Under consideration for publication in J. Functional Programming

The types who say \ni

CONOR T. MCBRIDE
University of Strathclyde
(*e-mail:* conor.mcbride@strath.ac.uk)

1 Introduction

This paper is about my religion. It introduces a discipline for constructing and validating bidirectional type systems, illustrated with a nontrivial running example — a bidirectional reconstruction of Per Martin-Löf's small and beautiful, but notoriously inconsistent dependent type theory from 1971 (Martin-Löf, 1971). Crucially, the fact that the system is not strongly normalizing is exploited to demonstrate concretely that the methodology relies in no way on strong normalization, which is perhaps peculiar given that bidirectional type systems are often (but not here) given only for terms in β-normal form (Pierce & Turner, 2000).

From the outset, it would seem prudent to manage expectations. I take the view that types are not inherent in things, but rather imposed on them by human design in arbitrary ways. Meaning is made, not found. A practical consequence of this viewpoint is that we may fix a generic syntax, simple and flexible, before giving any thought to the design of types and the meaningful forms of computation they justify. Every self-respecting religion pinches the parts it likes from other religions, so I will choose LISP s-expressions (McCarthy, 1960) as the generic syntax, but tighten up the treatment of variable binding with methods of de Bruijn (de Bruijn, 1972), distinguishing variables from atoms and naming them only in the interests of informal civility. One should not expect things expressible in this syntax to make sense: rather we should design ways to make sense of some of them. We get out what we put in, so let us seek understanding of how to direct our freedom towards virtuous outcomes.

Oft sought properties of type systems, such as 'stability under substitution' and 'type preservation', are not to be had for the proving, but rather by construction in accordance with good guidance. The prospectus of this paper is to develop a metametatheory in which to ensure the good metatheoretic properties of a whole class of theories.

2 A Generic Syntax for Bidirectional Type Systems

For the benefit of human beings, a *scope* is written as a list of identifiers $x_{n-1}, \ldots x_0$, although the inhuman truth is that it is just the number n which gives the length of the sequence. I refer to scopes by metavariables γ and δ, with ε denoting the empty scope. A *variable* is an identifier x_i selected from a scope, serving as the human face of its index i. I specify grammars relative to an explicit scope, γ and write $x_i \leftarrow \gamma$ to mean 'a variable from γ'.

Unlike variables, *atoms* (a, A) really are named global constants which play the role of constructors — their purpose is not to stand for things but to be told apart. The symbol $()$ is considered an atom and pronounced 'nil'.

The object-level syntax is specified, written as a subscript, and is divided into four distinct mutually defined grammatical classes, each with standard metavariable conventions, arranged as four quadrants thus:

	essential	**liberal**
constructions	k_γ, K_γ $::= \quad a$ $\quad \mid \quad (s_\gamma . t_\gamma)$ $\quad \mid \quad \backslash x\, t_{\gamma,x}$	$s_\gamma, t_\gamma, S_\gamma, T_\gamma$ $::= \quad k_\gamma$ $\quad \mid \quad [n_\gamma]$
computations	n_γ, N_γ $::= \quad x \leftarrow \gamma$ $\quad \mid \quad e_\gamma\, s_\gamma$	$e_\gamma, f_\gamma, E_\gamma, F_\gamma$ $::= \quad n_\gamma$ $\quad \mid \quad (k_\gamma : T_\gamma)$

The meaningfulness of *constructions* will always be *checked*, relative to some prior expectation. The *essential* constructions give us the raw materials for canonical values, and they will always be invariant under computation. Let us adopt the LISP convention that matching parentheses preceded by a dot may be deleted, along with the dot, which is conveniently prejudicial to right-nested, nil-terminated lists. Our constructions will often be such lists, with an atom at the head. For example, dependent function types in γ will have form

$$(\Pi\, S_\gamma\, \backslash x\, T_{\gamma,x})$$

where Π is an atom.

The *liberal* constructions extend the canonical constructions with *thunks* — essential computations which have not yet achieved canonical form, either because they have not yet reduced, or because a variable is impeding reduction.

