Automating Fraenkel-Mostowski Syntax*

Murdoch J. Gabbay[1]

Computer Laboratory, Cambridge University, UK
mjg1003@cl.cam.ac.uk
http://cl.cam.ac.uk/~mjg1003

Abstract. Work with Pitts and others has led to FM (Fraenkel-Mostowski) theory, a fresh understanding of modelling syntax in the presence of variable binding. We discuss the design and other issues encountered implementing these techniques in the mechanised theorem-prover Isabelle.

1 Introduction

It is easy to declare a naïve datatype of terms of some language, for example the untyped λ-calculus,

$$\Lambda = \mu X.\text{Var } \mathbf{of}\ \mathbf{Nat} + \text{App } \mathbf{of}\ X \times X + \text{Lam } \mathbf{of}\ \mathbf{Nat} \times X \tag{1}$$

where **Nat** is the natural numbers. Problems famously arise defining program transformations in the presence of variable binding. For example a substitution function $[t/a]s$ on Λ above should avoid "accidental variable capture" in $[\text{Var}(1)/\text{Var}(0)]s$ for $s = \text{Lam}(1, \text{Var}(0))$. Thus we rename 1 in s to some $i \neq 0, 1$, but then $\text{Var}(i)$ is no longer syntactically a subterm of s and we have made an arbitrary choice about the value of i. The former point causes difficulty with structural induction, the latter because we may have to formally prove irrelevance of the choice made.[1]

All this we could do without, especially in the unforgiving structure of a computer proof assistant such as Isabelle, HOL98, or COQ, or even programming in some language with datatypes. There is much research in this area, for example explicit substitutions ([2]), de Bruijn indices ([3]), and HOAS ([11], [4], [9]).

FM theories are another approach with a pleasingly elementary mathematical foundation. See [7] (my thesis), [5] and [6] (set theory), [8] (higher-order logic), [13] (programming languages), [12] (first-order logic). The label "Fraenkel-Mostowski" honours the creators of set theories designed to prove the independence of the axiom of choice, see [15]: a very special Fraenkel-Mostowski set theory was the first FM theory in the sense of this paper to be created.

In this paper we discuss principles of formally implementing a theory of FM syntax, based on experience doing so in Isabelle [14].

The first design decision of the implementation is the choice of system, Isabelle. We chose Isabelle for its paradigm of constructing arbitrary useable theories (Isabelle/Pure/FOL, Isabelle/Pure/HOL, Isabelle/Pure/CCL, ..., see [14]) in a fixed weak meta-language Isabelle/Pure. This meta-language is a very weak higher-order logic (HOL) containing little more than modus ponens, but to which we may add new types, constants of those types, and axioms on those constants. Thus we may axiomatise a theory in Isabelle/Pure and then work inside that theory. This is good for prototyping a new foundational system such as FM.

2 FM

'FM' may differ depending on whether we do computation or logic. For example compare the typed λ-calculus (a theory of computable functions) to higher-order logic (a theory of all functions). This paper is about logic, FM in computation (programming languages, unification) is under development, see [13, 1].

* The author gratefully acknowledges the funding of UK EPSRC grant GR/R07615 and thanks Andrew Pitts for his suggestions for improvements.

[1] Cf. the work of McKinna and Pollack in the LEGO system, e.g. [10]. FM is quite different but sometimes echoes this work.

'FM' is a set of techniques for α-equivalence with inductive definitions and not a particular theory. We shall now present FM in the style of a higher-order logic. This is not an axiomatic presentation (see [8]) but a 'sketch of salient features, in the style of higher-order logic'. First, three preliminary remarks:

1 (Types). We shall write type annotations in two styles: $x:\alpha$ and x^α both mean "x of type α". ◇

2 (HOL sets). Higher-order logic has a notion of set, where 'α-sets' is predicates $\alpha \to \texttt{Bool}$ also written $\mathcal{P}(\alpha)$. We borrow set notation, for example writing $x \in X$ for '$(X\,x)$', $X \subseteq Y$ for '$\forall x.\ (X\,x) \to (Y\,x)$', and $\emptyset$ for $\lambda x.\bot$ and α for $\lambda x^\alpha.\top$. ◇

3 (Meaning of infinite). In FM theories not all types can be well-ordered (bijected with an ordinal, see [8, Lemma 4.10(5)]). Therefore, a reading of 'X is infinite' as $X \cong \mathbb{N}$ is suspect. In FM we use '$X \notin \mathcal{P}_{fin}(X)$' where $\mathcal{P}_{fin}(X)$ is the inductively defined type of finite subsets of X. ◇

An FM theory has:

