A Theory of Inductive Definitions With α-equivalence:

Semantics, Implementation, Programming Language.

A PhD thesis by Murdoch J. Gabbay, 10 August 2000.
DPMMS and Trinity College,
Cambridge University, England.

This document was compiled from LaTeX source on 10 August 2000. Copies will be printed, bound, and submitted for the title of PhD in Mathematics from Cambridge University, England. Other copies will be passed to those interested. Those interested are invited to write to me at Trinity College, Cambridge, or e-mail `m.j.gabbay@dpmms.cam.ac.uk`.

I remind the reader that my examiners may well suggest corrections to this document so it need *not* necessarily be the final version of my thesis. If the reader is wondering, DPMMS stands for the "Department of Pure Maths and Mathematical Statistics".

Other work published on this field is [**66**] and [**18**]. See also the homepage of Andrew Pitts, [**68**].

FOR THE LAYMAN: If you are not a layman I would not read this; the same things are said better in §1. Otherwise, and if you want to know what this pamphlet is about, read on. When we calculate, the representation we choose for the objects we calculate *with* makes a big difference. E.g. consider numbers; $1, 2, 3, \ldots$. Arithmetic using roman numerals is *much* harder than in base ten. There are many other representations, all good for different purposes; computers do base two better than base ten so microchips do binary arithmetic.

Numbers are a powerful tool. Another such tool is language (imagine this thesis without it). How do we represent sentences of a language? We can use a pen to write just as we can use our fingers with base ten. How about computers? Current theories of syntax on computers have problems. The reader might take the obvious inspiration from word processors and suggest representing sentences as strings of ASCII characters, but this is kludgy because it does not allow immediate access to the grammatical structure of the sentence (just as roman numerals do not seem in practice to allow immediate access to the arithmetic structure of numbers). There are many better representations, but all seem to suffer from problems with what I shall call, without explanation '***variable binding***'.

I present a new theory which allows us to represent languages *with* variable binding. In fact the presentation of the theory is 60 pages at most. The rest of this document tries to show that it works.

So: why is this thesis interesting? Considering how much we humans use languages, anything that lets computers (and us humans) manipulate them better is a Good Thing.

To Jim and Andrew,
and to Cambridge,
where anyone can learn anything.

Contents

Chapter I

Introduction

1. First words

This thesis is about inductive structures with α-conversion.

By '***inductive structures***' I mean the whole ensemble of (co)inductively defined sets[1], datatypes, and (co)inductive reasoning on them.

Inductive structures are ubiquitous, part of the basic toolkit of modern computer science. We see them in the abstract syntax of formal languages of all kinds, models of computer circuits ([**48**], the gates plug circuits together to form new circuits just like term-formers plug terms together to form new terms), constructive theories of finite sets (see [**59**, §2.9.2, p.58]), datatypes in programming languages, lists, queues and natural numbers in mathematical theory, evaluation and transition relations in operational semantics, typing judgements on formal languages, labelled transition systems in process algebras, theories of bisimulation, and so on. Inductive structures have even been made the basis of an underlying universe in CIC (the Calculus of Inductive Constructions, [**9**],[**77**]), with great success.

Why do we do this? Because it is useful. If we have an inductively defined set we have a set with a well-defined inductive reasoning principle, one which furthermore is easily generated on a computer in an automated manner. There are many different flavours but the basic recipe is the same: structural induction, pattern-matching, primitive recursion, recursion, coinduction, and so on.

Given this, is it not surprising that the world has continued turning while datatypes of abstract syntax, and thus all inductively defined structures arising from them, are not particularly amenable to inductive treatment? I discuss how this is so in §4 below.

In this document I present a novel solution to the problem of abstract syntax with variable binding. The solution, which ***I shall henceforth call*** 'FM', seems to offer significant advantages in simplicity and power over currently existing technology.

Before I get into maths, I make a few general comments about this document.

1. *Provenance of the material.* My thesis is the product of three years working with Andrew Pitts [**68**] in the Computer Laboratory of Cambridge University. It is divided into five parts of which Chapter I and Chapter V are the introduction and conclusion. Chapter II, Chapter III and Chapter IV are devoted respectively to the 'semantics', 'implementation' and 'programming language' of the title.

[1] When I write 'sets' I mean "sets or whatever fulfills the same function in the reader's favourite universe".

Pitts and I developed the material presented in Chapter II and Chapter IV together, I was entirely on my own in Chapter III. Roughly speaking, these three sections took me a year each.

2. *Notation.* I have typeset new definitions and other pertinent information in ***bold font***. ***Read this:*** To make it a little easier to find equations (some of which are named, not numbered) I have labelled references to them with the page on which the equation appears: $(\text{Fresh})_{35}$, $(204)_{205}$. Similarly for footnotes and figures: ft.2_{11},[2] Fig.2_{26}.

   I found LaTeX[3] had trouble typesetting references to lemmas, theorems and the like. They were too long and caused line overruns. So I shortened them. Thus R14.6 means 'Remark 14.6'. Similarly 'L' means 'Lemma' as in L8.1.4, 'T' means 'Theorem' as in T21.9, 'C' means 'Corollary' as in C9.2.9, and 'N' means 'Notation' as in N9.2.8. 'Fig.', '*p*' and 'ft' stand for 'Figure', 'page' and 'footnote' respectively.

   All of theorems, corollaries, lemmas, remarks, and so on—for this paragraph call them all 'theorems'—are numbered consecutively within subsections as 'Section.Subsection.Theorem Number'; thus T3.14.15 is the fifteenth theorem of the fourteenth subsection of section three. Sections are numbered consecutively *not* within chapters. Before the first subsection of a section LaTeX deems the subsection number to be zero (and if there is no subsection in a section, the subsection number is zero throughout it). In this case the zero is dropped; thus T3.14 denotes the fourteenth theorem of section three, which does not occur in a subsection.

3. *Abuses of grammar and language.*

   Remark 1.1. I have used the 'remark' environment (e.g. R1.1) to number important paragraphs so I can refer to them later. In this I may have abused English, since my 'remarks' are not necessarily 'asides'. In reflection of this grand stature and to make scoping clear, I typeset a symbol at the end, like this: ◇

4. *Underlying universe.* I shall mostly use sets and a little bit of category theory to develop the 'FM idea'—thus Chapter II introduces a set theory and interprets datatypes of syntax with binding as initial algebras in the corresponding category of sets. This is merely because I found the simplest presentation to be as sets and I *do* want my presentation to be simple. No mathematical or ideological commitment is implied. E.g. a dependently typed version of FM is an obvious target for future work (§35).

[2] Hello world.

[3] Thank you to the designers of LaTeX.

2. Thanks

The two heroes of the drama which has been my PhD studies are without doubt Jim Roseblade and Andrew Pitts. Jim taught me ring theory as an undergraduate and has been not only a gracious teacher and friend over the past seven years, but also very useful. He put me in touch with Andrew Pitts. Without Jim, what the reader has before them would most probably not exist. For that and a great deal more, I thank him.

Andrew has been my teacher for the past three years, and he has been a good one. I am a demanding student and I have imposed on him more than I had a 'right' to. He never complained, on the contrary, his attention has been positively lavish. When I told him how I appreciated this he accepted the thanks but pointed out that he is unusually interested in the material of this thesis and therefore gave it particular attention. I think this need not change my gratitude in the slightest.

Other people who have taken significant time and trouble over me are: Martin Hyland for getting me the PhD position and keeping my set theory honest. Larry Paulson whose matter-of-fact willingness to help, when he could, has sometimes verged on the surreal (Larry also sent me style and LaTeX files which were useful producing this document). Tobias Nipkow for a very pleasant and informative correspondence including but not limited to Isabelle. John Harrison and Konrad Slind for discussing HOL and theorem-proving systems with me. Randy Pollack for our discussions on LEGO and my implementation. Tom Forster for discussing set theory with me. All these people deserve respect for their knowledge, and I thank them for sharing it with me. But they also displayed an interest for my work which did as much good for my soul as the knowledge they imparted did for this document. To these people and some others I have not named, thank you.

I would also like to thank—again—Andrew Pitts, Larry Paulson, Tobias Nipkow and Martin Hyland, this time for reading and commenting on sections of the thesis. I have considered all and acted on nearly all their suggestions. I hope they will not hold the difference against me.

3. Aims and target group

I have invented something new. Above all this document must be a thesis presenting this novel work for a PhD degree, but I have tried also to accommodate the following design aims:

1. I want to disseminate knowledge of my invention in the hope that people will use it. I have tried to keep things readable.

2. My target group is computer scientists, not necessarily theoretically inclined. Thus I assume familiarity with functional programming languages, in particular ML ([**57**], [**54**]). I do not assume knowledge of set theory[4], category theory, or any other underlying universe except where this is absolutely necessary.

3. Working against this is the fact that as a PhD thesis this document is expected to include for future reference the details of proofs too malignant to inflict upon the readerships of journals.

4. FreshML, first pass

This section tries to put the rest of the thesis in context by showing what the general ideas and problems are, and informally describing the kind of solution FM proposes.