Meanwhile, the *computations* will always admit a type synthesis process. The *essential* constructions comprise variables (whose type will be assigned by a context) and eliminations, overloading the application syntax — the computation e to the left is being eliminated, and its synthesizable type will tell us how to check that the construction s to its right is a valid eliminator. These two give us the traditional forms of *neutral* term.

The *liberal* computations extend the neutral computations with *radicals*, in the chemical sense, being canonical forms with a type annotation which gives the information needed to check both the canonical form it annotates and the eliminator it is 'reacting' with. If we exclude radicals from this syntax, we obtain the *normal forms*. Radicals are permitted only in eliminations, and these are the only eliminations which compute. That is, there is no computation step which proceeds uninformed by the type at which computation is happening.

Thunks and radicals form the connection between constructions and computations, but you will notice that they carefully exclude one another — there is syntax neither for a thunked radical nor a radical thunk. We may extend these term formers to thunk liberal

computations and radicalise liberal constructions by defining

$$[(k : T)] = k \qquad ([n] : T) = n$$

with the impact of deleting exactly the type annotations which inform no computation, either because they mark a canonical form which is not being eliminated or because they mark a neutral form whose elimination is necessarily stuck. Observe that, as a consequence,

$$[(t : T)] = t$$

whether t is canonical or not. Observe also that we may now lift *all* the term formers to act on liberal components, yielding liberal results, for everything apart from thunks and radicals admits liberal substructures. In particular, it becomes reasonable to substitute a liberal computation for a variable, in either a construction or a computation, yielding a liberal construction or computation, respectively.

With this apparatus in place, we may reformulate the traditional β-rule for functions as a rewriting rule on liberal computations, thus:

$$(\backslash x\, t : (\Pi\, S\, \backslash x\, T))\, s \rightsquigarrow (t/(s : S) : T/(s : S))$$

where t/e denotes substitution for the bound variable. If it so happens that s is canonical, then we will be substituting a radical for the bound variable, perhaps creating redexes. If it so happens that t is canonical, then the whole reduct will be radical. The type annotations thus mark the active computation sites and are discarded whenever computation stops.

3 The Category of Scopes and Thinnings

Definition 1 (*thinning*)
A thinning is an order preserving embedding between scopes

$$\theta : \gamma \sqsubseteq \delta$$

The thinnings are generated by

$$\frac{}{\varepsilon : \varepsilon \sqsubseteq \varepsilon} \qquad \frac{\theta : \gamma \sqsubseteq \delta}{\theta 0 : \gamma \sqsubseteq \delta, x} \qquad \frac{\theta : \gamma \sqsubseteq \delta}{\theta 1 : \gamma, x \sqsubseteq \delta, x}$$

That is, thinnings arise in the manner of Pascal's triangle: there are $\binom{m}{n}$ thinnings from n to m, which is not surprising, as they correspond to selections.

A thinning is given as a bit vector the length of its target scope with its source scope being the 'population count', i.e., number of 1s, in the vector. Thinnings form a well known category: the *semi-simplicial* category, often notated Δ_+.

Definition 2 (*identity thinning*)
The identity thinning $\mathbf{1}_\gamma : \gamma \sqsubseteq \gamma$ is given by

$$\mathbf{1}_\varepsilon = \varepsilon \qquad \mathbf{1}_{\gamma,x} = \mathbf{1}_\gamma 1$$

Informally, we may just write $\mathbf{1}$. The empty thinning, $\mathbf{0}$ is generated analogously, repeating 0 to the appropriate length.