4 (Atoms). An infinite type of ***atoms*** $a, b, c, \ldots : \mathbb{A}$ to model variable names. For example in an inductively defined type of expressions for types,

$$\Sigma ::= \mathbf{TypeVar\ of}\ \mathbb{A} + \mathbf{Product\ of}\ \Sigma \times \Sigma + \mathbf{DisjSum\ of}\ \Sigma \times \Sigma, \tag{2}$$

type variables are represented as $\mathbf{TypeVar}(a)$ for $a:\mathbb{A}$. ◇

5 (Transposition). There is a (polymorphically indexed class of) constant(s)

$$\mathbf{Tran} : \mathbb{A} \to \mathbb{A} \to \sigma \to \sigma, \tag{3}$$

read "***transposition***". Write $(\mathbf{Tran}\,a\,b\,x^\sigma)$ as $(a\ b).x$. The intuitive meaning of $(a\ b).x$ is as transposing a and b in x. For example if $x = <a, b>$ then $(a\ b).x$ should equal $<b, a>$. This is made formal by the following equational axioms which **Tran** must satisfy, and equivariance below:

$$(a\ a).x = x \tag{4}$$
$$(a\ b).(a\ b).x = x \tag{5}$$
$$(a\ b).(c\ d).x = (c\ d).\ ((c\ d).a\ (c\ d).b)\ .x \tag{6}$$
$$(a\ b).n^{\mathbb{A}} = if(n = a, b, if(n = b, a, n)) \tag{7}$$

where $if(test, t_1, t_2)$ is Isabelle-like notation meaning "if $test$ then t_1 else t_2". ◇

6 (Equivariance). For a term f with free variables $x_1, \ldots, x_n$,

$$(a\ b).f(x_1, \ldots, x_n) = f((a\ b).x_1, \ldots, (a\ b).x_n). \tag{8}$$

In the case that f has no free variables we have the special case that $(a\ b).f = f$.

We say the language is ***equivariant***. An ***equivariant element*** x is one such that for all a, b, $(a\ b).x = x$. From (8) for $n = 0$ it follows that closed terms denote equivariant elements. ◇

Definition 7 (Smallness, И). *Write $\mathcal{P}_{fin}(\mathbb{A})$ for the* HOL *set of finite subsets of $\mathbb{A}$. Say a set $X \subseteq \mathbb{A}$ is cofinite when its complement $\mathbb{A} \setminus X$ is finite. Write $\mathcal{P}_{cofin}(\mathbb{A})$ for the* HOL *set of cofinite subsets of $\mathbb{A}$. For $P : \mathbb{A} \to \texttt{Bool}$ write 'ИP' or 'Иa. $P(a)$' for $P \in \mathcal{P}_{cofin}(\mathbb{A})$.*

$\mathbb{A}$ is infinite from remark 4 so we can read Иa. $P(a)$ as "for all but finitely many $a:\mathbb{A}$, $P(a)$", or more loosely as "for *most* $a:\mathbb{A}$, $P(a)$". We may call finite $P : \mathbb{A} \to \texttt{Bool}$ ***small*** and their complements, cofinite sets, ***large***. Thus P is large precisely when Иa. $P(a)$, and small precisely when Иa. $\neg P(a)$.

Definition 8. *Define $a\#x \stackrel{\text{def}}{=} (И b.\ (b\ a).x = x)$ and read this as "a is not 'in' x" or "a is apart from x". The intuition is that, since transposition transposes b for a in x and since b is fresh, if $(b\ a).x = x$ then certainly a is not in x.*

We have an axiom stating that 'most' atoms are not 'in' $x:\sigma$:

$$И a.\ a\#x. \tag{Small}$$

Expanding definition 8 this becomes $И a.\ И b.\ (a\ b).x = x$. Write

$$\mathbf{Supp}(x) \stackrel{\text{def}}{=} \{a:\mathbb{A} \mid \neg a\#x\}$$

Then (Small) is equivalent to

$$\mathbf{Supp}(x) \in \mathcal{P}_{fin}(x) \tag{9}$$

and we can also read (Small) as "x ***has finite support***".

9 (Some observations). 'In' does *not* correspond to (HOL-)set membership. For example,

$$n \notin L = \mathbb{A} \setminus \{n\} \quad \textbf{but} \quad n \in \mathbf{Supp}(L).$$

We might think of $\mathbf{Supp}(x)$ as an object-level notion of those atoms occurring in some *meta-level* term which x denotes.

Datatypes of syntax T certainly satisfy (9). Terms $t:T$ are finite[2] so mention only finitely many atoms, and cofinitely many $a:\mathbb{A}$ satisfy $a\#t$. ◇

10 (И excellent properties). Higher types such as $\mathbb{A} \to \texttt{Bool}$ also satisfy (9). Observe that (using some sets notation)

$$P:\mathbb{A} \to \texttt{Bool} = \{x \mid P(x)\}.$$

It follows from (8) that

$$(a\ b).P = \{(a\ b).x \mid P(x)\}.$$

We can verify by calculation that $(a\ b).P = P$ if and only if $a,b \in P$ or $a,b \notin P$. When we combine this with (9) it follows that either 'most' atoms are in P or most are not in P:

$$\mathcal{P}(\mathbb{A}) \cong \mathcal{P}_{fin}(\mathbb{A}) + \mathcal{P}_{cofin}(\mathbb{A}). \tag{10}$$

