FUNCTIONAL PEARL
Type-Preserving Renaming and Substitution

CONOR MCBRIDE
University of Nottingham

Abstract

I present a substitution algorithm for the simply-typed λ-calculus, represented in the style of Altenkirch and Reus (1999) which is statically guaranteed to respect scope and type. Moreover, I use a single traversal function, instantiated first to renaming, then to substitution. The program is written in Epigram (McBride & McKinna, 2004).

1 Introduction

In this paper, I give a small but indicative example of programming with *inductive families of datatypes* (Dybjer, 1991) in the dependently typed functional programming language, Epigram (McBride & McKinna, 2004). I present *type-preserving* renaming and substitution for the *type-correct* representation of the simply-typed λ-calculus given by Altenkirch and Reus (1999). I should draw attention to two aspects of this program:

- Renaming and substitution turn out to be instances of a *single* traversal operation, pushing functions from variables to 'stuff' through terms, for a suitable notion of 'stuff'. This traversal operation is structually recursive, hence clearly total.
- Type preservation is clearly promised by the types of these programs; in their bodies, the amount of syntax required to fulfil this promise is *none whatsoever*. The ill-concealed ulterior motive of this paper is to put Epigram's notational innovations through their paces.

The only significant cosmetic treatment I have given the code is to delete a few brackets and make some of the operators infix: the honest ASCII thus looks a little clumsier. I suppressed no details: the process usually referred to vaguely as 'type inference' is hard at work here, inferring *values* determined by the dependent types in which they occur.

2 Simply Typed λ-Calculus

Figure 2 gives the now traditional definition of the type-correct simply-typed λ-terms in Epigram's two-dimensional syntax. The natural deduction style emphasises

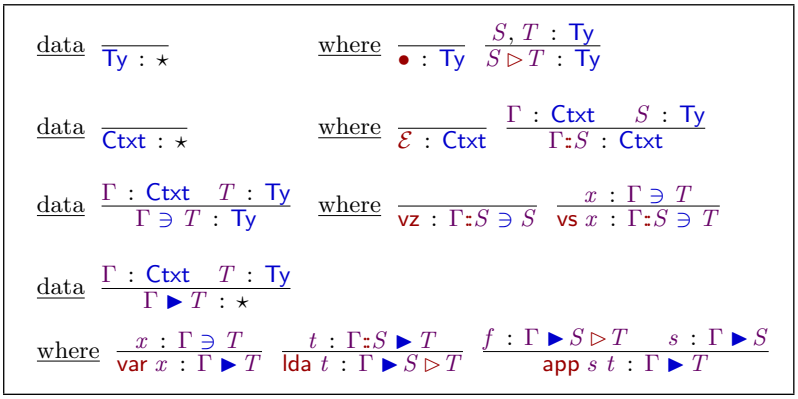

Fig. 1. The Simply-Typed λ-Calculus

the connection between inductive families of datatypes and deduction systems. Each rule types the general usage of a new symbol, below the line, in terms of parameters typed above the line. You can read ':' as 'has type' and '$\star$' as the type of types.

This may seem like overkill for 'simple' definitions like Ty (for simple types) and Ctxt (for contexts—reversed lists of simple types), which you might imagine writing grammar-style, like this:

$$\begin{array}{rl} \underline{\mathsf{data}}\ \mathsf{Ty} & = \bullet \mid \mathsf{Ty} \triangleright \mathsf{Ty} \\ \underline{\mathsf{data}}\ \mathsf{Ctxt} & = \mathcal{E} \mid \mathsf{Ctxt} \mathbin{:\!\!\cdot} \mathsf{Ty} \end{array}$$

The production $\mathsf{Ty} \triangleright \mathsf{Ty}$ makes perfect sense in a language where types and values are rigidly separated, but in Epigram it's actually a well-formed but ill-typed application! Our notation invites the programmer to write a set of *patterns* for data, naming their parts: these patterns and these names reappear when you edit interactively, as the machine generates exhaustive case analyses on the left-hand sides of programs.

Naming becomes practically indispensable once types start to depend on values. Moreover, *inductive families of datatypes* assign individual return types for each constructor, an unconventional practice which again leads us away from the conventional notation.

Here, the '$\ni$' family presents *variables* as an inference system for context membership. This representation amounts to de Bruijn indices (de Bruijn, 1972), but correctly typed and scoped. Meanwhile, the '$\blacktriangleright$' family presents simply-typed terms via their typing rules, extending the context under a lda and policing the compatility of domain and argument in an app. Note that '$\mathbin{:\!\!\cdot}$' and '$\triangleright$' bind more tightly than '$\ni$' and '$\blacktriangleright$'.

