JFP, 5 pages, 2022.
doi:10.1017/xxxxx

How to do something relevant for a change

CONOR MC BRIDE
Mathematically Structured Programming Group,
Department of Computer and Information Sciences, University of Strathclyde, G1 1XH
(e-mail: conor.mcbride@strath.ac.uk)

Abstract

This paper explores a particular manner of dependently typed programming in which specifications stand in the foreground. Where possible, the program text concerns irrelevant proofs of propositions about relevant existential witnesses which are left implicit and inferred. In other words, the text is the explanation which provokes the mechanical construction of the part of the program that gets executed. The technique is illustrated by the construction of the category of relevant simultaneous substitutions for generic 'codeBruijn' syntax. The latter imposes the invariant that every variable in the scope of a term is used at least once within it; the former requires that every variable in the output scope of a substitution occurs in the image of at least one variable in the input scope; the action of substitution on terms (and on substitutions) thus maintains the invariant. An Agda implementation accompanies all the work presented herein.

1 Introduction

Dependently typed programming in the *intrinsic* style involves the imposition of key structural invariants *a priori* on data and the maintenance of those invariants by operations on the data. Sometimes, the poor benighted typechecker (which can do *arithmetic*, but very little *algebra*) cannot see for itself that the programmer has successfully (re)established an invariant: the peg fits the hole but will not go in without the ministrations of the mallet that is proof.

There is too much tell, here, and not enough show. Perhaps that's inevitable.

Reasoning about such programs is then complicated by reasoning about the proofs which hold them together, often requiring rather tight coordination between the proofs *in* the programs and the proofs *about* the programs, and so on, up the razor blade of meta. The 'with'-abstraction feature of dependently typed languages, abstracting the value of an expression of interest from all types present and offering its value for inspection, provides a basic coordination mechanism, but sometimes the coupling is just too tight and the consequent game of 'with-jenga' too macho. Let us make the game easier by loosening the coupling, rather than increasing the cleverness with which we play it.

Let me briefly sketch the prospectus of this paper by way of furnishing some sort of example. We shall be working in a multisorted syntax where scope is explicitly managed. Scopes Γ, Δ will be snoc-lists of the sorts S, T of the free variables. More particularly, *relevance* will be managed. There will be a notion of relevant *term* $S \dashv \Gamma$ requiring that every variable in Γ occurs at least once. We shall further have a notion of relevant *substitution*

$\Gamma \Rightarrow \Delta$ such that every variable in Δ occurs in the image of at least one variable from Γ. It should then be possible to construct an operation of type

$$S \dashv \Gamma \rightarrow \Gamma \Rightarrow \Delta \rightarrow S \dashv \Delta$$

as the images of the Γ-variables are all substituted at least once, and every Δ-variable occurs in at least one image.

Wander we must, inward through the structure of the input, and we shall sometimes encounter *pairing* constructs: relevance requires that every variable in scope occurs on the left or on the right or both. That is we must make two selections, $\theta_l : \Gamma_l \leq \Gamma$ and $\theta_r : \Gamma_r \leq \Gamma$ which collectively form a *cover*. We say $\theta_l \cup \theta_r$ if every variable in Γ is in Γ_l or Γ_r or both. Our *relevant* pair will thus consist of two subterms in $S_l \dashv \Gamma_l$ and $S_r \dashv \Gamma_r$, respectively, together with a proof that $\theta_l \cup \theta_r$.

How are we to push substitution $\sigma : \Gamma \Rightarrow \Delta$ into these subterms? If have only variables in Γ_l, then we need only the part $\sigma_l : \Gamma_l \Rightarrow \Delta_l$ which gives the images of those variables, and there must be some $\phi_l : \Delta_l \leq \Delta$. Similarly, we must have $\sigma_r : \Gamma_r \Rightarrow \Delta_r$ and $\phi_r : \Delta_r \leq \Delta$. To reestablish the relevance of the substitution images, we shall need to show that $\phi_l \cup \phi_r$ whenever $\theta_l \cup \theta_r$. That is, if the two components covered the input variables, then their substitution images cover the output variables. Given such a proof, we may pair our $S_l \dashv \Delta_l$ and $S_r \dashv \Delta_r$.