We can rewrite this as $\neg И a.\ P(a) \Leftrightarrow И a.\ \neg P(a)$. Now the full set $\lambda x.\top = \mathbb{A} \subseteq \mathbb{A}$ is clearly cofinite so $И a.\ \top = \top$. Combining this with other properties of cofinite and finite sets we obtain the algebraic commutativity properties:

$$\begin{aligned}
И a.\ P(a) \wedge И a.\ Q(a) &\iff И a.\ P(a) \wedge Q(a) && (11)\\
И a.\ P(a) \vee И a.\ Q(a) &\iff И a.\ P(a) \vee Q(a) && (12)\\
И a.\ \neg P(a) &\iff \neg И a.\ P(a) && (13)\\
&\qquad\quad И a.\ \top && (14)\\
&\qquad\quad \neg И a.\ \bot && (15)\\
(И a.\ P(a)) \wedge Q &\iff И a.\ (P(a) \wedge Q), && (16)
\end{aligned}$$

that is, И distributes over $\wedge$, $\vee$, $\to$, $\neg$, $\top$ and $\bot$. These strong properties make И convenient to work with in a mechanised context. They also place И in an interestingly 'in between' $\forall$ and $\exists$, the equations being informally:

$$\neg\forall\neg = \exists \qquad \neg И \neg = И \qquad \neg\exists\neg = \forall.$$

◇

[2] Extending FM to infinitary syntax is possible and interesting.

3 α-equivalence on a simple datatype

11. Tran is how an FM theory renames object-level variables. It interacts with object-level syntax better than atom-substitution $[b/a]$. For example $[b/a]$ applied to $t = \mathbf{Lam}(a).\mathbf{Lam}(b).a(b)$ raises all the usual problems with capture avoidance, whereas $(b\ a).t = \mathbf{Lam}(a).\mathbf{Lam}(b).b(a)$ is α-equivalent to t itself. Similarly, $(b\ a).(\mathbb{A} \setminus \{a\}) = \mathbb{A} \setminus \{b\}$ wheras $[b/a](\mathbb{A} \setminus \{a\}) = \mathbb{A} \setminus \{a\}$. $\diamond$

We can define an α-equivalence relation $=_\alpha$ by cases on inductive types. For example:

Definition 12.

$$\begin{array}{rcll} L & \stackrel{\text{def}}{=} & \textit{TyVar of } \mathbb{A} + \textit{TyProd of } L \times L + \textit{TyAbs of } \mathbb{A} \times L & \\ =_\alpha & \stackrel{\text{def}}{=} & \textit{TyVar}(a) =_\alpha \textit{TyVar}(b) & \leftarrow a = b \\ & & \textit{TyProd}(t_1, t_2) =_\alpha \textit{TyProd}(t_1', t_2') & \leftarrow t_1 =_\alpha t_1' \wedge t_2 =_\alpha t_2' \\ & & \textit{TyAbs}(a, t) =_\alpha \textit{TyAbs}(a', t') & \leftarrow \mathcal{И}b.\ (b\ a).t =_\alpha (b\ a').t' \end{array}$$

Here L is intended to be a type of expressions for types. The definition of L above might be written in more familiar style as

$$l ::= \mathbf{TyVar}(a) \mid \mathbf{TyProd}(l, l) \mid \mathbf{TyAbs}(a, l) \qquad a : \mathbb{A},$$

and sugared to (writing σ for l a type and α for $a : \mathbb{A}$ a type variable)

$$\sigma ::= \alpha \mid \sigma \times \sigma \mid \bigwedge \alpha.\, \sigma.$$

We shall use L, $=_\alpha$, and $=_\alpha{}'$ defined below in (18), as a running object of study in the rest of this paper. In the rest of this section and elsewhere the proofs given are semi-formal accounts of the formal proofs as they might be conducted in Isabelle.

This machinery allows us to quite easily prove some nice properties for $=_\alpha$, for example transitivity:

Lemma 13. $=_\alpha$ is transitive.

Proof. By induction on syntax using hypothesis

$$\phi(t_1) \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \forall t_2, t_3.\ (t_1 =_\alpha t_2 \wedge t_2 =_\alpha t_3) \rightarrow t_1 =_\alpha t_3.$$

The significant case is of $t_1 = \mathbf{TyAbs}(a_1, t_1')$. So suppose $\phi(t_1')$, $t_1 =_\alpha t_2$, and $t_2 =_\alpha t_3$. Then $t_2 = \mathbf{TyAbs}(a_2, t_2')$ and $t_3 = \mathbf{TyAbs}(a_3, t_3')$, and

$$(\mathcal{И}b.\ (b\ a_1).t_1' =_\alpha (b\ a_2).t_2') \wedge (\mathcal{И}b.\ (b\ a_2).t_2' =_\alpha (b\ a_3).t_3').$$

We now equationally apply (11) to deduce

$$\mathcal{И}b.\ (b\ a_1).t_1' =_\alpha (b\ a_2).t_2' =_\alpha (b\ a_3).t_3'. \tag{17}$$

Now we assumed $\phi(t_1')$, not $\phi((b\ a_1).t_1')$. But we can apply equivariance (8) to $\phi(x)$ to deduce $\phi(t_1') \iff \phi((b\ a_1).t_1')$, which allows us to complete the proof. $\square$