In Chapter IV we design an ML-like programming language FreshML with facilities for handling datatypes of abstract syntax with variable binding.[5] FreshML is designed using FM set theory, described in Chapter II. Here I motivate both by discussing the problems they address. Other solutions include de Bruijn indices and Higher Order Abstract Syntax (HOAS) but a discussion of them would disturb the flow of the development and has its own section §33.

Consider two datatypes:

```
datatype Nat   = ZERO of unit
               | SUCC of Nat;
datatype oNat  = oZERO of unit
               | oONE  of unit
               | oSUCC of oNat;
```

Observe that `Nat` and `ONat` have identical semantics $\mathbb{N}$, the set of natural numbers.[6]

If their semantics are identical, what's the difference between them? It lies in the programs we can write, or put another way the functions we can define out of them. For example it is a little harder to define $\lambda x.x+1$ on `oNat` than it is on `Nat`:

[4] Beyond the basic idea of a set as a collection of objects axiomatised in some first-order language. See [**39**] and [**40**].

[5] Note that FreshML is designed to be easy to prove things about, in particular T21.9. It is not designed necessarily to be easy to program in. A real programming language (also called 'FreshML') is future work under development, see [**66**].

[6] Of course the semantics of programming languages are far more complicated than that but it is irrelevant here.

The reader may also prefer to think in terms of, say, categories rather than sets. This makes no odds here, and since I shall go on to construct a set theory (Chapter II) I choose to think in sets. There is some discussion of presheaf models of this work in [**18**, §6], I do not expand on that work in this thesis.

```
fun add1  x = SUCC(x);
fun oadd1 (oZERO()) = (oONE())
  | oadd1 (oONE())  = (oSUCC(oZERO()))
  | oadd1 (oSUCC x) = (oSUCC (oadd1 x));
```

On the other hand, it's a little harder (barely) to define the predicate "is even" on `Nat`:

```
fun iseven (ZERO())        = true
  | iseven (SUCC(ZERO()))  = false
  | iseven (SUCC(SUCC(x))) = iseven(x);
fun oiseven (oZERO())  = true
  | oiseven (oONE())   = false
  | oiseven (oSUCC(x)) = oiseven(x);
```

When we design programs to manipulate datatypes our first design choice is the datatype's structure. A good one makes programs easy to write. A bad one can make them almost impossible.

We now try to construct the syntax of the untyped λ-calculus.

Definition 4.2 (`lam1`)**.** *We declare in our informal language:*

```
datatype lam1 = Var1 of Nat
              | App1 of lam1 * lam1
              | Lam1 of Nat * lam1;
```

So (semi-formal) typing rules for forming values are:

$$(1) \qquad \frac{\texttt{V:Nat}}{\texttt{Var1(V):lam1}} \qquad \frac{\texttt{V1:lam1 V2:lam1}}{\texttt{App1(V1,V2):lam1}} \qquad \frac{\texttt{V1:Nat V2:lam1}}{\texttt{Lam1(V1,V2):lam1}}.$$

Can we define substitution `subst1:lam1*Nat*lam1->lam1`, which given arguments `t2,x,t1'` calculates the syntax obtained by substituting `t2` for `x` in `t1'`?

```
fun subst1 (t,x,Var1(y))     = if x=y then t else (Var1 y)
  | subst1 (t,x,App1(s1,s2)) = App1(subst1(t,x,s1),subst1(t,x,s2))
  | subst1 (t,x,Lam1(y,s)) =
       if (x=y) then Lam1(y,s) else UNKNOWN;
```

The problem, and it is a familiar one, is variable capture. In the third clause we cannot blindly replace every `x` in `s` by `t` because it may be that there are occurrences of `y` in `t` that would be wrongly 'captured' by the outermost binder `Lam1(y,-)`. Let us consider an example of this in abstract terms:

$$[y/x](\lambda y.x) \neq \lambda y.y$$

So what *is* $[y/x]\lambda y.x$? The traditional response is

> "$[y/x]\lambda y.x$ is equal to $\lambda y'.y$ where y' is any variable symbol that *isn't* equal to y."

This should give us pause for thought for two reasons. Firstly we're having trouble defining the substitution function on our datatype `lam1`. Maybe we somehow have the wrong datatype. Secondly, do we really not care what y' is? If so evaluation is nondeterministic.

We now abandon our pseudo-programming-language and consider just the underlying semantics of the datatypes in question. As commented in ft.6_{13} we ignore the complexities of real programming languages and consider a very simple semantics in sets. The semantics of `lam1` is therefore a set which we shall write

$$\textbf{lam1} = \mu X.\textbf{Var1 of } \mathbb{N} \mid \textbf{App1 of } X \times X \mid \textbf{Lam1 of } \mathbb{N} \times X. \tag{2}$$

This is notation for the set built up using only the rules

$$\frac{x \in \mathbb{N}}{\textbf{Var1}(x) \in \textbf{lam1}} \quad \frac{t1, t2 \in \textbf{lam1}}{\textbf{App1}(t1, t2) \in \textbf{lam1}} \quad \frac{x \in \mathbb{N}, t \in \textbf{lam1}}{\textbf{Lam1}(x, t) \in \textbf{lam1}}. \tag{3}$$

Remark 4.3 (Choice of semantics)**.** We have a fairly free choice for **Var1**, **App1** and **Lam1** so long as they are injective and pairwise disjoint on the set-universe, meaning that (for example) $\textbf{App1}(x, y) \neq \textbf{Lam1}(u, t)$ for all sets x, y, u and t. Reasonable choices would be $\textbf{Inl}(-)$, $\textbf{Inr}(\textbf{Inl}(-,-))$ and $\textbf{Inr}(\textbf{Inr}(-,-))$ where $\lambda x.\textbf{Inl}(x)$ and $\lambda x.\textbf{Inr}(x)$ are left and right injection functions into set disjoint sum implemented perhaps as $\lambda x.(x, \emptyset)$ and $\lambda x.(x, \{\emptyset\})$. We go into no more detail on the matter. We construct such sets in §10.2. ◇

Remark 4.4. We now have two objects associated to a term `t:lam1`:

1. The syntax of the term itself, `t`, which for this document is an abstract syntax tree in a computer or other external universe.
2. Its semantics in the set **lam1**, written $[\![\texttt{t}]\!]$ or just $\mathbf{t}$.

There is a third object floating around which I shall call the *denotation* of `t`. We build our datatypes to reflect in a formal framework entities taken from nature. In the case of `lam1` we are trying to capture the untyped λ-calculus, so the denotation of `Lam1(a,Var1(a))` is the function $\lambda a.a$ living "in nature". ◇

We shall adhere to the following convention:

Notation 4.5 (`Syntax` and **Semantics**)**.** *We use typewriter font for syntax like* `lam1` *and* `App1(Var1(a),Var1(a))`*. We write their set-theoretic semantics/implementations* $[\![\texttt{lam1}]\!]$ *or* ***lam1****.*

In summary, we have ***syntax*** written `Lam1(a,Var(a))`, ***semantics*** written **Lam1(a,Var1(a))**, and ***denotation*** written $\lambda a.a$.

Remark 4.6 (Terminology confusing)**.** The standard terminology for these three objects can be confusing. The denotational object containing $\lambda a.a$ and

$\lambda a.\lambda b.a$, in other words the untyped λ-calculus, is routinely called "the syntax of the untyped λ-calculus". It is completely different from the *syntax* of terms of the datatype `lam1`. To complete the confusion, the set **lam1** is often referred to as a "set of syntax". But really the same situation exists with, say, the number two, which exists as (at least) three things: arithmetic expression `2` (syntax), set $\{\emptyset, \{\emptyset\}\}$ (semantics), and the number 2 (denotation). ◇

Remark 4.7 (Proper semantics)**.** When we say we should quotient `lam1` by α-equivalence, we mean:

> "The proper semantics for the syntax of the λ-calculus with variable binding is (a set isomorphic to) $\mathbf{lam1}/{=_\alpha}$, where **lam1** is the set constructed in $(3)_{15}$ and $=_\alpha$ is α-equivalence on it."

◇

The semantics of `lam1` is not $\mathbf{lam1}/{=_\alpha}$, it is **lam1**. This is one reason we have trouble defining substitution on it. We need a datatype with semantics $\mathbf{lam1}/{=_\alpha}$. But also remember the example of `Nat` and `oNat`: it is certainly necessary that the semantics of our datatype be isomorphic to $\mathbf{lam1}/{=_\alpha}$, but *in addition* it must be defined with an inductive structure that allows us to write the programs we wish to write.

Many realisations of such datatypes exist in the literature, see §33. As promised at the beginning of the section we concentrate on FreshML here.