Definition 3 (*composition of thinnings*)
If $\theta : \gamma \sqsubseteq \gamma'$ and $\phi : \gamma' \sqsubseteq \gamma''$, then $\theta;\phi : \gamma \sqsubseteq \gamma''$ defined as follows:

$$\varepsilon;\varepsilon = \varepsilon \qquad \theta;\phi 0 = (\theta;\phi)0 \qquad \theta b;\phi 1 = (\theta;\phi)b$$

Lemma 4 (*category of thinnings*)
We have the usual categorical laws:

$$\mathbf{1};\theta = \theta = \theta;\mathbf{1} \qquad (\theta;\phi);\psi = \theta;(\phi;\psi)$$

Proof
Functional induction on the (graph of) composition readily establishes these results. □

Remark 5 (*thinnings as an integer monoid*)
In another life, I teach undergraduates about computer hardware. Consequently, I recognize the identity thinning as the two's complement representation of -1. Effectively, we may regard the integers as the infinite right-to-left bit vectors which eventually stabilise as all 0 (for non-negative integers) or all 1 (for negative integers). Thinning composition induces a monoid on the integers whose neutral element is -1. The details are left to the curious reader.

Meanwhile, thinnings form a monoidal category by concatenation γ, γ' of scopes and θ, θ' of thinnings.

Definition 6 (*readable notation for variables and thinnings*)
Just as de Bruijn indices are not a suitable notation for human-readable variables, so bit vectors are not a suitable notation for human-readable thinnings. I write $\langle \gamma\text{-}var \rangle$ for the grammar of variables in γ and use names from γ as the valid forms in that grammar. I write $\langle \gamma\text{-}thinning \rangle$ for the grammar of thinnings whose target scope is γ, where

$\langle \gamma\text{-}thinning \rangle$	::=		— $\mathbf{1}$, includes all the variables in γ
	|	$\langle \gamma\text{-}var \rangle \langle \gamma\text{-}var \rangle^*$	— include only the listed variables
	|	$-$	— $\mathbf{0}$, exclude all the variables in γ
	|	$-\langle \gamma\text{-}var \rangle \langle \gamma\text{-}var \rangle^*$	— exclude only the listed variables

It is straightforward to compute the source scope of a $\langle \gamma\text{-}thinning \rangle$, given that we know its target scope: by a slight abuse of notation, I write thinnings θ in places where scopes γ are expected, meaning the source scope that θ effectively selects from some larger target scope.

It is my habit to index grammars by scopes whenever variable binding may be involved.

4 Binding Lisps

Given the mission to develop type theories in broad generality, we shall need to be open minded about what constucts they might offer. Let us adopt a simple but generic way of representing them: tasteful notations for specific theories can be layered on top, but for purposes of metatheory, the less artful we are, the better. There are many tolerable choices we might make, but my childhood guides me to jump in Lisp puddles, and my training as a graduate student warns me to be cautious about variable binding.

Definition 7 (*binding Lisp*)
A **binding Lisp** is some grammar, $\langle \gamma\text{-}lisp \rangle$ with at least the following constructs:

$$\begin{array}{rcll} \langle \gamma\text{-}lisp \rangle & ::= & \langle atom \rangle & \text{— a sequence of alphanumeric characters} \\ & | & (\langle \gamma\text{-}lisp \rangle \dot{\langle} \gamma\text{-}lisp \rangle) & \text{— a cons-cell} \\ & | & x \texttt{->} \langle \gamma, x\text{-}lisp \rangle & \text{— an abstraction} \end{array}$$

As ever, we allow syntactic sugar to avoid an excess of dotted pairs.

Definition 8 (*binding Lisp, relaxed notation*)

$$\begin{array}{rcll} \langle \gamma\text{-}lisp \rangle & ::= & \langle atom \rangle & \text{— a sequence of alphanumeric characters} \\ & | & (\langle \gamma\text{-}lisp \rangle \dot{\langle} \gamma\text{-}lisp \rangle) & \\ & | & (\langle \gamma\text{-}constr \rangle) & \text{— a construction in parentheses} \\ & | & x \texttt{->} \langle \gamma, x\text{-}lisp \rangle & \text{— an abstraction} \\ \langle \gamma\text{-}constr \rangle & ::= & & \text{— nil, the empty atom} \\ & | & . \langle \gamma\text{-}lisp \rangle & \text{— a non-nil tail} \\ & | & \langle \gamma\text{-}lisp \rangle \, \langle \gamma\text{-}constr \rangle & \text{— cons} \end{array}$$

Note that

1. some atoms may look like variables;
2. `()` is a way to write empty atom;
3. $.\cdot$ and $(\cdot)$ cancel, so that $(.l)$ is the same $\langle \gamma\text{-}lisp \rangle$ object as l. and $.(c)$ is the same $\langle \gamma\text{-}constr \rangle$ object as c;
4. this basic syntax offers no way to *use* a variable.