We can also define a more traditional α-equivalence $=_\alpha{}'$:

$$\begin{array}{ll} \mathbf{TyVar}(a) =_\alpha{}' \mathbf{TyVar}(b) & \leftarrow a = b \\ \mathbf{TyProd}(t_1, t_2) =_\alpha{}' \mathbf{TyProd}(t_1', t_2') & \leftarrow t_1 =_\alpha{}' t_1' \wedge t_2 =_\alpha{}' t_2' \\ \mathbf{TyAbs}(a, t) =_\alpha{}' \mathbf{TyAbs}(a', t') & \leftarrow \exists b.\ [b/a].t =_\alpha{}' [b/a']t' \wedge \\ & \qquad b \notin n(t) \cup n(t') \cup \{a, a'\} \end{array} \tag{18}$$

in terms of an inductively defined names-of function $n(t)$

$$\begin{array}{ll} n(\mathbf{TyVar}(a)) & = \{a\} \\ n(\mathbf{TyProd}(t_1, t_2)) & = n(t_1) \cup n(t_2) \\ n(\mathbf{TyAbs}(a, t)) & = \{a\} \cup n(t) \end{array} \tag{19}$$

and an inductively defined atom-for-atom substitution function

$$\begin{array}{ll} [b/a]\mathbf{TyVar}(a) & = \mathbf{TyVar}(b) \\ [b/a]\mathbf{TyVar}(n) & = \mathbf{TyVar}(n) \quad n \neq a \\ [b/a]\mathbf{TyProd}(t_1, t_2) & = \mathbf{TyProd}([b/a]t_1, [b/a]t_2) \\ [b/a]\mathbf{TyAbs}(n, t) & = \mathbf{TyAbs}([b/a]n, [b/a]t). \end{array} \tag{20}$$

$n(t)$ and $[b/a]t$ are simple and make no allowance for free variables or capture-avoidance, but they suffice for our needs.

Suppose we want to prove $t_1 =_\alpha t_2 \leftrightarrow t_1 =_{\alpha'} t_2$. A pleasing and clean method would be to prove

$$И b.\ [b/a]t = (b\ a).t \tag{21}$$

$$\exists b.\ [b/a].t =_{\alpha'} [b/a']t' \ \wedge\ b \notin n(t) \cup n(t') \cup \{a, a'\} \iff И b.\ [b/a]t =_{\alpha'} [b/a']t'. \tag{22}$$

Proof (of (21)). We can use structural induction for a fixed with hypothesis ϕ

$$\phi(t) \quad \stackrel{\text{def}}{\iff} \quad И b.\ [b/a]t = (b\ a).t.$$

Suppose $t = \mathbf{TyProd}(t_1, t_2)$. By definition from (20), $[b/a]\mathbf{TyProd}(t_1, t_2) = \mathbf{TyProd}([b/a]t_1, [b/a]t_2)$, and by equivariance (8), $(b\ a).\mathbf{TyProd}(t_1, t_2) = \mathbf{TyProd}((b\ a).t_1, (b\ a).t_2)$. By hypothesis we know

$$(И b.\ [b/a]t_1 = (b\ a).t_1) \wedge (И b.\ [b/a]t_2 = (b\ a).t_2).$$

By (11) and applying the equalities under **TyProd** we obtain the result.

The cases of **TyVar** and **TyAbs** are no different. Each time, equivariance of $(b\ a)$ as illustrated in (8) allows us to push transposition down through the structure of a term and replicate the inductive behaviour of $[b/a]$. This is a general pattern. □

Note from this proof how transposition with equivariance has provided a 'general axiomatic theory of (purely inductive) renaming'.

Proof (of (22)). The proof of (22) is rather more involved. It is best to work from the following lemmas:

$$n(t) \in \mathcal{P}_{fin}(\mathbb{A}) \tag{23}$$

$$X \in \mathcal{P}_{fin}(\mathbb{A}) \implies (b \notin X \leftrightarrow b\#X) \tag{24}$$

$$b \notin n(t) \iff b\#t \tag{25}$$

$$b\#x \wedge b\#f \implies b\#f(x) \tag{26}$$

$$b\#(f(x)) \wedge b\#f \wedge f\ \textbf{injective} \implies b\#x \tag{27}$$

$$b\#c \quad c\ \textbf{a closed term} \tag{28}$$

$$b\#P^{\mathbb{A}\to\texttt{Bool}} \wedge P(b) \implies И b.\ P(b) \tag{29}$$

$$И b.\ P(b) \implies \exists b.\ b\#P \wedge P(b) \tag{30}$$

$$И b.\ P(b) \implies \forall b.\ b\#P \implies P(b) \tag{31}$$

$$\exists b.\ b\#x \tag{32}$$

The proof now proceeds as follows. We must prove

$$\exists b.\ [b/a]t =_{\alpha'} [b/a']t' \wedge b \notin n(t) \cup n(t') \cup \{a, a'\} \iff И b.\ [b/a]t =_{\alpha'} [b/a']t'.$$

Write $P \stackrel{\text{def}}{=} \lambda a, a', t, t'.\lambda b.[b/a]t =_{\alpha'} [b/a']t'$ and use (25) (proved from (23) and (24)) and to rewrite this to

$$\exists b.\ b\#t, t', a, a' \wedge P(t, t', a, a', b) \iff И b.\ P(t, t', a, a', b),$$

where $b\#x_1,\ldots,x_n$ denotes the conjunction $\bigwedge_i b\#x_i$.
Left-right implication. We must prove

$$\exists b.\ b\#t,t',a,a' \wedge P(t,t',a,a',b) \implies И b.\ P(t,t',a,a',b).$$