The solution we propose is this: we introduce into the language a new '***type of atoms***' written `Atm`. It has no term-formers so all values of type `Atm` are variables `x`. The intended semantics $[\![\texttt{Atm}]\!]$ is an infinite set of distinct symbols with no internal structure which we shall usually write $\mathbb{A}$. They are meant to represent variable symbols but to avoid confusion with the variable symbols of the overlying language we call them ***atoms***. We let ourselves compare atoms for equality and make `Atm` an equality type so we have the term-forming rule

$$\frac{\texttt{x,y:Atm t1,t2:Atm*Atm-->T}}{\texttt{if x=y then t1(x,y) else t2(x,y) :T}}.$$

Now we introduce a type former "***abstraction type of*** `X`" written `[Atm]X`. The rule for forming terms of type `[Atm]X` is

$$\frac{\texttt{a:Atm x:X}}{\texttt{a.x:[Atm]X}}. \tag{4}$$

The intended semantics of `a.t` is "$[\![\texttt{t}]\!]$ with the atom $[\![\texttt{a}]\!]$ bound", the intended semantics of `[Atm]X` is "the set of `a.t` for `a:Atm` and `t:X`". Whatever these really are, we write them $a.t$ and $[\mathbb{A}]X$ so that $[\![\texttt{a.t}]\!] = a.t$ and $[\![\texttt{[Atm]X}]\!] = [\mathbb{A}]X$.

Consider what this means for the type `[Atm]Atm`. Terms of it are pairs `(a,b)` written `a.b`. Their semantics are more subtle. Suppose `a`,`b` and `c` are variables of type `Atm` representing pairwise distinct atoms in $[\![\texttt{Atm}]\!]$, write them a, b and c. Then $a.a$ is supposed to be a with a bound, so it is equal to $b.b$ and $c.c$. $a.c$ is c with a bound. Because $a \neq c$ and $b \neq c$ this is equal to $b.c$ but not equal to $c.c$. This behaviour is precisely that of λ-terms, where $\lambda a.a$, $\lambda b.b$ and $\lambda c.c$ are all equal up to α-equivalence, and distinct from $\lambda a.c$ and $\lambda b.c$, which are equal.

Now consider another version of a datatype for the λ-calculus:

Definition 4.8 (`lam2`)**.** *We declare in our informal language:*

```
datatype lam2 = Var2 of Atm
              | App2 of lam2 * lam2
              | Lam2 of [Atm]lam2;
```

(We shall repeat this rigorously in D10.3.4.) So (semi-formal) typing rules for values of this type are

$$(5) \qquad \frac{\texttt{V:Atm}}{\texttt{Var1(V):lam2}} \quad \frac{\texttt{V1:lam2 V2:lam2}}{\texttt{App1(V1,V2):lam2}} \quad \frac{\texttt{V:[Atm]lam2}}{\texttt{Lam2(V):lam2}}.$$

Remark 4.9 (Values of `lam2`)**.** Comparing $(1)_{14}$ and $(5)_{17}$ we see the difference is in `Lam1` and `Lam2`. It makes little difference to the syntax of the language: values of type `[Atm]lam2` are by $(4)_{16}$ precisely `a.V` for `V:lam2`. The values of `lam2` are in 1-1 correspondence with those of `lam1`, we just write `Lam2(a.t)` instead of `Lam1(a,t)`. *The semantics are different.* Equalities like $a.a = b.b$ mean that **lam2** 'quotients itself' by α-equivalence as it is built (cf. D10.3.4). ◇

Remark 4.10 (Sanity of semantics)**.** Since the semantics of `Lam2(a.a)` and `Lam2(b.b)` are equal we expect them to be contextually equivalent. Our language is unformalised so we cannot pursue this question further here. We return to it in Chapter IV and in particular T21.9. ◇

Instead, let us consider some examples of how we can use the apparatus we have already assembled to write some programs.

```
fun eval2 (Lam2(t'))    = Lam2(t')
  | eval2 (App2(t1,t2)) =
       case (eval2(t1)) of Lam2(s') => eval2(subst2(t2,s'));
val eval2 = fn : lam2->lam2
```

Here variables of abstraction type have names tagged with a `'` for easy identification. This code says

1. An abstraction `Lam2(t')` evaluates to itself.

2. To evaluate `App2(t1,t2)`, evaluate `t1`. If it does not evaluate to an abstraction `Lam2(s')` raise an exception. If it does, substitute `t2` for the distinguished bound variable of `s'` and proceed with evaluation.

So `subst2:lam2*[Atm]lam2->lam2` should have semantics a function that substitutes a term for the distinguished bound atom of an atom-abstracted term.

Recall the typing of `subst1` as `lam1*Nat*lam1`. That function carries out capture-avoiding substitution `t` for `a` in `s`. We could formulate a function `subst2':lam2*Atm*lam2` along similar lines, but since we now have abstraction types we might as well use them.

We can implement `subst2` as follows:

Definition 4.11. *We declare a function* `subst2` *in our informal language as follows:*

```
fun subst2 (t,a.(Var2 b))     = Var2(b)
  | subst2 (t,a.(Var2 a))     = t
  | subst2 (t,a.(App2 s1 s2)) = App2(subst2(t,a,s1),subst2(t,a,s2))
  | subst2 (t,a.(Lam2 b.s))   = Lam2(b.subst2(t,a,s));
val subst2 = fn : lam2*[Atm]lam2 -> lam2
```

This declaration brings us to the question of patterns of abstraction type. They are of the form

$$p ::= \mathtt{x} \mid \mathtt{Con}\vec{p} \mid \mathtt{a}.p.$$

It is usual to only allow linear patterns (patterns such that each variable symbol `x` appears at most once) so as not to make all types equality types by the back door, e.g. consider this ***false*** polymorphic declaration:

```
FALSE fun eqfun (x,x) => true
        | eqfun (x,y) => false;
```

We now relax this restriction and admit nonlinear patterns *on the condition* that the nonlinear variables are typed as `Atm`. Since `Atm` is intended to be an equality type is this not a problem.

Remark 4.12 (Freshness)**.** We also place a '***freshness***' or '***newness***' ***condition*** on the `a` in the pattern `a.p`. We can think of it as meaning

> "When inventing a name for a bound atom, we may as well choose a completely fresh one. So when we pattern-match some atom-abstraction against `a.p` we can assume `a` is not free in anything else in the context at the time we choose it."

◇

This is reminiscent of Barendregt's 'variable convention', see [**1**]. He is working in ZF and instructs the reader to assume that names of bound and free names of variables are distinct. The idea there being, as here, that "we might as well". However, the slogan above is more than just a convention because it occurs in the rigorous context of a new set theory. We make the idea of "choose a fresh atom" rigorous via the И quantifier in §9.4. See also R12.3.2, which restates R4.12 rigorously. We shall also see a mechanism allowing programmers to explicitly insist that an atom be chosen fresh in FM (**fresh** $a.\ t$, see C9.6.7) and later in FreshML (`fresh` a **in** t, see §22.1).[7]

We can now read `subst2` as follows: the name of the variable for which to substitute `t` is passed to `subst2` bound and therefore nameless. The pattern-matching at abstraction types in each clause in the definition of `subst2` creates a new name `a` for this bound variable. Then the clauses read:

1. The 'freshness' condition (R4.12) means that `a` and `b` can be assumed nonequal, so return `Var2(b)`.
2. If `a` *is* the variable under `Var2`, output `t`.
3. Substitution distributes over application.
4. Choose an entirely new name for the other bound variable `b`, take the resulting inner term `t` and substitute in it for `a` as shown.

Compare `lam2`, `subst2` and `eval2` above with the corresponding definitions `term`, `subst_bound` and `betaapply`, see Fig.1_{20}, which are at the heart of Isabelle98-1. `term` is a λ-calculus but implemented in de Bruijn style and not the naïve datatype of `lam1`. It is also a typed and with some bells and whistles so the comparison between it and `lam1` or `lam2` is not direct. Nevertheless, the definition of `subst_bound` requires three ancillary functions where `subst2` is defined directly.

Remark 4.13 (FM sets)**.** We are now ready to construct an underlying universe within which to give all this a rigorous semantics. We shall call it FM ***set theory*** (cf. Item 4 on *p.*11). It contains a set of atoms $\mathbb{A}$ intended to model the type of atoms `Atm` (see *p.*16) along with axioms to make sense of the abstraction types `[Atm]X` (see *p.*16). And there will be more, such as a new term-former **fresh** (C9.6.7) and a new quantifier И (D9.4.2). ◇

Remark 4.14 (Name-carrying, nameless, nameful)**.** I shall refer to a treatment of syntax as '***name-carrying***' when we deal with terms containing names of bound variables. Thus for example in D4.2 we build a set of syntax **lam1** of