Let us remedy the latter posthaste!

Definition 9 (*terms*)
The syntax of **terms** is a binding Lisp augmented with the following additional construct

$$\begin{array}{rcll} \langle \gamma\text{-}term \rangle & ::= & \cdots & \text{— all the binding Lisp constructs} \\ & | & [\langle \gamma\text{-}elim \rangle] & \text{— an elimination in brackets} \\ \langle \gamma\text{-}elim \rangle & ::= & \langle \gamma\text{-}var \rangle & \text{— variable usage} \\ & | & (\langle \gamma\text{-}term \rangle : \langle \gamma\text{-}term \rangle) & \text{— a radical} \\ & | & \langle \gamma\text{-}elim \rangle \, \langle \gamma\text{-}term \rangle & \text{— an action} \end{array}$$

where a $\langle \gamma\text{-}var \rangle$ is one of the variables in γ, readily relieved of its name by determining its position in γ.

The binding Lisp constructs give us the means to express the canonical forms of a theory; the eliminations allow us to express computations. The latter amount to a *head* with a possibly empty sequence, or *spine*, of actions attached. A head may be a variable, as bound by an abstraction, or it may be a *radical* — a term (possibly canonical) activated for computation by annotation with its type.

We shall define our type theories by saying which terms are types and which terms are *checked* by those types. By contrast, and bidirectionally, we shall always be able to *synthesize* types for meaningful eliminations.

To aid comprehension, I shall write atoms in `teletype` font and variables in *italics*. However, you can always spot a variable binding by the `->` to its right and a variable usage by the `[` to its left.

Example 10 (*polymorphic identity function*)
We may write `Type` for the type of types and (`Pi` S x`->` T) for a dependent function type. The type of the polymorphic identity function may thus be written

$$\texttt{(Pi Type } X \texttt{ -> (Pi [}X\texttt{] } x \texttt{ -> [}X\texttt{]))}$$

with the polymorphic identity function being just

$$X \texttt{ -> } x \texttt{ -> [}x\texttt{]}$$

The role of atoms is to be distinct from one another. The role of variables is to connect usage with abstraction: naming them variables is an informal concession to readability.

All binding Lisps admit thinning.

Definition 11 (*thinning action*)
If s is an object in some $\langle\gamma\text{-}lisp\rangle$ and $\theta : \gamma \sqsubseteq \delta$ is a thinning, then $s\theta$ is the object in the corresponding $\langle\delta\text{-}lisp\rangle$ given by

$$\begin{array}{lcllcl}
\texttt{a}\theta & = & \texttt{a} & \theta & = & \\
(c)\,\theta & = & (c\theta) & (.t)\theta & = & .(t\theta) \\
(x\texttt{->}t)\theta & = & x\texttt{->}(t\theta 1) & (t\ c)\theta & = & (t\theta)\,(c\theta)
\end{array}$$

This action has a partial inverse, written $\theta?t$ for some $\langle\delta\text{-}lisp\rangle$ object t, giving the $\langle\gamma\text{-}lisp\rangle$ object s such that $s\theta = t$ if it exists. It remains to specify the action of thinnings on the extra constructs of specific $\langle\gamma\text{-}lisp\rangle$s. For $\langle\gamma\text{-}term\rangle$, we have

$$\begin{array}{lcl}
\texttt{[}e\texttt{]}\,\theta & = & \texttt{[}e\theta\texttt{]} \\
\\
x\theta & = & x \\
(t\texttt{:}T)\,\theta & = & (t\theta\texttt{:}T\theta) \\
(e\ t)\theta & = & (e\theta)\ (t\theta)
\end{array}$$

where, in a de Bruijn representation $x\theta = x$ computes the representation for x in the target scope from the representative of x in the source scope.