The data $\sigma_l, \Delta_l, \phi_l$ are computed from the $\Gamma_l, \theta_l, \Gamma, \sigma, \Delta$ and likewise on the right, and the proof that $\phi_l \cup \phi_r$ is essential to the substitution operation. Proving properties of the substitution operation thus, apparently, necessitates negotiating the behaviour of these auxiliary computations and the key covering proof. How are we to show that the action of composed substitutions is the composition of the action of substitutions with a gnarly knot like this?

The key strategy to ameliorate these woes is *wishful thinking*. We can write relational specifications for our operations which *presuppose* all the coherence properties necessary to make the types fit together properly. Our specification of substitution will presume that the computed and proven data arrive as if by magic. We will say what 'diagrams' we need without specifying how they are to be computed. Our metatheorems will be entirely diagram-pasting. Meanwhile, our implementations will be proofs of diagram *completion*.

We shall, for example, express what it is to be a square x_l

$$\begin{array}{ccc} \Gamma & \sigma & \Delta \\ \theta_l & x_l & \phi_l \\ \Gamma_l & \sigma_l & \Delta_l \end{array}$$

making a *double category* with vertical edges in $\leq^{\mathrm{op}}$ and horizontal edges in $\Rightarrow$. To substitute through a pair, it enough to have *somehow* acquired left and right squares x_l and x_r, together with coverings $\theta_l \cup \theta_r$ and $\phi_l \cup \phi_r$. We can *reason* about relevant substitution assuming we have all of these pieces and that they all fit together.

To prove that we can *perform* substitution, we need to show that these squares can be completed from their top left boundaries, and that if the two left boundaries form a covering, then so do the two right boundaries. The proof x_l that the square is valid can be presented as data but is entirely *propositional* — there is at most one way such a square can be valid, and the code will be explicit about that explanation; the useful, bit-bearing data are the σ_l, Δ_l, ϕ_l which constitute the rest of the boundary, and these the program text will suppress.

We thus begin to subvert what I have sometimes called 'Milner's Coincidence'. In ML, the program text is the program that is executed, and the types are inferred then erased. Here, the program text is the explanation which can be erased at run time: its purpose is to provoke the inference of a correct program that is retained for execution. That is to say, by putting Milner's idea of unification-based inference to work on the program as well as the types, we acquire a formalism for correct program calculation.

2 Relevant syntax

Let us begin at the beginning, which is to introduce a notion of multi-sorted syntax with binding, in the style of universal algebra. The whole setup is parametrised over some $\mathit{Sort} : \mathbf{Set}$ of *term* sorts, to be defined by mutual induction. We shall need to explain how to build terms from collections of subterms. So let us introduce a notion of *arity*, $\mathbf{Ar}\,\mathit{Sort}$, so that we can also have a parameter specifying the *constructors* $\mathit{Ctor} : \mathit{Sort} \rightarrow \mathbf{Ar}\,\mathit{Sort} \rightarrow \mathbf{Set}$, such that $\mathit{Ctor}\ S\ T$ tells us which constructor tags make a term of sort S with arity T. All $\mathbf{Ar}$ does is close Sort under unit, pairing and abstraction:

$$\frac{S : \mathit{Sort}}{S : \mathbf{Ar}\,\mathit{Sort}} \qquad \frac{}{1 : \mathbf{Ar}\,\mathit{Sort}} \qquad \frac{T, U : \mathbf{Ar}\,\mathit{Sort}}{T \times U : \mathbf{Ar}\,\mathit{Sort}} \qquad \frac{T, U : \mathbf{Ar}\,\mathit{Sort}}{T \rhd U : \mathbf{Ar}\,\mathit{Sort}}$$