We eliminate the existential quantifier and obtain

$$b\#t,t',a,a' \wedge P(t,t',a,a',b) \implies И b.\ P(t,t',a,a',b). \tag{33}$$

We resolve against (29) to obtain

$$b\#t,t',a,a' \wedge P(t,t',a,a')(b) \implies b\#P(t,t',a,a') \wedge P(t,t',a,a')(b),$$

which simplifies to $b\#t,t',a,a' \implies b\#P(t,t',a,a')$. We repeatedly resolve against (26) to reduce to $b\#P$, and finish this off with (28).
Right-to-left implication. We must prove

$$И b.\ P(t,t',a,a',b) \implies \exists b.\ b\#t,t',a,a' \wedge P(t,t',a,a',b).$$

Now here we have a problem. Clearly we would like to eliminate И using (30) to obtain

$$b\#P(t,t',a,a') \wedge P(t,t',a,a',b) \implies \exists b.\ b\#t,t',a,a' \wedge P(t,t',a,a',b),$$

identify the b in the conclusion with the b in the hypotheses, and simplify. But we obtain

$$b\#P(t,t',a,a') \implies b\#t,t',a,a'.$$

This implication does not follow for general P, nor even for our particular P: if P were injective we could apply (27) repeatedly, but it is a predicate mapping into `Bool` and is not injective.

However we can use (32) to introduce into the context some b fresh for *any* x, so instantiate x to the 3-tuple

$$<n(t),n(t'),\{a,a'\}>. \tag{34}$$

Now we can apply (27) repeatedly to obtain

$$b\#t,t',a,a' \wedge И b.\ P(t,t',a,a',b) \implies \exists b.\ b\#t,t',a,a' \wedge P(t,t',a,a',b).$$

(Here there is also a hypothesis $b\#\lambda x_1,x_2,x_3.<x_1,x_2,x_3>$ but this gives us no information since we get it for free from (28), so we drop it.) This simplifies to

$$b\#t,t',a,a' \wedge И b.\ P(t,t',a,a',b) \implies P(t,t',a,a',b).$$

But now we have another problem. If we eliminate И using (30) we obtain for a variable symbol b',

$$b\#t,t',a,a' \wedge b'\#P(t,t',a,a') \wedge P(t,t',a,a',b') \implies P(t,t',a,a',b).$$

We need a different elimination rule for И which does not introduce a new variable into the context, and this is provided by (31), with which we can finish off the proof. □

4 Morals from the proofs

In the previous section we have seen the beginnings of the automated theory of $(a\ b)$, $\#$, introducing a fresh name, and И. We now bring it out explicitly.

14 (Theory of transposition). Given a conclusion of the form $s=(a\ b).t$, use (8) to simplify the RHS by drawing transposition down to the variables on the right hand side. Similarly for other binary predicates such as $\leftrightarrow$ or also $\#$. So for example

$$s=(a\ b).<x,y> \quad \textbf{simplifies to} \quad s=<(a\ b).x,(a\ b).y>.$$

This algorithm can fail, for example on the goal $(a\ b).<x,y>=(a\ b).<x,y>$. Call it `push`, because it 'pushes' transposition into the structure of the term on the right of an equality. In an implementation `push` would denote a tactic. We shall continue to give such names to algorithms which would denote tactics. ◇

15 (Theory of #). Given a goal of the form $a\#t$ repeatedly apply (26) and (28) to simplify it to component parts. So for example

$$a\#<x,y>$$

reduces to $a\#x \wedge a\#y \wedge a\#\lambda x,y.<x,y>$, and then to $a\#x \wedge a\#y$. This algorithm can also fail, for example in $a\#\pi_1<\top,a>$ we should perform β-reduction first, otherwise we finish up with $a\#a$, which is untrue. Call the algorithm `split#`. ◇

Inductive proof on inductive types can, with proper handling and properly coordinated automated procedures, be made to produce very uniform proof-obligations which are amenable to this kind of treatment, with only slightly more sophisticated algorithms.

16 (Introducing a fresh name). (32) allows us to introduce a new variable b into the context, fresh for x for any x: given the proof-state

$$\forall x_1,\ldots,x_n.\ Conds(x_1,\ldots,x_n) \implies Concl(x_1,\ldots,x_n)$$

we can reduce to

$$\forall x_1,\ldots,x_n,b.\ Conds(x_1,\ldots,x_n) \wedge b\#t(x_1,\ldots,x_n,b) \implies Concl(x_1,\ldots,x_n)$$

for any t.