[7] In our paper [**66**] Pitts and I discuss the design of an ML-like language similar to this one in much more detail. There we use linear patterns and a system of '***guards***' on them to achieve more-or-less the same effect as the nonlinear patterns and freshness condition (R4.12) used here.

```
datatype term =
    Const of string * typ
  | Free  of string * typ
  | Var   of indexname * typ
  | Bound of int
  | Abs   of string*typ*term
  | op $  of term*term;
...
(*increments a term's non-local bound variables
  required when moving a term within abstractions
     inc is  increment for bound variables
     lev is  level at which a bound variable is considered 'loose'*)
fun incr_bv (inc, lev, u as Bound i) = if i>=lev then Bound(i+inc) else u
  | incr_bv (inc, lev, Abs(a,T,body)) =
        Abs(a, T, incr_bv(inc,lev+1,body))
  | incr_bv (inc, lev, f$t) =
      incr_bv(inc,lev,f) $ incr_bv(inc,lev,t)
  | incr_bv (inc, lev, u) = u;

fun incr_boundvars  0  t = t
  | incr_boundvars inc t = incr_bv(inc,0,t);
...
(*Substitute arguments for loose bound variables.
  Beta-reduction of arg(n-1)...arg0 into t replacing (Bound i) with (argi).
  Note that for ((%x y. c) a b), the bound vars in c are x=1 and y=0
        and the appropriate call is  subst_bounds([b,a], c) .
  Loose bound variables >=n are reduced by "n" to
     compensate for the disappearance of lambdas.
*)
...
(*Special case: one argument*)
fun subst_bound (arg, t) : term =
  let fun subst (t as Bound i, lev) =
            if i<lev then  t    (*var is locally bound*)
            else  if i=lev then incr_boundvars lev arg
                           else Bound(i-1)  (*loose: change it*)
        | subst (Abs(a,T,body), lev) = Abs(a, T,  subst(body,lev+1))
        | subst (f$t, lev) =  subst(f,lev)  $  subst(t,lev)
        | subst (t,lev) = t
  in  subst (t,0)  end;

(*beta-reduce if possible, else form application*)
fun betapply (Abs(_,_,t), u) = subst_bound (u,t)
  | betapply (f,u) = f$u;
```

FIGURE 1. Substitution in Isabelle98-1/Pure

the λ-calculus and interpret the abstraction type as `Nat*X` or `Atm*X`. That was a name-carrying representation. Things can get a little ambiguous: we might class $\mathbf{lam1}/{=_\alpha}$ (R4.7) as name-carrying even though terms are quotiented by α-equivalence. The reason is that the structure of the quotient set forces us to take representatives most of the time we want to prove anything useful.

Because FM datatypes of syntax are up to α-equivalence and do *not* contain the names of bound variables, I shall call them '***nameless***' or (of course) '***FM***'.

This is accurate but misleading because FM allows us to give bound variables names at will. We saw this for example in the last clause of D4.11 where we invent the name `b` for a bound variable in an abstraction `b.s`. I shall call this 'ZF-***style***' or, following a suggestion by Pitts, '***nameful***' reasoning. Thus the slogan is

FM has ***nameless*** terms but permits ***nameful*** reasoning on them.

We continue this discussion after we develop FM, see R12.1.3 and later on R32.1.3. The first concrete illustration of the slogan above is in R12.4.1.

Of course, the original nameful reasoning arises when the datatype is of (standard ZF) name-carrying terms like `lam1` (D4.2), cf. §33.4. ◇

5. A brief resumé of ...

This section is *not* an overview of the thesis as such, for that see §6 below.

I have informally discussed what this thesis is about in §4. Now, in this section, I try to give my reader some idea of what the rigorous mathematics to follow looks like *without* said reader having to read said rigorous mathematics in full.[8]

5.1. ... FM-**logic.** FM-logic is First Order predicate Logic (FOL)—not a *particular* FOL though I assume my FOL to be classical and typed—augmented to facilitate reasoning about syntax with binding. FM-logic is extended with one or more ***types of atoms***, write one of them $\mathbb{A}$ (D8.1.1) as an example. Types of atoms are intended to represent variable symbols in the underlying syntax. We may need many of them because the underlying language may have many types of variables (type variables, term variables, even sets of constant symbols, etc, see R16.1.2.).

FM-logic is also augmented with a quantifier $\mathsf{И}$ (D9.4.2), used as follows:

$$\mathsf{И}a \in \mathbb{A}.\ \phi(a).$$

The reading of this is "for all/some new atom(s) a, $\phi(a)$ holds". 'All' will be called the '***universal*** ***reading of*** $\mathsf{И}$. 'Some' will be called the '***existential*** ***reading of*** $\mathsf{И}$ (T9.4.6, R9.4.10).

Note that 'new' and 'fresh' are used synonymously. So we shall also read the formula above as "for all/some ***fresh*** atom(s)".

By the universal reading we have that $\mathsf{И}$ distributes over conjunction. By the existential reading we have that $\mathsf{И}$ distributes over disjunction. Combining the two readings we have that $\mathsf{И}$ even commutes with negation and therefore implication.

[8] Though that would be jolly nice.

Thus И commutes freely with the propositional part of FOL (C9.4.5, C9.4.11). И interacts less trivially with the predicate part of FOL (R9.4.12).

И seems to be unique to FM, I am not aware of it or anything like it in other approaches to syntax with binding—the notable exception being Pollack's work (§33.5), in which the spirit of И is very visible. It is mostly through И that FM gains its particular 'personality'. И really is astonishingly well-behaved, and as a direct result of that the treatment of binding in FM comes out particularly clean and easy.

The down-side to И is that, being unique to FM, the reader will not recognise it. Some people take this as a threat; "the simple things were discovered long ago, so if I don't recognise it, it's complicated". This is not the case, this thesis is elementary mathematics. Technical in places, as it must be, but elementary nonetheless.

5.2. ... FM**-sets.** It is FM set theory constructed in Chapter II which gives and FM-logic a rigorous semantics (as assertions about FM sets). In particular of course И has a semantics which 'justifies' it and puts it in the context of a variety of set-operators, including a permutation action (D8.1.8) and **Supp** and # (N9.2.4).

FM set theory has abstraction types $[\mathbb{A}]X$ (D9.6.1). They play the same rôle as Plotkin's δ (§33.6), X in de Bruijn, $\mathbb{A} \times X$ in name-carrying (R4.14) representations, and $\mathbb{A} \to X$ or $X \to X$ in HOAS. $[\mathbb{A}]X$ has one term-former $a.t$ (D9.5.1) which takes a term t and atom a and creates a term $t_* = a.t$ which is t with the name of the variable symbol a 'bound'. It genuinely *is* bound in the sense that, for one simple example $a.a, b.b \in [\mathbb{A}]\mathbb{A}$ are identical sets. Throughout this document we label abstractions with a star as in "$x_* = a.x \in [\mathbb{A}]X$" (N9.5.3). FM also provides a destructor @ (D9.5.14) which takes an abstraction t_* and an atom a and returns $t_*@a$, which is the term obtained by giving the bound variable in t_* the name a.

$[\mathbb{A}]X$ displays particularly nice algebraic properties similar to (and inherited from) those of И. It commutes (up to set-isomorphism) with product $X \times Y$, disjoint sum $X + Y$ *and* function types $X \to Y$ (C9.6.9).

At this point everything comes together and we build quite an extensive theory of datatypes-with-binding in FM sets (§10). The datatypes obtained have nameless terms because of the way $[\mathbb{A}]X$ destroys the name of abstracted atoms. However, we can obtain ordinary-looking inductive principles (e.g. $(45)_{68}$) by using И and @ to choose new names for the bound atoms and thus manipulate these nameless

terms as if bound variables did have names. The good behaviour of И makes it work.

Incidentally, we can move И from FM-logic to FM-programming. We introduce a term-former **fresh** used as **fresh** $a.f(a)$ (C9.6.7). This invents a new name a and uses it to calculate $f(a)$. The choice of a is supposed to be arbitrary so we insist for **fresh** $a.f(a)$ to be well-formed that x be bound in $f(x)$. И and **fresh** are connected by $(32)_{52}$.

In the course of this thesis we actually devote a lot of attention to this nascent programming language. An informal discussion is in §4. The various term-formers are given rigorous (and sometimes technical) set-theoretic semantics throughout Chapter II some of which I have just referenced. We also see many examples of programs in §22.2 and §24.2. Of course all of Chapter IV is devoted to a particular programming language FreshML.

I discuss inductive reasoning in FM in a relatively nontechnical but still rigorous way in §12.

6. Overview of thesis

I conclude Chapter I with an 'overview of the thesis'. This section is nothing more than a systematic annotated table of contents.

In §4 I discuss the problem this thesis addresses and introduce an informal ML-like language designed to motivate the rest of the document. In §5 I resumé a few of the more technical details of the solution. Finally, in this section, I give an overview of the thesis, section by section.

In Chapter II I introduce FM set theory. It is based on an underlying set theory ZFA (see §8) extended with an extra axiom $(\text{Fresh})_{35}$ which turns ZFA into FM (see §9). In §10 I discuss datatypes of syntax with binding by constructing a rigorous framework for declaring and building them in FM. In §11 I pause to ask a few questions about why I did things in the way I did. In §12 I return to the theme of datatypes and use §10 to build a datatype for a (deliberately) trivial programming language. I then use it as a concrete example with which to illustrate inductive reasoning on datatypes *with binding*. Finally in §13 I tie up a few loose ends.

