the mere addition of parameters. In Prolog no restructuring is needed, and no expressions have to be lifted.

3.5. **The Music.** At this stage the reader may wonder what kind of music PR-1 produces. To satisfy their curiosity we reproduce our composition "De loodgieters" in Figure 4. The reader can listen to the composition by clicking on this link: http://www.cs.ru.nl/barendregt60/essays/hartel_vree/midi/loodgieters.mp3.

Serial music, with its many dissonant chords, irregular melodies and rhythms is difficult to appreciate for the untrained ear. We have tried to make our composition closer to "normal" music by tuning the PR-1 parameters in a specific way. First we restrict the durations to just a few different values between 1/2 and 1/16th of a beat. Next we use regular textures for chords and durations. And finally we base the serial tone rows on the two most common intervals: major (C-E) and minor (C-E♭).

There is a second version http://www.cs.ru.nl/barendregt60/essays/hartel_vree/midi/dynbnd.mp3 that demonstrates accidental name capture. We have simulated accidental name capture by replacing the 21 calls to the function random by fixed values. The result now definitely shows too much regularity and too many repetitions.

4. Conclusions

Why do engineers worry about details? Because ignoring even a small detail may cause a disaster. For example a magnificent piece of engineering such as the Ariane space rocket ultimately failed because of something so mundane as arithmetic overflow [6]. To the best of our knowledge (implementations) of the λ-calculus have not been blamed for multi million dollar disasters but this does not mean that we can ignore the details.

In a purely functional programming style, providing fresh names, or providing fresh random numbers is a tedious detail. In the logical style this is less tedious. We have shown that the most common methods of providing fresh names in the functional style require non trivial restructuring of the program whereas no restructuring is needed in the logical style.