We can now take t to be the n-tuple $<\mathbf{Supp}(x_1),\ldots,\mathbf{Supp}(x_n)>$. Repeated applications of (27) reduce $b\#t$ to $\bigwedge_i b\#\mathbf{Supp}(x_i)$. It is a lemma that $b\#\mathbf{Supp}(u) \iff b\#u$, so we obtain

$$\forall x_1,\ldots,x_n,b.\ Conds(x_1,\ldots,x_n) \wedge \bigwedge_i b\#x_i \implies Concl(x_1,\ldots,x_n).$$

In other words, "we can always invent a fresh b". We applied this technique ad-hoc in (34). Call the algorithm `newname`. ◇

17 (Theory of И). The treatment of И is more complex. There are two broad styles of reasoning on И, equational reasoning using for example properties such as (11) and (16), and directed reasoning using intro- and elim- rules such as (29), (30), and (31). Both are useful. For example equations in (17), and intro- and elim- rules in the proof of (22).

A further complication of the treatment of intro- and elim- rules is that И seems to have two pairs of them. In full, they are

$$\exists b.\ (b\#P^{\mathbb{A}\to\mathtt{Bool}} \wedge P(b)) \implies И b.\ P(b) \tag{35}$$

$$И b.\ P(b) \implies (\exists b.\ b\#P \wedge P(b)) \tag{36}$$

$$\forall b.\ (b\#P^{\mathbb{A}\to\mathtt{Bool}} \to P(b)) \implies И b.\ P(b) \tag{37}$$

$$И b.\ P(b) \implies (\forall b.\ b\#P \implies P(b)). \tag{38}$$

For practical purposes these pair off naturally as (35) with (38) and (37) with (36). The first pair requires we find in the context a fresh b. The second pair introduces that fresh b, but only fresh for P. We can do better than this using (32) as in remark 16, so this latter pair seems less useful.

The complete algorithm is therefore: simplify using (11) to collect all И quantifiers in the hypotheses into one single quantifier. Also use (13) to draw negations under the И quantifier. Finally, apply the intro- and elim- rules (35) with (38), possibly augmented with remark 16 to generate a fresh name where necessary. Thus for example

$$\forall params.\ Иa.\ P(a) \wedge \neg Иa.\ Q(a) \implies \neg Иa.\ R(a)$$

simplifies to

$$\forall params.\ Иa.\ P(a) \wedge \neg Q(a) \implies Иa.\ \neg R(a),$$

a fresh b is introduced

$$\forall params,b.\ b\#params \wedge Иa.\ P(a) \wedge \neg Q(a) \implies Иa.\ \neg R(a),$$

the intro- and elim- rules reduce this to

$$\forall params,b.\ P(b) \wedge \neg Q(b) \implies \neg R(b),$$

and proof-search proceeds as normal. In the case that the fresh b is already in the context, as happened in (33), we use that supplied b instead. Call this algorithm `newsimp`. ◇

5 Difficulties implementing the algorithms

There are many technical difficulties putting the ideas of section 4 into practice.

18 (`split#`). `split#` is described in remark 15. The steps of the algorithm are:

1. repeated resolution with (26) followed by, when this fails,
2. resolution with (28).

There are difficulties with both steps.

1. Isabelle resolution with Isabelle unification is higher-order. $a\#f(x)$ unifies with a goal $a\#t$ for x matches t and f matches $\lambda x.x$ the identity, and we have a non-terminating loop. The solution is to write ML code to only allow this step when t is syntactically an application term `t1 $ t2`, and package this up as an Isabelle wrapper. An Isabelle wrapper, simplistically put, is an Isabelle theorem 'wrapped' in ML code which provides some intelligent control on how it may be applied, see remark 20.
2. It is impossible at object level to decide whether a term of the meta-level is closed or not. Again, we need an ML wrapper.

The algorithm `push` described in remark 14 is similar and also requires wrappers. ◇

19 (`newname`). To introduce a fresh b fresh for all variables $x_1, \ldots, x_n$ in the context, as we saw in remark 16, we must examine those names. This is, as in the previous remark, an operation on the meta-level syntax and must be implemented by an ML wrapper which examines that syntax. ◇

20 (Isabelle wrappers). We observed in remarks 19 and 18 that three significant FM features require ML wrappers in implementation (`split#`, `push`, and `newname`).

Isabelle proof proceeds imperatively by applying tactics to a proof-state. Simple tactics may apply a particular transformation to the state. More complex tactics will carry out some kind of proof-search. These automated tactics (written in ML) give Isabelle proving much of its power. They are all essentially tree-search algorithms of various kinds based on a library of Isabelle theorems which may be equalities, intro-rules, elim-rules, as the case may be. In inductive reasoning we use this automation to automatically handle the dozens if not hundreds and thousands of separate cases which a proof may entail. Wrappers are applied in between proof-steps and perform well as intelligent agents which may examine the way the proof-state is developing and perform for example some kind of garbage-collection.

But consider the example of `split#`. This is implemented as a wrapper as discussed above in remark 18 but morally it is clearly a pair of intro-resolution rules:

$$a\#f \wedge a\#x \implies a\#fx \quad \textbf{and} \quad a\#c \quad \textbf{if } c \textbf{ closed}.$$

In proof-search however `split#` will only be applied if none of the standard Isabelle theorems is applicable. We cannot, using wrappers, interleave it 'horizontally' with the standard Isabelle theorems, only 'vertically' with lower precedence, and in consequence proof-search is inefficient. Unfortunately there seems no cure other than dedicated FM proof-search ML code, or to hack existing code to hardwire algorithms such as `split#`, `push`, and `newsimp`. ◇

Now consider our treatment of the logic of И. This consists of equational theory such as (11) and (16), of intro- and elim- rules

$$a\#P,\ P(a) \implies И a.\ P(a) \quad \textbf{and} \quad (И a.\ P(a)),\ a\#P \implies P(a),$$

and of `newname` discussed in remark 17.