In Chapter III I discuss my Isabelle implementation of FM sets, Isabelle/FM. In its gross structure Isabelle/FM mirrors FM set theory. It is constructed as Isabelle/ZFQA (§15) extended with an extra axiom to full Isabelle/FM (§16). The two sections are very different because the first concerns re-engineering an existing theory, and the second constructing one from scratch. In §17 I discuss one or two simple ideas which helped me control and efficiently use the Isabelle

system. In §18 I discuss some ideas that did not work, and why. In §19 I discuss one of my attempts to use Isabelle/FM to declare a real datatype with binding and how well it worked. In §20 I summarise the snags remaining in Isabelle/FM and explain how to eliminate them.

Chapter IV concerns the design of a programming language with datatype declarations and associated term-formers and destructors allowing programming with syntax with binding. A design of an industrial strength ready-to-use language is *not* given, that is a topic of future research. Instead I introduce a much simpler language FreshML, designed to be amenable to a rigorous theoretical analysis and in particular a proof of a technical correctness result T21.9. The syntax is defined in §22.1 and some reasonably nontrivial programs expressed using it in §22.2. Typing judgements are developed in §23 and §24. A big-step evaluation judgement is introduced in §25. This all lays the groundwork for §26, §27 and §28, which between them define notions of operational and contextual equivalence and prove them equal. Note that §26.6 gives an overview of the whole proof from about one-third of the way in. After all that, it is easy in §29 to restate and prove our correctness result. §30 records the reasons for some of my design decisions and §31 discusses how all of Chapter IV could (and should) be automated using Isabelle/FM.

Chapter V reviews some of the problems of FM (§32), compares FM to other approaches to syntax with binding (§33), describes what I see as the main accomplishments of my thesis (§34), and concludes with a 'to do' list (§35).

Chapter II

Semantics: FM set theory

(Sets)	$\forall x, y.\ x \in y \to y \notin \mathbb{A}$
(Extensionality)	$\forall x, y \notin \mathbb{A}.\ (\forall z.\ z \in x \leftrightarrow z \in y) \to x = y$
(Collection)	$\forall x.\ \exists y \notin \mathbb{A}.\ \forall z.\ z \in y \leftrightarrow (z \in x \wedge \phi) \quad (y \text{ not free in } \phi)$
($\in$-Induction)	$(\forall x.\ (\forall y \in x.\ [y/x]\phi) \to \phi) \to \forall x.\ \phi$
(Replacement)	$\forall x.\ \exists z.\ \forall y.\ y \in z \leftrightarrow \exists x'.\ (x' \in x \wedge y = F(x'))$
(Pairset)	$\forall x, y.\ \exists z.\ x \in z \wedge y \in z$
(Union)	$\forall x.\ \exists y.\ \forall z.\ z \in y \leftrightarrow (\exists w \in x.\ z \in w)$
(Powerset)	$\forall x.\ \exists y.\ \forall z.\ z \in y \leftrightarrow \forall w \in z.\ w \in x$
(Infinity)	$\exists x.\ \exists y.\ y \in x \wedge \forall y \in x.\ \exists w \in x.\ y \in w$
(AtmInf)	$\mathbb{A} \notin pow_{\text{fin}}(\mathbb{A})$

F any function-class, ϕ any predicate defined in the logic of ZFA.

FIGURE 2. D8.1.1 - Axioms of ZFA

---

7. Introduction

FM set theory is similar to ZF set theory except in so far as it is different[9], which is not terribly far. It will in due course provide semantics for all the phenomena discussed in §4. The underlying logic of FM set theory is identical to that of ZFA, ZF set theory with atoms[10], so we start with that.

8. ZFA set theory

8.1. Axioms of ZFA.

Definition 8.1.1 (ZFA)**.** *The underlying logic of* ZFA *is the usual first-order logic with equality. Its signature contains just a binary predicate set membership* $\in$ *and a constant symbol* $\mathbb{A}$*. The axioms of* ZFA *are presented in Fig.*2_{26}*.*

Notation 8.1.2 ($\in$-related elements)**.** *We may write "x* ***is*** $\in$***-related to*** *y" meaning 'x* $\in$ *y'.*

Of the rules in Fig.2_{26} only $(\text{AtmInf})_{26}$ is unusual.

---

[9]The basis of mathematics is the tautology.

[10]See the index of [**72**] under 'individuals'. If that work did not introduce atoms, it discusses them as if they were new.

Notation 8.1.3. *$pow_{\text{fin}}(X)$ is the set of finite subsets of X defined by induction using two rules*

$$\emptyset \in pow_{\text{fin}}(X) \quad \textit{and} \quad U \in pow_{\text{fin}}(X) \wedge x \in X \Rightarrow U \cup \{x\} \in pow_{\text{fin}}(X).$$

Throughout this document the meaning of "x is a finite set" is precisely "$x \in pow_{\text{fin}}(x)$". This is in fact equivalent in ZFA *and later in* FM *to "x is in bijective correspondence with a subset of the natural numbers", proof omitted. These issues are important because* FM *set theory (§9) lacks AC (§11.4).*

So $(\text{AtmInf})_{26}$ insists that $\mathbb{A}$ is not in its own own set of finite subsets.[11] Otherwise the theory is standard.[12] $\mathbb{A}$ clearly corresponds to the 'type of atoms' `Atm` mentioned on *p.*16. Here are some of its properties:

Lemma 8.1.4 (Atoms empty)**.** *Atoms $a \in \mathbb{A}$ have no $\in$-related elements; they are empty. There is a unique empty set which is not an atom. This is the empty set $\emptyset$.*

Proof. From the axioms of Fig.2_{26}. □

Remark 8.1.5 (Atoms are sets)**.** Some distinguish two classes in the underlying model: 'Atoms' which are elements of $\mathbb{A}$, and 'Sets', which are everything else. I disapprove. In this document all elements of the underlying model of a set theory are 'sets', including elements of $\mathbb{A}$ which are also called 'atoms'. Thus atoms are sets. ◇

Notation 8.1.6 ($\mathcal{V}_{\text{ZFA}}$)**.** *Write $\mathcal{V}_{\text{ZFA}}$ for the* ZFA *universe over which variables in* ZFA *logic range when functions and predicates are given semantics on a particular model. I shall write $\mathcal{V}_{\text{ZFA}}$ even when no such semantics is in sight because the notation is useful. For example, we can now write "$x \in \mathcal{V}_{\text{ZFA}}$" for "x a set" and $F\colon \mathcal{V}_{\text{ZFA}} \to \mathcal{V}_{\text{ZFA}}$ for a function-class from* ZFA *sets to* ZFA *sets.*[13]

[11] Why not use an axiom insisting that $\mathbb{A} \cong \mathbb{N}$? It is inconsistent with FM. See §11.4.

[12] Readers not familiar with the methods of set theory may be disturbed by the use of function symbols like pow_{fin} in $(\text{AtmInf})_{26}$, since in D8.1.1 we claimed the signature contains only $\in$ and $\mathbb{A}$—no pow_{fin}. It is standard to introduce extra symbols so long as they are defined entirely in terms of existing ones—so they are macros/syntactic sugar or at least conservative extensions to the theory. We should be at least peripherally aware of the precise nature of the language we are using so that we can prove metatheoretic results by induction on its structure, e.g. equivariance in T8.1.10. Cf. R16.1.1.

[13] The point being e.g. in the second case that we actually mean a binary predicate $P_F(x, y)$ in the logic of ZFA giving the graph of F, but if the reader can obtain a model of $\mathcal{V}_{\text{ZFA}}$ (somewhere in some external universe), there really will be an associated external function $F\colon \mathcal{V}_{\text{ZFA}} \to \mathcal{V}_{\text{ZFA}}$ arising from the predicate.

The following remark sketches some pure set theory which the reader can safely ignore if he or she wishes.

Remark 8.1.7 (ZFA from the inside)**.** Using the axioms of ZFA we can construct a function-class $V\colon \textbf{Ordinals} \to \mathcal{V}_{\mathrm{ZFA}}$ such that

$$V(\alpha) = \big(\bigcup_{\gamma\in\alpha} pow(\,V(\gamma))\big) \cup \mathbb{A}.$$

This is the cumulative hierarchy inside ZFA. We can also prove the proposition

$$\forall x.\ \exists \alpha \in \textbf{Ordinals}.\ x \in V(\alpha).$$

Thus $\mathcal{V}_{\mathrm{ZFA}}$ is provably equal to the class $\bigcup V(\alpha)$ constructed using ZFA on it and in that sense we can say

> "A model $\mathcal{V}_{\mathrm{ZFA}}$ of ZFA looks like a standard cumulative hierarchy from within ZFA."

This hereby entitles the reader to do the same. $V(\alpha)$ is usually called the ***αth stage of the cumulative hierarchy*** and written V_α. ◇

We shall do a similar thing when we consider FM, see R9.1.7.