5 Meta-syntax: patterns, thinnings and expressions

For my purposes, it is not enough to be formal about the type theory which is the object of study. I intend to establish general results about classes of type theories. I must therefore be formal about the *meta*-language in which I am formal about type theories. When we write typing rules or contraction schemes, we do not write terms of our type theory, but rather *formulae* containing *meta*variables, which describe terms in our type theory with particular structural properties. What are those formulae?

It is conventional to generate such formulae by the free extension of the object theory with metavariables, intending all the terms which are given by instantiating the metavariables. I fear I shall be more painstaking, distinguishing two classes of formula, the *patterns* which contain the binding sites of metavariables, from the *expressions* which contain their

use sites. The benefit of this distinction will be to allow much tighter control of information flow through the rules of the theory.

Definition 12 (*pattern*)
The syntax of *patterns* is a binding Lisp augmented with the following additional construct

$$\begin{array}{rcll} \langle \gamma\text{-}pat \rangle & ::= & \cdots & \text{— all the binding Lisp constructs} \\ & | & \{\langle mvar \rangle\ \langle \gamma\text{-}thinning \rangle\} & \text{— a metavariable binding in braces} \end{array}$$

and thinnings acting by

$$\{m\ \phi\}\theta = \{m\ \phi; \theta\}$$

A metavariable binding gives a name to a metavariable, for the sake of human readability: these names should be mutually distinct. In practice, metavariable bindings may effectively be distinguished by *position* and need not have names at all. As a metavariable binding may occur inside zero or more object variable abstractions, we write a *thinning* to indicate upon which of those object variables a term instantiating the metavariable may depend. The surface syntax of thinnings is conveniently redundant, allowing us to specify permitted dependencies either by inclusion or by exclusion.

Note that patterns allow us to analyse only

1. the canonical form structure induced by a binding Lisp;
2. dependency on *bound* variables.

That is, the pattern language is designed carefully to ask only questions whose answers are invariant under substitution of free variables.

Crucially, from every pattern, we may compute its *problem* — the sequence of name/source scope pairs m/δ given by each $\{m\ \theta\}$ contained therein.

Example 13 (*function types*)
The pattern that describes a dependent function type is

$$(\texttt{Pi}\ \{S\}\ x\texttt{->}\ \{T\}) \qquad \text{or, equivalently, the explicit} \qquad (\texttt{Pi}\ \{S\}\ x\texttt{->}\ \{T\ x\})$$

whose problem is

$$S/; T/x$$

The pattern that captures the non-dependent special case is

$$(\texttt{Pi}\ \{S\}\ x\texttt{->}\ \{T - x\}) \qquad \text{or, equivalently, the implicit} \qquad (\texttt{Pi}\ \{S\}\ x\texttt{->}\ \{T-\})$$

whose problem is

$$S/; T/$$

One way to solve a problem Ω is to match a $\langle \gamma\text{-}term \rangle$ to a $\langle \varepsilon\text{-}pat \rangle$, mapping each m/θ to a $\langle \gamma, \theta\text{-}term \rangle$, yielding a sequence of definitions $m(\theta) = t$. I write ε for the empty solution.

Definition 14 (*pattern matching*)

If p is a $\langle \delta\text{-}pattern \rangle$ with problem Ω and t a $\langle \gamma, \delta\text{-}term \rangle$, then $p\,?\,t$ is the $\Omega \Rightarrow \gamma$ solution partially defined as follows:

$$\begin{array}{lll} \mathtt{a} \quad ?\mathtt{a} & = & \varepsilon \\ (p_a \,.\, p_d)\,?\,(t_a \,.\, t_d) & = & p_a\,?\,t_a;\ p_d\,?\,t_d \\ (x\mathtt{->}\,p)\ ?(x\mathtt{->}\,t) & = & p\,?\,t \\ \{m\ \theta\}\quad ?t & = & m(\theta) = 1, \theta\,?\,t \end{array}$$

Note that in the abstraction case, it is not a requirement that the names match, rather, by α-equivalence, the names do match.