The form of a rule's conclusion quietly specifies which arguments are to be kept implicit and which should be shown. In the data constructors for '$\ni$' and '$\blacktriangleright$', you'll find Γ, S and T undeclared—their types are inferrable from usage by standard techniques (Damas & Milner, 1982). Moreover, the natural deduction rule serves like 'let' in the Hindley-Milner system to indicate the point at which variables

should be generalised where possible. In effect, we may omit declarations for an *initial* segment of parameters to a rule, provided their types are inferrable.

What has happened? The usual alignment of the implicit-versus-explicit with type-versus-value is so traditional that you almost forget it's a design choice. Dependent types make that choice untenable, but it's not the end of the world.

3 Renaming and Substitution, Together

Renaming and substitution are both term traversals, lifting an operation on variables structurally to the corresponding operation on terms. Each must perform an appropriate lifting to push an operation under a lda. Where these operations differ is in the image of variables: renamings map variables to variables and susbtitutions map variables to terms. Here, I abstract the pattern, showing how to traverse terms, mapping variables to any stuff which supports the necessary equipment. What's stuff? It's a type family '$\blacklozenge$' indexed by Ctxt and Ty. What's the necessary equipment? Here it is:

$$\underline{\text{data}}\ \frac{\dfrac{G : \mathsf{Ctxt} \qquad T : \mathsf{Ty}}{G \blacklozenge T : \star}}{\mathsf{Kit}\ (\blacklozenge) : \star}$$

$$\underline{\text{where}}\ \frac{(\blacklozenge) \qquad \dfrac{x : \Gamma \ni T}{vr\ x : \Gamma \blacklozenge T} \qquad \dfrac{i : \Gamma \blacklozenge T}{tm\ i : \Gamma \blacktriangleright T} \qquad \dfrac{i : \Gamma \blacklozenge T}{wk\ i : \Gamma{:}S \blacklozenge T}}{\mathsf{kit}\ vr\ tm\ wk : \mathsf{Kit}\ (\blacklozenge)}$$

We need '$\blacklozenge$' to support three things: a mapping in from the variables, a mapping out to the terms and a weakening map which extends the context. Renaming will instantiate '$\blacklozenge$' with '$\ni$'; for substitution, we may choose 'Tm' instead. Now we need to show how to traverse terms with any Kit, and how to build the Kits we need.

By the way, you may have noticed that I have nested natural deduction rules in order to declare parameters which themselves have functional types. Epigram has 'hypothetical hypotheses' hereditarily.As before, these rules indicate points where variables should be generalised where possible. Correspondingly, it is sometimes necessary to insert an (untyped) declaration at an outer level, in order to suppress generalisation at an inner level. For example, in the declaration of kit, it's important that each operation is general with respect to contexts and types, but they should all apply to a fixed instance of '$\blacklozenge$', hence its explicit declaration.

How do we traverse a term, given a Kit? In general, we have a type-preserving map τ from variables over context Γ to stuff over Δ. We can push that map through

terms in a type-preserving way as follows:

$$\underline{\mathsf{let}}\ \frac{K\ :\ \mathsf{Kit}\ (\blacklozenge) \qquad \Gamma, \Delta \qquad \dfrac{x\ :\ \Gamma \ni X}{\tau\ x\ :\ \Delta \blacklozenge X} \qquad t\ :\ \Gamma \blacktriangleright T}{\mathbf{trav}\ K\ \tau\ t\ :\ \Delta \blacktriangleright T}$$

$$\begin{array}{l}
\mathbf{trav}\ K\ \tau\ t \Leftarrow \underline{\mathsf{rec}}\ t\ \{ \\
\quad \mathbf{trav}\ K\ \tau\ t \Leftarrow \underline{\mathsf{case}}\ t\ \{ \\
\quad\quad \mathbf{trav}\ K\ \tau\ (\mathsf{var}\ x) \Leftarrow \underline{\mathsf{case}}\ K\ \{ \\
\quad\quad\quad \mathbf{trav}\ (\mathsf{kit}\ vr\ tm\ wk)\ \tau\ (\mathsf{var}\ x)\ \Rightarrow\ tm\ (\tau\ x)\ \} \\
\quad\quad \mathbf{trav}\ K\ \tau\ (\mathsf{lda}\ t')\ \Rightarrow\ \mathsf{lda}\ (\mathbf{trav}\ K\ (\mathbf{lift}\ K\ \tau)\ t') \\
\quad\quad \mathbf{trav}\ K\ \tau\ (\mathsf{app}\ f\ s)\ \Rightarrow\ \mathsf{app}\ (\mathbf{trav}\ K\ \tau\ f)\ (\mathbf{trav}\ K\ \tau\ s)\ \}\}
\end{array}$$

Epigram programs are tree-structured. The nodes, marked with '⇐' symbols (pronounced 'by'), explain how to refine the problem of delivering an output from the inputs, by invoking 'eliminators' which specify problem-decomposition strategies, such as structural recursion or case analysis. The leaves, marked with '⇒' symbols (pronounced 'return'), indicate the output which the program should produce in a given case. Informally, you can imagine that the program only consists of leaves, defined by pattern matching. More formally, the program is checked with respect to its eliminators—each of the case nodes is exhaustive, and the recursive calls are checked to be structural with respect to the parameter indicated in the rec node. This program is thus seen to be total.