Definition 8.1.8 ($\Sigma_{\mathbb{A}}$)**.** *Call the* ***set of permutations of*** $\mathbb{A}$ *by the name* $\Sigma_{\mathbb{A}} \in \mathcal{V}_{\mathrm{ZFA}}$*. We can extend the action to all sets in a standard way by distributing* $\pi \in \Sigma_{\mathbb{A}}$ *over the* $\in$*-structure of sets. The principle of* $\in$*-induction in* ZFA *ensures this definition is sensible:*

$$\begin{aligned}
&\pi{\cdot}a = \pi(a) \in \mathbb{A} && a \in \mathbb{A} \text{ as } \pi \text{ on } \mathbb{A}\\
&\pi{\cdot}\emptyset = \emptyset &&\\
&\pi{\cdot}x = \big\{\pi{\cdot}y \;\big|\; y \in x\big\} && \text{Distribute over } \in\text{-structure otherwise.}
\end{aligned}$$

(The second clause is subsumed by the third.) We shall see in due course that we shall mostly be concerned with the case π *a transposition* $(a\ b)$*.*

Notation 8.1.9. *For* $\vec{x} = (x_1, \ldots, x_n)$ *write* $\pi{\cdot}\vec{x}$ *for* $(\pi{\cdot}x_1, \ldots, \pi{\cdot}x_n)$*.*

This is a fundamental property of ZFA:

Theorem 8.1.10 (Equivariance)**.** *For any* ϕ *in the logic of* ZFA[14] *with free variables* $\vec{x}$ *and for* $\pi \in \Sigma_{\mathbb{A}}$ *(D8.1.8),*

$$\phi(\vec{x}) \iff \phi(\pi{\cdot}\vec{x}) \tag{6}$$

is provable in ZFA*.*

[14] ... or the logic of FM, see the comment in bold font below and T9.1.6.

INFORMAL PROOF. From the construction of the permutation action D8.1.8 we can prove that $\pi{\cdot}\mathbb{A} = \mathbb{A}$ and for all u, v, $u \in v \iff \pi{\cdot}u \in \pi{\cdot}v$. Because the action is permutative and therefore injective we can prove $\pi{\cdot}u = \pi{\cdot}v \iff u = v$. This covers all the symbols of the language of ZFA that take arguments. We now argue by induction on the syntax of the language that for each ϕ we could produce a proof of $(6)_{28}$. □

This is a 'meta-result'; it is not and cannot be proved in the logic of ZFA because it works by induction on that logic's syntax. Its validity is mostly independent of the particular axioms of the set theory; it is a property of the language and **will still hold when we add an axiom and turn** ZFA **into** FM (T9.1.6).

Notation 8.1.11. *For a predicate $\lambda\vec{x}.\phi(\vec{x})\colon \mathcal{V}^n_{\mathrm{ZFA}} \to$ booleans write $\pi{\cdot}\phi$ for the predicate $\lambda\vec{x}.\phi(\pi{\cdot}\vec{x})$ constructed syntactically as $\phi[\pi{\cdot}x_i/x_i]$. In this notation equivariance is*

(7) *"$(\pi{\cdot}\phi \iff \phi)$ is provable in* ZFA *for all ϕ."*

Function-classes f are modelled by predicates ϕ_f describing the graph of f, so that

$$\phi_f(\vec{x}, y) \iff (y = f(\vec{x})).$$

The permutation action on ϕ_f translates into an action on (the graph of) the function as follows (f is unary for simplicity):

(8) $f = \lambda x, y.\phi_f(x, y) \overset{\pi}{\longmapsto} \lambda x, y.\phi_f(\pi{\cdot}x, \pi{\cdot}y) \overset{\mathrm{def}}{=} \pi{\cdot}f$

that is, $(\pi{\cdot}f)(\pi{\cdot}x) = \pi{\cdot}(f(x))$.

By equivariance (T8.1.10) $\phi_f(x, y) \iff \phi_f(\pi{\cdot}x, \pi{\cdot}y)$ and so we have

(9) $$\pi{\cdot}f = f.$$

We can combine $(9)_{29}$ with $(8)_{29}$ to obtain:

Lemma 8.1.12 (Equivariance of functions)**.** *For a function symbol F with arguments $\vec{x}$ defined in the language of* ZFA *and introduced into its theory the discussion above tells us π **commutes with** F:*

(10) $$\pi{\cdot}(F(\vec{x})) = F(\pi{\cdot}\vec{x}).$$

If F is parameterised[15] *this becomes*

$$\pi{\cdot}(F_{\vec{z}}(\vec{x})) = F_{\pi\cdot\vec{z}}(\pi{\cdot}\vec{x}). \tag{11}$$

We call this ***equivariance of functions****.*

Proof. Rewriting $(8)_{29}$ with $\vec{x}' = \pi^{-1}{\cdot}\vec{x}$ gives us

$$(\pi{\cdot}f)(\vec{x}') = \pi{\cdot}f(\pi^{-1}{\cdot}\vec{x}').$$

We now apply $(9)_{29}$, take $\sigma = \pi^{-1}$ and obtain the result $(10)_{29}$ for σ. □

This situation arises rather frequently because we often introduce defined function symbols into the theory (i.e. $\emptyset$, pow_{fin}, $\cap$, $\bigcup$, and so on).

Remark 8.1.13 (Commutativity results)**.** A plethora of ***commutativity*** or ***equivariance*** results for function-classes (and predicates) come from L8.1.12. In the last clause the predicate ϕ has free variables at most x.

$$\pi{\cdot}(\mathbf{Inl}(x)) = \mathbf{Inl}(\pi{\cdot}x) \qquad \pi{\cdot}(\mathbf{Inr}(x)) = \mathbf{Inr}(\pi{\cdot}x)$$
$$\pi{\cdot}(x, y) = (\pi{\cdot}x, \pi{\cdot}y) \qquad \pi{\cdot}\emptyset = \emptyset$$
$$\pi{\cdot}(pow_{\text{fin}}(x)) = pow_{\text{fin}}(\pi{\cdot}x) \qquad \pi{\cdot}\left\{x \in X \mid \phi(x)\right\} = \left\{x \in \pi{\cdot}X \mid \phi(x)\right\}.$$

For any binary predicate R (such as $\in$, $=$ or later $\#$, see N9.2.4),

$$x\ R\ y \iff (a\ b){\cdot}x\ R\ (a\ b){\cdot}y. \tag{12}$$

Furthermore, because $(a\ b){\cdot}(a\ b){\cdot}x = x$

$$x\ R\ (a\ b){\cdot}y \iff (a\ b){\cdot}x\ R\ y. \tag{13}$$

◇

These results are *extremely* useful for proofs by direct calculation, e.g. L13.1.1. In Isabelle/FM and Isabelle/ZFQA almost all proofs are by direct calculation and these results are at the core of the development. See R16.6.3.

Now one mark of good mathematics is when the same result reappears in different contexts. We have discussed function-classes, but permutation acts on $\mathcal{V}_{\text{ZFA}}$ and on function-sets within it. This action agrees with class-action, as we now show.

Notation 8.1.14. *For $x \in \mathcal{V}_{\text{ZFA}}$ a set we say x* ***is equivariant*** *when $\forall\pi \in \Sigma_{\mathbb{A}}.\ \pi{\cdot}x = x$ (cf. N9.2.8).*

[15] Parameters are often left implicit, so in theory there is scope for error applying this result. In practice this seems not to be a problem. For example $\lhd^*$ (D26.7.1) has quite a complicated definition but it is crystal clear (at least to me, I hope the reader will agree) that it is not parameterised.

Lemma 8.1.15. *For $f \in \mathcal{V}_{\text{ZFA}}$ a function-set, $\pi \in \Sigma_{\mathbb{A}}$ acts on it as a set (D8.1.8). Then $\pi{\cdot}f$ is itself a function-set and*

$$(\pi{\cdot}f)x = \pi^{-1}(f(\pi{\cdot}x)) \quad \textit{or equivalently} \quad f(x) = (\pi{\cdot}f)(\pi{\cdot}x). \tag{14}$$

In the case that f is equivariant (N8.1.14) this simplifies to a 'commutativity property'

$$\pi{\cdot}(f(x)) = f(\pi{\cdot}x)$$

—just as for function-classes.

Proof. I could write both a concrete and an abstract proof. I favour the first because it shows what really happens to the sets and the second to show the higher truth of the result. Both points of view are important. I hope the reader will excuse me if I include both. In future I shall not.

1• *One way of doing it.* The standard implementation of f is its graph

$$f = \left\{(x, y) \;\middle|\; y = f(x)\right\}.$$

The π action on sets distributes over $\in$ and maps to another graph

$$\pi{\cdot}f = \left\{(\pi{\cdot}x, \pi{\cdot}y) \;\middle|\; y = f(x)\right\}.$$

When we unpack this (writing $x' = \pi^{-1}.x$) we obtain

$$(\pi{\cdot}f)x' = \pi{\cdot}(f(\pi^{-1}.x')).$$

Here it is not necessarily the case that $\pi{\cdot}f = f$ any more than $\pi{\cdot}x = x$ for sets in general but if f is equivariant (N8.1.14) then

$$\pi{\cdot}(f(x)) = f(\pi{\cdot}x).$$

2• *Another way of doing it.* The discussion above depended on the implementation of f as a graph and is in that sense not a general result. Another argument is more attractive and more general. We consider only unary functions for simplicity. Let ϕ be a predicate in the language of ZFA

$$\phi(f, x, y) = \text{“}f \text{ is a unary function”} \wedge f(x) = y.$$

Suppose for some f, x, y, ϕ is true. Equivariance tells us that

$$\phi(\pi{\cdot}f, \pi{\cdot}x, \pi{\cdot}y) = \text{“}\pi{\cdot}f \text{ is a function”} \wedge (\pi{\cdot}f)(\pi{\cdot}x) = \pi{\cdot}y.$$

So permutation acts[16] on a set-function f such that

$$(\pi{\cdot}f)x = \pi(f(\pi^{-1}x)).$$

[16]Independently of the particular implementation of functions in the set-theory.