In this and in the equations immediately following we introduce two items of notation. $\bigwedge$ here is not a conjunction (as previously used written $\bigwedge_i prop_i$) but a meta-level Isabelle/Pure universal quantification ($\bigwedge x.\ prop(x)$). Also, a comma , denotes meta-level conjunction. I shall not be completely strict about distinguishing meta-level Pure from object-level HOL, but $\bigwedge$ and , where used will definitely denote the former.

As a simple example of a proof involving И consider a proof of

$$\bigwedge P,Q.\ И a.\ P(a),\ И a.\ Q(a) \implies И a.\ P(a) \wedge Q(a). \tag{39}$$

Inductive reasoning tends to be resolution-based, so we prefer an algorithm in that style. Accordingly we apply the intro- and elim- rules above, along with the conjunction intro-rule $A, B \Rightarrow A \wedge B$, to obtain

$$\bigwedge P, Q.\ P(?a(P,Q)), Q(?b(P,Q)) \implies ?a(P,Q)\#P$$
$$\bigwedge P, Q.\ P(?a(P,Q)), Q(?b(P,Q)) \implies ?a(P,Q)\#Q$$
$$\bigwedge P, Q.\ P(?a(P,Q)), Q(?b(P,Q)) \implies P(?a(P,Q))$$
$$\bigwedge P, Q.\ P(?a(P,Q)), Q(?b(P,Q)) \implies Q(?b(P,Q)).$$

Here $?a(P,Q)$ and $?b(P,Q)$ are unknowns which may be instantiated to any expression with free variables at most P, Q. The two freshness subgoals cannot be proved. We can use `newname` to introduce a fresh parameter into the context, but that only gives us

$$\bigwedge P, Q, b.\ P(?a(P,Q)), Q(?b(P,Q)), b\#P, b\#Q \implies ?a(P,Q)\#P$$

and $?a(P,Q)$ cannot be instantiated to b because b is a new free variable not amongst P, Q. Thus we need to apply `newname` before the resolution steps and then the proof succeeds.

In another situation such as proving (31) the fresh name may be provided by the previous proof-context and we certainly do not want to apply `newname`: it will cause unknowns to be instantiated to an irrelevant fresh parameter. It seems difficult to express a sensible and efficient compromise algorithm for this kind of proof-search.

In the rest of this section we step back and take a high-level view of these problems. In Isabelle and other theorem provers there are actually two kinds of variables. Free variables a, b, x, y and unknowns $?a, ?b, ?x, ?y$. Free variables are 'universal': they have an arbitrary value which ranges over all possible values. Unknowns are 'existential': they should, by the end of the proof-search, be instantiated to some specific term t. With these built into the meta-level of Isabelle/Pure the intro- and elim- rules for universal and existential quantification are easy to write. This need not be the case. For example in second-order λ-calculus existential quantification can be expressed using universal quantification. Theorem provers do not use this because it is nasty to work with in implementation.

It seems that the underlying problem may be that we are trying to encode using both a and $?a$ a kind of 'freshness' variable corresponding to the 'new' quantifier И. The fact that we need both reflects the $\forall/\exists$ duality of И mentioned in remark 10. Like unknowns, $?a$ a freshness variable depends on a context, for which it is fresh, and two sufficiently fresh freshness variables may be assumed equal, 'instantiated to each other', where convenient (think for instance of proving (11) or (12)). Like universals, freshness variables when introduced extend the context, and other terms and variables may depend on them if they are introduced later (e.g. other freshness variables). Trying to usefully express this in a dedicated logic belongs to future work.

6 The technical lemmas

This section can be skipped. For the interested reader we show a simple algorithm in action, constructed using the tactics developed in section 4. The point is that it neatly settles most of (23) to (32), which means we have a decent algorithm. We skip to the fourth one (26) $b\#x \wedge b\#f \implies b\#f(x)$.

Unfold definition 8 and apply `newname`. We obtain

$$\bigwedge b, x, f.\ Иc.\ (c\ b).x, Иc.\ (c\ b).f \implies Иc.\ (c\ b).f(x).$$

Apply `newname`

$$\bigwedge b, x, f, c.\ c\#b, x, f Иc.\ (c\ b).x, Иc.\ (c\ b).f \implies Иc.\ (c\ b).f(x)$$

then `newsimp` to obtain

$$\bigwedge b, x, f, c.\ c\#b, x, f(?c1(b,x,f,c)\ b).x = x, (?c1(b,x,f,c)\ b).f = f \implies (?c2(b,x,f,c)\ b).f(x) = f(x).$$

It is now simple to instantiate $?c1(b,x,f,c)$ and $?c2(b,x,f,c)$ to c, but we cannot apply `push` to the conclusion and finish the proof because $(?c2(b,x,f,c)\ b)$ is on the left, not the right. I had elided the following detail: transposition is invertible on each type by (5) so $x = (u_1\ u_2).y \Rightarrow (u_1\ u_2).x = y$. `push` applies this as an intro-rule, to "draw transposition to the right". With this elaboration the proof runs smoothly.