In the var case, our map τ gives us some stuff, which we can turn into a term with some help from our kit. The other two cases go with structure, but we shall need to **lift** τ to source and target contexts extended by a bound varaible, in order to push it under a binder—we shall see how to do this in a moment. The rules of the simply-typed λ-calculus are respected without a squeak!

Note that the 'patterns' to the left of '⇐' or ' ⇒ ' were not written by me, but by the *editor*, provoked by my choices of eliminator. The notation may be a touch verbose, but the effort involved is less than usual. This somewhat austere notation allows for the possibility of *user-defined* eliminators, rather than rec and case, a possibility explored more fully in 'The view from the left' (McBride & McKinna, 2004), but it could readily be tuned to privilege normal behaviour, suppressing case eliminators inferrable from the constructor symbols in patterns.

But I digress, when I should be writing **lift**. This just maps the new variable to itself (or rather, its representation as 'stuff'), and each old variable to the weakening of its old image.

$$\underline{\mathsf{let}}\ \frac{K\ :\ \mathsf{Kit}\ (\blacklozenge) \qquad \Gamma, \Delta \qquad \dfrac{x\ :\ \Gamma \ni X}{\tau\ x\ :\ \Delta \blacklozenge X} \qquad x\ :\ \Gamma{:}{:}S \ni T}{\mathbf{lift}\ K\ \tau\ x\ :\ \Delta{:}{:}S \blacklozenge T}$$

$$\begin{array}{l}
\mathbf{lift}\ K\ \tau\ x \Leftarrow \underline{\mathsf{case}}\ K\ \{ \\
\quad \mathbf{lift}\ (\mathsf{kit}\ vr\ tm\ wk)\ \tau\ x \Leftarrow \underline{\mathsf{case}}\ i\ \{ \\
\quad\quad \mathbf{lift}\ (\mathsf{kit}\ vr\ tm\ wk)\ \tau\ \mathsf{vz} \qquad \Rightarrow vr\ \mathsf{vz} \\
\quad\quad \mathbf{lift}\ (\mathsf{kit}\ vr\ tm\ wk)\ \tau\ (\mathsf{vs}\ x) \Rightarrow wk\ (\tau\ x)\ \}\}
\end{array}$$

From here, renaming and substitution are easy! We just need to construct the kits for '$\ni$' and '$\blacktriangleright$' respectively.

$$\underline{\text{let}}\ \frac{\Gamma, \Delta \qquad \dfrac{x \mathrel{:} \Gamma \ni X}{\rho\ x \mathrel{:} \Delta \ni X} \qquad t \mathrel{:} \Gamma \blacktriangleright T}{\mathbf{rename}\ \rho\ t \mathrel{:} \Delta \blacktriangleright T}$$

$$\mathbf{rename}\ \rho\ t \Rightarrow \mathbf{trav}\ (\mathsf{kit}\ \mathbf{id}\ \mathsf{var}\ \mathsf{vs})\ \rho\ t$$

The identity function makes variables from variables; the var constructor takes variables to terms; the vs constructor weakens each variable into an extended context. Meanwhile, substitution goes like this:

$$\underline{\text{let}}\ \frac{\Gamma, \Delta \qquad \dfrac{x \mathrel{:} \Gamma \ni X}{\sigma\ x \mathrel{:} \Delta \blacktriangleright X} \qquad t \mathrel{:} \Gamma \blacktriangleright T}{\mathbf{subst}\ \sigma\ t \mathrel{:} \Delta \blacktriangleright T}$$

$$\mathbf{subst}\ \sigma\ t \Rightarrow \mathbf{trav}\ (\mathsf{kit}\ \mathsf{var}\ \mathbf{id}\ (\mathbf{rename}\ \mathsf{vs}))\ \sigma\ t$$

That is, var makes terms from variables, **id** takes terms into terms, and a term is weakened by *renaming* with vs.

4 Conclusion and Further Work

References

Altenkirch, Thorsten, & Reus, Bernhard. (1999). Monadic presentations of lambda-terms using generalized inductive types. *Computer Science Logic 1999*.

Damas, Luis, & Milner, Robin. 1982 (January). Principal type-schemes for functional programming languages. *Pages 207–212 of: Ninth annual symposium on principles of programming languages (popl) (albuquerque, nm)*. ACM.

de Bruijn, Nicolas G. (1972). Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes mathematicæ*, **34**, 381–392.

Dybjer, Peter. (1991). Inductive Sets and Families in Martin-Löf's Type Theory. Huet, Gérard, & Plotkin, Gordon (eds), *Logical Frameworks*. CUP.

McBride, Conor, & McKinna, James. (2004). The view from the left. *Journal of functional programming*, **14**(1).