If it happens that f is equivariant as a set this becomes

$$\pi{\cdot}(f(x)) = f(\pi{\cdot}x).$$

□

8.2. Semantics of syntax in ZFA.

Remark 8.2.1 (Visualise permutation)**.** By R8.1.7 we can picture sets as trees whose leaves are labelled with $x \in \mathbb{A} \cup \{\emptyset\}$. $\Sigma_{\mathbb{A}}$ acts on sets by acting on the labels $\mathbb{A} \cup \{\emptyset\}$ (where $\pi{\cdot}\emptyset = \emptyset$). ◇

Recall the set **lam1** constructed somewhat informally in $(3)_{15}$. It gave semantics to the datatype `lam1` (D4.2) representing λ-terms. We did not have $\mathbb{A}$ then. It is convenient to reimplement using it.

Definition 8.2.2 (`lam3`)**.** *We declare in our informal language:*

```
datatype lam3 = Var3 of Atm
              | App3 of lam3 * lam3
              | Lam3 of Atm * lam3;
```

Definition 8.2.3 (**lam3**)**.** *A semantics for this datatype, the* ***set of syntax for the datatype*** `lam3`*, is the set* ***lam3*** *built up using the rules*

$$\frac{x \in \mathbb{A}}{\boldsymbol{Var3}(x) \in \boldsymbol{lam3}} \quad \frac{t1, t2 \in \boldsymbol{lam3}}{\boldsymbol{App3}(t1, t2) \in \boldsymbol{lam3}} \quad \frac{x \in \mathbb{A}, t \in \boldsymbol{lam3}}{\boldsymbol{Lam3}(x, t) \in \boldsymbol{lam3}}.$$

Notation 8.2.4 (Semantic terms)**.**

1. *We call a term* `t:X` *a* ***syntactic term*** *or* ***term*** *and* `X` *a* ***type****.*
2. *We call a set in its semantics $t \in X$ a* ***term*** *or* ***semantic term*** *and X a* ***set of syntax****.*

So ***lam3*** *is a set of syntax for the λ-calculus not up to α-equivalence.*

We can take **Var3**, **App3** and **Lam3** to be **Inl**$(-)$, **Inr**(**Inl**$(-)$) and **Inr**(**Inr**$(-)$) (cf. R4.3).

Recall from R4.4 that a term `t` is assumed to be an abstract syntax tree. Using R8.2.1 to imagine sets as trees, **Var3** etc. bolt their arguments together with unique $\in$-structure that emulate the syntactic labels on the nodes of `t`. Variable names, i.e. atoms, occur on the leaves of both set and abstract syntax trees. The permutation action on semantic terms relabels these leaves, just as it does in the abstract syntax. The commutativity results of R8.1.13 prove this rigorously.

However, the permutation action acts on all sets, whether they represent abstract syntax or not. Thus we have a general theory of 'renaming variable names in sets' independent of any particular syntax and crucially, not defined by induction

on the terms of a particular datatype. Any theorems we prove about permutation immediately become theorems about abstract syntax—whatever the datatype.

Now consider α-equivalence on the set of syntax **lam3**. Here are three notions of variable-renaming for elements t of **lam3**, where $a, a' \in \mathbb{A}$:

$[a'/a]t$ capture-avoiding substitution of a' for all free occurrences of a in t.

$\{a'/a\}t$ textual substitution of a' for all free occurrences of a in t.

$(a'\ a)\cdot t$ interchange of *all* occurrences (be they free, bound, or binding) of a and a' in t.

The third version may be unfamiliar but it is more basic than the other two for two reasons:

1. We need not know which constructors of **lam3** are binders to define it.
2. Nevertheless we can use it to define α-conversion, as the following result shows (cf. [**22**, p 36], which uses $\{a'/a\}(-)$ in place of $(a'\ a)\cdot(-)$ for the same purpose).

Theorem 8.2.5. *Recall that* α-conversion, $=_\alpha$, *is usually defined as the least congruence on* Λ *that identifies* $\mathbf{Lam3}(a,t)$ *with* $\mathbf{Lam3}(a',[a'/a]t)$. *Then* $=_\alpha$ *coincides with the binary relation* $\sim$ *on* **lam3** *inductively generated by the following rules:*

$$\mathbf{Var3}(a) \sim \mathbf{Var3}(a) \qquad \frac{t_1 \sim t_1' \quad t_2 \sim t_2'}{\mathbf{App3}(t_1,t_2) \sim \mathbf{App3}(t_1',t_2')}$$

$$\frac{(c\ a)\cdot t \sim (c\ b)\cdot t'}{\mathbf{Lam3}(a,t) \sim \mathbf{Lam3}(b,t')} \quad \begin{array}{l}\text{if } c \text{ does not}\\ \text{occur in } t, t'.\end{array} \tag{15}$$

Proof. It is not hard to see that $(b\ a)\cdot(-)$ preserves $=_\alpha$ and hence that $=_\alpha$ is closed under the axioms and rules defining $\sim$. Therefore $\sim$ is contained in $=_\alpha$.

We prove the converse by showing $\sim$ is a congruence relating $\mathbf{Lam3}(a,t)$ to $\mathbf{Lam3}(b,[b/a]t')$. This follows from two facts: if c does not occur in t then $(c\ a)\cdot t \sim [c/a]t$, and the permutation action $(a_1\ a_2)\cdot(-)$ preserves $\sim$ (by an appropriate case of R8.1.13). □

We see $\sim$ redefined in a different context (using M) in D26.3.1.

We can define 'permute variable names' on a particular datatype or the set-universe using $\in$-induction. In T8.2.5 we defined α-equivalence on a particular datatype. Can we translate this too on the set-universe?

Remark 8.2.6 (Synthetic)**.** We call a translation of a syntactic concept to the semantic universe '***synthetic***'. So permutation is a synthetic notion of renaming of variables. In §9 below we shall define synthetic notions of "free variable symbols

of a set", "α-equivalence of two sets", "variable-symbol abstraction in a set", "pick a fresh variable name", and so on.

The advantage of a synthetic version of a notion is that we can prove results about it which will hold for all its concrete realisations on particular datatypes. This is nice in theory, useful in practice, and wonderful in automation. ◇

9. Elementary FM set theory

9.1. Axioms of FM. Till now our set of permutations has been $\Sigma_{\mathbb{A}}$ (D8.1.8). We now concentrate on a restricted permutation set $F_{\mathbb{A}}$:

Definition 9.1.1 ($F_{\mathbb{A}}$)**.** *Write $F_{\mathbb{A}}$ for the subgroup of $\Sigma_{\mathbb{A}}$ (D8.1.8) generated by the* ***transpositions*** *$(a\ b)$, for $a, b \in \mathbb{A}$. Of course $F_{\mathbb{A}}$ inherits the $\Sigma_{\mathbb{A}}$ action on the universe.*

Definition 9.1.2 (Φ)**.** *Consider $\Phi(U, x)$ as defined in Fig.3$_{35}$. For $U \subseteq \mathbb{A}$ we read $\Phi(U, x)$ as "U* ***supports*** *x".*

We can read the meaning of this definition as

> "If $U \subseteq \mathbb{A}$ and x is any set then "U supports x" when for all permutations $\pi \in F_{\mathbb{A}}$, if π fixes U pointwise then π fixes x."

Remark 9.1.3 (Finite Support)**.** So (Fresh)$_{35}$ asserts that sets have at least one finite set supporting them. The slogan is:

> ***All sets are of finite support.***

We call this the ***finite support property of*** FM. ◇

ZFA does not have the finite support property. A subset of $\mathbb{A}$ which is neither finite nor cofinite does not have finite support. We do not discuss this further, but the reader is referred to L9.4.3. There we show that such subsets do not exist in FM. They cannot, (Fresh)$_{35}$ outlaws them.

An obvious question is why we cut down to $F_{\mathbb{A}}$ from $\Sigma_{\mathbb{A}}$ when we define support. $F_{\mathbb{A}}$ is much easier to work with and it makes no difference in the end, see §11.3.

Remark 9.1.4 (Reformulate Φ)**.** Since $F_{\mathbb{A}}$ is generated by transpositions we can reformulate Φ as

$$\Phi(U, x) \iff \forall a, b \in \mathbb{A} \setminus U.\ (a\ b)\cdot x = x.$$

◇

Note that $\Phi(\mathbb{A}, x)$. Does this mean that all sets are trivially finitely supported? No, because $\mathbb{A}$ is infinite by (AtmInf)$_{26}$. But we now add precisely this as an axiom (Fresh)$_{35}$. This turns ZFA into FM.