The proof of (27) runs along similar lines. To prove (28) $b\#c$ we unfold definitions and use `newsimp` to obtain

$$\bigwedge b, c.\ (?n(b,c)\ b).c = c.$$

If c is a closed term `push` solves this completely (otherwise proof fails, as it should).

(29), (30), (31), are proved by the same script as (26). In fact, the script also proves (27) though its behaviour for that goal, the path outlined in the previous paragraph, is a little special.

(32) underlies `newname`. The proof is best tailor-made. We rewrite it as $\exists b.\ b\#x \wedge \top$ and intro-resolve against (36) for $P = \lambda b.\top$, we now have $И b.\ b\#x$, an instance of the axiom (9).

7 The state of the implementation

An Isabelle/FM implementation exists but it is based on set theory rather than higher-order logic. This creates technical difficulties which ultimately proved insurmountable for the following reason. Consider the theorem

$$\mathbf{TyProd}(t_1,t_2) = \mathbf{TyProd}(t_2',t_2') \implies t_1 = t_1'.$$

In HOL this is rendered as

$$\mathbf{TyProd}(t_1^L,t_2^L) = \mathbf{TyProd}({t_2'}^L,{t_2'}^L) \implies t_1^L = {t_1'}^L$$

where we include all type annotations. In sets the same theorem is

$$\mathbf{TyProd}(t_1,t_2) = \mathbf{TyProd}(t_2',t_2'),\ t_1 \in L, t_2 \in L, t_1' \in L, t_2' \in L \implies t_1 = t_1'.$$

The difference is that when we intro-resolve against the HOL version we get one subgoal, whereas the sets version produces five (one each for each hypothesis of the implication, which must be established in order to apply it). A sets-based treatment of inductive datatypes overcomes this by implementing **TyProd** by some constructor which is is injective on the entire sets universe "by coincidence", probably $\mathbf{Inr}(\mathbf{Inl}(-))$. In FM this is not possible for various reasons which we now sketch.

Atoms must be marked as belonging to atomic type, the $И$ quantifier introduces fresh variables of atomic type which must be marked as such, and atom-abstraction $a,x \mapsto a.x$ (which we have not discussed in this paper, see [8, Section 6] or [6, Section 5]) is fundamentally non-injective so that the typing conditions can actually get quite complex.

Considerable ingenuity went into minimising the impact of these typing conditions in a sets environment (this should soon be the subject of a technical report). The price of using a HOL environment is precisely its benefit, the relative rigidity the typing gives the theory relative to sets, with both theoretical and practical consequences. In the recently-published [8] we provide what we hope is an elegant solution to the theoretical difficulties which will also be implementable, and it remains to try implementing the approach.

8 Conclusions

This paper has given a very simplified account of the problem of producing an implementation of a new foundational system FM with new and unfamiliar predicates and constructors. We considered two simple examples:

- Some theory of a datatype of types with universal types Σ and relations of α-equivalence for it $=_\alpha$ (defined using FM structure) and $=_\alpha{}'$ (defined in a more traditional style).
- Some technical FM lemmas (23) to (32).

These examples illustrated a fairly rich and representative selection of problems. We presented solutions to these problems and discussed their limitations. Another contribution of this paper is in what it elides: there are complications to automating FM which this paper has tried to bring out, but the short, slick, parts in between are the proof of how far we have already come.

References

1. FreshML homepage. `http://www.cl.cam.ac.uk/~amp12/research/freshml/index.html`.
2. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.
3. H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics (revised ed.)*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1984.
4. M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In *14th Annual Symposium on Logic in Computer Science*, pages 193–202. IEEE Computer Society Press, Washington, 1999.
5. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, Washington, 1999.
6. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 2001. Special issue in honour of Rod Burstall. To appear.
7. Murdoch J. Gabbay. *A Theory of Inductive Definitions with alpha-Equivalence*. PhD thesis, Cambridge, UK, 2000.
8. Murdoch J. Gabbay. FM-HOL, a higher-order theory of names. In *35 Years of Automath*. Heriot-Watt University, Edinburgh, Scotland, April 2002.
9. M. Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual Symposium on Logic in Computer Science*, pages 204–213. IEEE Computer Society Press, Washington, 1999.
10. James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3-4):373–409, 1999.
11. Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, 1988.
12. A. M. Pitts. Nominal logic: A first order theory of names and binding. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001, Proceedings*, volume 2215 of *Lecture Notes in Computer Science*, pages 219–242. Springer-Verlag, Berlin, 2001.
13. A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction. 5th International Conference, MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.
14. Isabelle Projects. http://www.cl.cam.ac.uk/users/lcp/Isabelle/projects.html.
15. J. Truss. Permutations and the axiom of choice. In H.D.Macpherson R.Kaye, editor, *Automorphisms of first order structures*, pages 131–152. OUP, 1994.