For example, the untyped lambda calculus might be given by taking $\mathit{Sort} = \mathbf{1}$ (the unit type, with $\langle\rangle : \mathbf{1}$) and having

$$\mathrm{lam} : \mathit{Ctor}\ \langle\rangle\ (\langle\rangle \rhd \langle\rangle) \qquad \mathrm{app} : \mathit{Ctor}\ \langle\rangle\ (\langle\rangle \times \langle\rangle)$$

Scopes are snoc-lists of sorts:

$$\frac{}{\varepsilon : X^*} \qquad \frac{xz : X^* \quad x : X}{xz, x : X^*}$$

Were we to introduce a *de Bruijn* syntax, we should have terms

$$\frac{T : \mathbf{Ar}\,\mathit{Sort} \quad \Gamma : \mathit{Sort}^*}{T \dashv_{\mathbf{dB}} \Gamma : \mathbf{Set}}$$

with variables, constructors, unit, pairing and binding:

$$\frac{i : S \leftarrow \Gamma}{i : S \dashv_{\mathbf{dB}} \Gamma} \qquad \frac{c : \mathit{Ctor}\ S\ T \quad t : T \dashv_{\mathbf{dB}} \Gamma}{c(t) : S \dashv_{\mathbf{dB}} \Gamma}$$

$$\frac{}{\varepsilon : 1 \dashv_{\mathbf{dB}} \Gamma} \qquad \frac{t : T \dashv_{\mathbf{dB}} \Gamma \quad v : V \dashv_{\mathbf{dB}} \Gamma}{t, v : T \times V \dashv_{\mathbf{dB}} \Gamma} \qquad \frac{t : T \dashv_{\mathbf{dB}} \Gamma, S}{\lambda t : S \rhd T \dashv_{\mathbf{dB}} \Gamma}$$

where $S \leftarrow \Gamma$ is the type of locations of S in Γ, i.e., glorified numbers guaranteed to be in bounds and to index the intended sort.

However, my aim is to be relevant, for a change. We shall have *the* variable, but constructors are as before:

$$\frac{}{\bullet : S \dashv \varepsilon, S} \qquad \frac{c : \mathit{Ctor}\ S\ T \quad t : T \dashv \Gamma}{c(t) : S \dashv \Gamma}$$

Unit and pairing are sensitive to scope. I shall detail $\leq$ and $\cup$ shortly, but they allow us to ensure that the left scope Γ_l and the right scope Γ_r collectively cover the whole scope Γ.

$$\frac{}{\varepsilon : 1 \dashv \varepsilon} \qquad \frac{t : T \dashv \Gamma_l \quad \theta_l : \Gamma_l \leq \Gamma \quad u : \theta_l \cup \theta_r \quad \theta_r : \Gamma_r \leq \Gamma \quad v : V \dashv \Gamma_l}{t \,\langle u \rangle\, v : T \times V \dashv \Gamma}$$

Binding comes in vacuous and relevant variants, so that the unused variable never comes *into* scope:

$$\frac{t : T \dashv \Gamma}{\kappa\, t : S \rhd T \dashv \Gamma} \qquad \frac{t : T \dashv \Gamma, S}{\lambda\, t : S \rhd T \dashv \Gamma}$$

Note that every *scope* $\Gamma : \mathit{Sort}^*$ induces an *arity* $\bar{\Gamma} : \mathbf{Ar}\,\mathit{Sort}$ by tupling:

$$\bar{\varepsilon} = 1 \qquad \overline{\Gamma, S} = \bar{\Gamma} \times S$$

Kindly let me suppress the $\bar{\cdot}$ and just write scopes in places that arities are expected. We thus acquire $\Gamma \dashv \Delta$ as the type of *relevant substitutions*, giving an image over some subscope of Δ for each variable in Γ, so that all variables in Δ get used. Our mission is thus to construct an action

$$\cdot \mathbin{;} \cdot : S \dashv \Gamma \to \Gamma \dashv \Delta \to S \dashv \Delta$$

which induces categorical structure.