Now, let us turn to the expressions.

Definition 15 (*expression*)
The syntax of expressions is a binding Lisp augmented with the following additional constructs:

$$\begin{array}{rcll} \langle \gamma\text{-}expr \rangle & ::= & \cdots & \text{— all the binding Lisp constructs} \\ & | & \mathtt{[}\langle \gamma\text{-}image \rangle\mathtt{]} & \text{— as 'elim' is to 'term'} \\ & | & \mathtt{\{}\langle mvar \rangle(/\langle \gamma\text{-}image \rangle)^*\mathtt{\}} & \text{— a metavariable binding in braces} \\ \langle \gamma\text{-}image \rangle & ::= & \langle \gamma\text{-}var \rangle & \text{— variable usage} \\ & | & \mathtt{(}\langle \gamma\text{-}expr \rangle\mathtt{:}\langle \gamma\text{-}expr \rangle\mathtt{)} & \text{— a radical} \\ & | & \langle \gamma\text{-}image \rangle\ \langle \gamma\text{-}expr \rangle & \text{— an action} \end{array}$$

with thinning acting structurally.

Each metavariable must be instantiated with an image for each bound variable on which they depend, as determined by the pattern which binds the metavariable. Expressions are the language that rules use for synthesis, making use of information gained by pattern matching. E.g., once we have matched a function type with pattern

$$(\mathtt{Pi}\ \mathtt{\{}S\mathtt{\}}\ x\mathtt{->}\ \mathtt{\{}T\mathtt{\}})$$

and an argument with pattern

$$\mathtt{\{}s\mathtt{\}}$$

we may deliver the type of the application with the expression

$$\mathtt{\{}T/(\mathtt{\{}s\mathtt{\}}\mathtt{:}\mathtt{\{}S\mathtt{\}})\mathtt{\}}$$

6 Judgements and Rules

A judgement form is a crude variety of 'session type', characterising the data involved as *inputs*, *subjects* or *outputs*. A subject is something to be validated by the judgement form: not all judgements require a subject, but for every sort of thing, there must be a judgement form for which it is the subject. An input must have a *client promise*, being a judgement with that input as its subject, explaining what sort of validation the client is expected to perform in advance of seeking to derive a judgement with that input. An output must have a corresponding *server promise*, being a judgment with that output as its subject, explaining what sort of validation must be ensured by whoever claims to have derived the judgement

There are four judgement forms which are considered primitive. I list them in their ascii notation, with their associated promises given in braces:

1. type formation
$$\texttt{type}\ T_{\text{subj}}$$
takes a putative type as its subject;
2. type checking
$$\{\texttt{type}\ T_{\text{in}}\} \qquad T_{\text{in}}\ \texttt{:>}\ t_{\text{subj}}$$
takes a putative term as its subject and checks it in a given type;
3. type synthesis
$$e_{\text{subj}}\ \texttt{<:}\ T_{\text{out}} \qquad \{\texttt{type}\ T_{\text{out}}\}$$
takes an elimination as its subject and tries to find a type for it;
4. subtyping
$$\{\texttt{type}\ S_{\text{in}} \quad \texttt{type}\ T_{\text{in}}\} \qquad S_{\text{in}}\ \texttt{<=}\ T_{\text{in}}$$
takes two given types and checks whether one is subsumed by the other.

7 Computation

Making no presumption of strong normalization, I adopt a *small-step* notion of reduction, generated by the contextual closure of contraction schemes. Whatever other constructs and contractions we may legitimize, we shall certainly have the following:

Definition 16 (υ*-contraction*)

$$[(t : T)] \rightsquigarrow_{\upsilon} t$$

A spineless radical ceases to be a radical at all. It loses its type annotation and delivers its now deactivated result into the term which is its host.

References

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

Martin-Löf, Per. (1971). A theory of types. *Unpublished manuscript*.

McCarthy, John. (1960). Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the acm*, **3**(4), 184–195.

Pierce, Benjamin C., & Turner, David N. (2000). Local type inference. *ACM trans. program. lang. syst.*, **22**(1), 1–44.