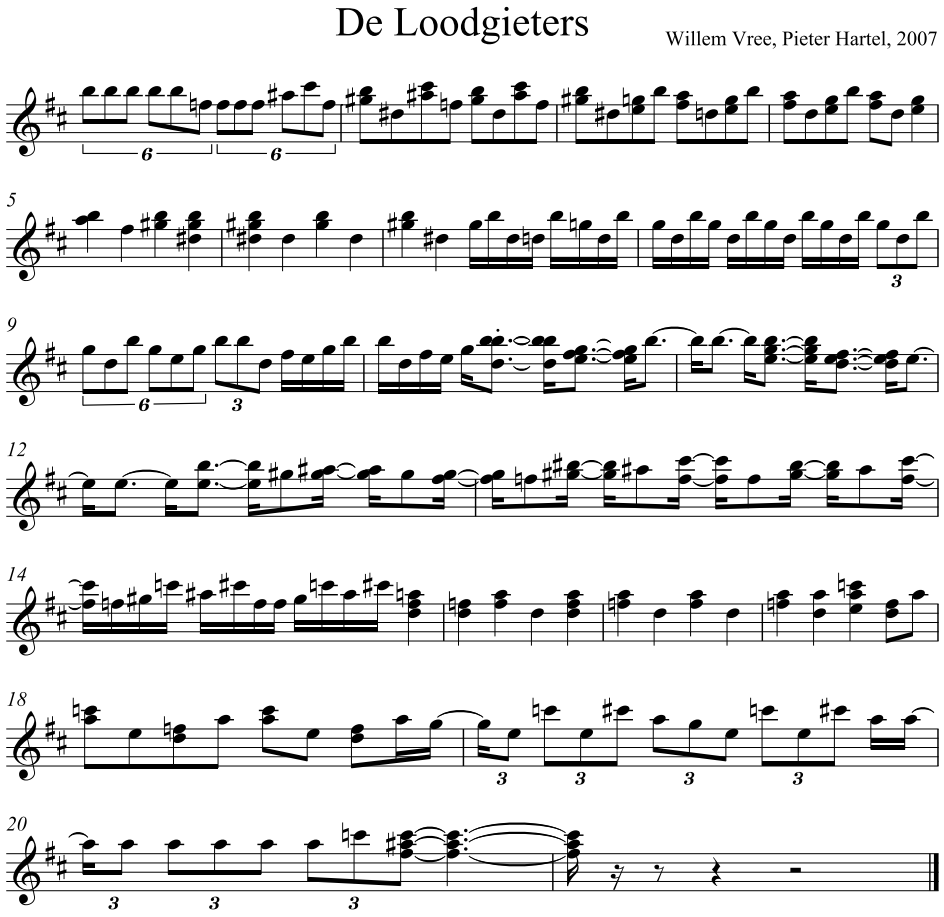

Figure 4: De Loodgieters with parameter settings 1,7,6

The tedium of providing fresh names in the functional style can be relieved somewhat by permitting the use of hidden stateful computations, which gives impure languages like SML, Scheme and Prolog an advantage. The disadvantage is the loss of referential transparency. After so many years of research the main challenge in this area is still to find an elegant solution that would relieve the tedium of plumbing name or random number supplies without having to resort to impure solutions.

We have analysed the plumbing problem using a small example, the β substitution rule of the λ-calculus, and a larger example, the PR-1 music composition program. Using the latter, we have been able to illustrate audibly the effect of a common problem found in the implementation of the former.

The analysis of PR-1 confirms our claim that providing random numbers in the logical style requires no restructuring. In the functional style about seven percent of the code had to be restructured.

We have shown that inspiration to cope appropriately with mundane details can be found in mundane engineering activities such as installing a central heating. We also hope to have shown that engineers and logicians can work together in harmony, and that they can have a lot of fun together as well!

Acknowledgements

We thank Sandro Etalle, Jan Kuper, and the anonymous referees for their comments on the paper.

References

[1] L. Augustsson, M. Rittri, and D. Synek. On generating unique names. *J. of Functional Programming*, 4(1):117–123, Jan 1994.

[2] H. P. Barendregt. *The lambda calculus, its syntax and semantics*. North Holland, Amsterdam, Amsterdam, The Netherlands, 1984.

[3] R. J. Boulton. Generating embeddings from denotational descriptions. In J. Grundy and M. Newey, editors, *11th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, volume LNCS 1479, pages 67–86, Canberra, Australia, Oct 1998. Springer-Verlag, Berlin.

[4] J. Cheney. Scrap your nameplate (functional pearl). In *10th Int. Conf. on Functional programming (ICFP)*, pages 180–191, Tallinn, Estonia, Sep 2005. ACM Press, New York.

[5] J. Cheney and C. Urban. αprolog: A logic programming language with names, binding and a-equivalence. In B. Demoen and V. Lifschitz, editors, *20th Int. Conf. on Logic Programming (ICLP)*, volume LNCS 3132, pages 269–283, Saint-Malo, France, Sep 2004. Springer, Berlin.

[6] M. Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84, Mar 1997.

[7] M. J. Grant. *Serial Music, Serial Aesthetics Compositional Theory in Post-War Europe*. Cambridge University Press, Jun 2005.

[8] G. M. Koenig. Project one: A program for music composition. *Electronic Music Reports*, 2:32–44, 1970.

[9] O. Laske. Composition theory in koenig's project one and project two. *Computer Music Journal*, 5(4):54–65, Winter 1981.

[10] J. Launchbury and S. L. Peyton Jones. State in haskell. *LISP and Symbolic Computation*, 8(4):293–341, Dec 1995.

[11] S. L. Peyton Jones. *The implementation of functional programming languages*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[12] S. L. Peyton Jones and S. Marlow. Secrets of the glasgow haskell compiler inliner. *J. of Functional Programming*, 12(4+5):393–434, Jul 2002.

[13] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. Freshml: programming with binders made simple. In *8th Int. Conf. on Functional programming (ICFP)*, pages 263–274, Uppsala, Sweden, Aug 2003. ACM Press, New York.

[14] P. L. Wadler. The essence of functional programming. In *19th Principles of programming languages (POPL)*, pages 1–14, Albuquerque, New Mexico, Jan 1992. ACM Press, New York.