3 The category of thinnings

Let me first, however, pick up on my promise to give the details of these ‘thinnings’, such as our $\theta_l : \Gamma_l \leq \Gamma$. These may be thought of as bit vectors which witness the embedding (dually, selection) of a sublist into (from) a list. The 1s in the vector mark the places where the sublist elements go to (come from). They amount to binomial coefficients, generalised from numbers to lists and reified as types.

$$\frac{}{\varepsilon : \varepsilon \leq \varepsilon} \qquad \frac{\theta : \Gamma \leq \Delta}{\theta, 1 : \Gamma, S \leq \Delta, S} \qquad \frac{\theta : \Gamma \leq \Delta}{\theta, 0 : \Gamma \leq \Delta, S}$$

We may compute the ‘all 0s’ and ‘all 1s’ thinnings

$$\bar{0} : \varepsilon \leq \Gamma \qquad \bar{1} : \Gamma \leq \Gamma$$

by iteration over Γ, with $\bar{1}$ giving us a suitable notion of ‘identity thinning’.

What about composition? I *could* define it as a *function*,

$$\cdot \mathbin{;} \cdot : \Gamma \leq \Delta \to \Delta \leq \Theta \to \Gamma \leq \Theta$$

but then I would find myself referring to particular composite thinnings by but one of the compositions $\theta \mathbin{;} \phi$ which might deliver it. If I have a commuting square

$$\begin{array}{ccc} \Gamma & \theta_0 & \Delta_0 \\ \theta_1 & & \phi_o \\ \Delta_1 & \phi_1 & \Theta \end{array}$$

then its diagonal is *both* $\theta_0 \mathbin{;} \phi_0$ and $\theta_1 \mathbin{;} \phi_1$, an equation relating the *outputs* of two computations from which we will learn nothing without inspecting the *inputs*!

The way out of this trouble is to specify commuting triangles *inductively*,

$$\frac{\theta : \Gamma \leq \Delta \quad \phi : \Delta \leq \Theta \quad \psi : \Gamma \leq \Theta}{\theta \mathbin{;} \phi \cong \psi : \mathbf{Set}} \qquad \frac{}{\varepsilon : \varepsilon \mathbin{;} \varepsilon \cong \varepsilon}$$

with a base case and three step cases,

$$\frac{v : \theta \mathbin{;} \phi \cong \psi}{v, 0 : \theta \mathbin{;} \phi, 0 \cong \phi, 0} \qquad \frac{v : \theta \mathbin{;} \phi \cong \psi}{v, 01 : \theta, 0 \mathbin{;} \phi, 1 \cong \phi, 0} \qquad \frac{v : \theta \mathbin{;} \phi \cong \psi}{v, 1 : \theta, 1 \mathbin{;} \phi, 1 \cong \phi, 1}$$

which is of course, the *graph* of the function we might otherwise have implemented. The difference is that dependent pattern matching will allow us to manipulate triangles directly, rather than via their edges or vertices.

Allow me to write $\langle P\ \cdot \rangle$ to abbreviate (constructive) existential quantification '$P\,x$ is inhabited for some x. We may thus write $\langle \theta \mathbin{;} \phi \cong \cdot \rangle$ as the *proposition* that θ and ϕ can be completed to a composition triangle. It is immediate that any two proofs of this proposition are equal, because the relation is the graph of a deterministic function. Moreover, we may readily define (leaving the existential witnesses implicit).

$$\frac{\theta : \Gamma \leq \Delta \quad \phi : \Delta \leq \Theta}{\theta \bigtriangleup \phi : \langle \theta \mathbin{;} \phi \cong \cdot \rangle} \qquad \begin{aligned} \theta \bigtriangleup \phi, 0 &= \theta \bigtriangleup \phi, 0 \\ \theta, 0 \bigtriangleup \phi, 1 &= \theta \bigtriangleup \phi, 01 \\ \theta, 1 \bigtriangleup \phi, 1 &= \theta \bigtriangleup \phi, 1 \\ \varepsilon \bigtriangleup \varepsilon &= \varepsilon \end{aligned}$$