(Fresh) $$\forall x.\ \exists U \in pow_{\mathrm{fin}}(\mathbb{A}).\ \Phi(U,x)$$

where

(16) $$\Phi(U,x) \stackrel{\text{def}}{=} \forall \pi \in F_{\mathbb{A}}.\ \Big(\forall u \in U.\ \pi{\cdot}u = u\Big) \implies \pi{\cdot}x = x.$$

An alternative form of Φ (see R9.1.4) is

$$\Phi(U,x) \iff \forall a,b \in \mathbb{A} \setminus U.\ (a\ b){\cdot}x = x.$$

FIGURE 3. D9.1.5 - Axioms of FM

Definition 9.1.5 (FM set theory)**.** *We described* ZFA *in §8.1 and axiomatised it in Fig.2*$_{26}$*.* FM *is the extension of* ZFA *by one axiom* (Fresh)$_{35}$ *given in Fig.3*$_{35}$*.*

Theorem 9.1.6 (Equivariance of FM)**.** *The language of* FM *is identical to that of* ZFA *and all equivariance results (e.g. T8.1.10 and L8.1.12) remain valid.*

Remark 9.1.7 (FM from inside)**.** As we did for ZFA in R8.1.7 we can build a function-class V : **Ordinals** $\to \mathcal{V}_{\mathrm{FM}}$ taking α to the αth stage of the cumulative hierarchy V_α, this time in $\mathcal{V}_{\mathrm{FM}}$. We can also prove in FM that $\mathcal{V}_{\mathrm{FM}} = \bigcup V_\alpha$. This entitles the reader to imagine a model $\mathcal{V}_{\mathrm{FM}}$ of FM to be a standard cumulative hierarchy, at least while we reason inside FM. As in R8.1.7 I shall use the notation $\mathcal{V}_{\mathrm{FM}}$ even when no such model is in sight, for notational convenience.

In §11.5 we shall do something more subtle. We shall take a particular $\mathcal{V}_{\mathrm{ZFA}}$ and build a cumulative hierarchy *inside it*, using a slightly different powerset function than *pow* (see L11.5.11 and surrounding text). This hierarchy will *not* be the whole universe $\mathcal{V}_{\mathrm{ZFA}}$, nor is it a model of ZFA—but it *is* a model of FM. That will prove relative consistency of FM wrt ZFA. ◇

9.2. Support and #. R9.1.3 states that all sets in $\mathcal{V}_{\mathrm{FM}}$ have finite support. In fact they have a *unique smallest* supporting set:

Theorem 9.2.1 (Support)**.** *For $x \in \mathcal{V}_{\mathrm{FM}}$ there is a unique smallest (finite) set $\boldsymbol{Supp}(x) \in pow_{\mathrm{fin}}\mathbb{A}$ such that $\Phi(\boldsymbol{Supp}(x), x)$ (see (16)*$_{35}$ *in Fig.3*$_{35}$*), which we call the **support of** x.*

PROOF. By (Fresh)$_{35}$ some finite $X \subseteq \mathbb{A}$ such that $\Phi(X,x)$ exists. By L9.2.2 below, $\mathbf{Supp}(x)$ is the smallest set below X such that $\Phi(\mathbf{Supp}(x), x)$. □

Lemma 9.2.2 (Infimum property of finite supporting sets)**.**

$$\forall U, V \in pow_{\mathrm{fin}}(\mathbb{A}).\ \Phi(U,x) \wedge \Phi(V,x) \implies \Phi(U \cap V, x).$$

Proof. In this proof x is understood so we write $\Phi(X,x)$ as $\Phi(X)$. We need only consider transpositions $(a\ b)$ because they generate $F_{\mathbb{A}}$. We show for $a, b \notin U \cap V$ that $(a\ b)\cdot x = x$. There are various cases to consider.

1. *Suppose* $a \notin U \wedge b \notin U$. By assumption, $(a\ b)\cdot x = x$.
2. *Suppose* $a \notin V \wedge b \notin V$. By assumption, $(a\ b)\cdot x = x$.
3. *Suppose* $a \notin U \wedge b \notin V$. Since U and V are finite and by $(\mathrm{AtmInf})_{26}$ $\mathbb{A}$ is not, we can pick some $c \in \mathbb{A}$ such that $c \notin U$ and $c \notin V$ and $a \neq c \neq b$. By previous cases we know that $(c\ a)\cdot x = x$ and $(c\ b)\cdot x = x$, and so that
$$(a\ b)\cdot x = (a\ b)\cdot(c\ b)\cdot x \overset{(17)_{36}}{=} (c\ b)\cdot(a\ c)\cdot x = x,$$
as required.
4. *Suppose* $a \notin V \wedge b \notin U$. Similar to the previous case.

□

The proof above uses the following elementary technical lemma about transposition actions:

Lemma 9.2.3 (Technical Lemma)**.** *For all* $a, b, c, d \in \mathbb{A}$*,*

$$(a\ b)\cdot(c\ d) = (c\ d)\cdot((c\ d)\cdot a\ (c\ d)\cdot b). \tag{17}$$

Similarly,

$$(a\ b)\cdot(c\ d) = ((a\ b)\cdot c\ (a\ b)\cdot d)\cdot(c\ d). \tag{18}$$

Proof. Omitted. □

This is the terribly important Isabelle/ZFQA result `Perm_commute`, see §16.6.

Notation 9.2.4 (Support and Apartness)**.** *By T9.2.1 we can construct a function-class*

$$\begin{aligned} \boldsymbol{Supp}\colon \mathcal{V}_{\mathrm{FM}} &\longrightarrow pow_{\mathrm{fin}}(\mathbb{A}) \\ x &\longmapsto \boldsymbol{Supp}(x). \end{aligned}$$

We read $\boldsymbol{Supp}(x)$ *as "**the support of** x". In addition we write*

$$a\#x \overset{\mathrm{def}}{=} a \notin \boldsymbol{Supp}(x)$$

*and say "a **is apart from** x". These judgements will collectively be called "**apartness judgements**".*[17]

[17] cf. D23.1.6 where we build a syntactic approximation to this relation. There we shall sometimes find it convenient to write apartness judgements $x\#a$ instead of $a\#x$.

The following is a trivial corollary of $(\text{AtmInf})_{26}$ ($\mathbb{A}$ infinite) and the construction of **Supp**:

Corollary 9.2.5. *For all sets x there is a cofinite $A \subseteq \mathbb{A}$ such that for all $a \in A$, $a \notin \boldsymbol{Supp}(x)$.*

Having introduced these function/predicate-classes we can immediately deduce:

Lemma 9.2.6. *By L8.1.12 we extend R8.1.13 with*

$$(a\ b)\cdot\boldsymbol{Supp}(X) = \boldsymbol{Supp}((a\ b)\cdot X) \quad \textit{and} \quad n\#x \iff (a\ b)\cdot n\#(a\ b)\cdot x.$$

Lemma 9.2.7 (Properties of #)**.** 1. *If $a, b\#x$ then $(a\ b)\cdot x = x$.*

2. *If $a\#x$ and $\neg(b\#x)$ then $(a\ b)\cdot x \neq x$.*
3. *If $(a\ b)\cdot x \neq x$ then at least one of $\neg(a\#x)$ or $\neg(b\#x)$.*

Proof. 1. $\mathbf{Supp}(x)$ supports x; $\Phi(\mathbf{Supp}(x), x)$. When we unpack the definition of Φ in D9.1.2 we obtain the desired result.

2. By L9.2.6 above if $b \in \mathbf{Supp}(x)$ and $a \notin \mathbf{Supp}(x)$ then certainly $\mathbf{Supp}((a\ b)\cdot x) \neq \mathbf{Supp}(x)$—and so $(a\ b)\cdot x \neq x$.
3. Contrapositive of Item 1.

□

Notation 9.2.8 (Equivariant sets)**.** *We say X **is an equivariant set** when $\boldsymbol{Supp}(X) = \emptyset$, or put another way when $a\#X$ for all $a \in \mathbb{A}$.*

This notion of equivariance coincides precisely with that of N8.1.14:

Corollary 9.2.9. *A set x is equivariant if and only if for all $\pi \in F_{\mathbb{A}}$, $\pi\cdot x = x$. In particular this holds if and only if for all $a, b \in \mathbb{A}$, $(a\ b)\cdot x = x$.*

Proof. We unfold the definition of $\Phi(\emptyset, x)$ in D9.1.2 and use the fact that $F_{\mathbb{A}}$ is generated by transpositions. □

Remark 9.2.10. Supp is a synthetic (R8.2.6) version of "free variable symbols of" and $a\#x$ is a synthetic version of "not in the free variables of"; `a` $\notin$ `FV(x)`. We work through the details in $(19)_{39}$ and state it as a general result in L10.7.7. ◇

9.3. Calculating Supp for particular sets. It is useful to calculate **Supp** for specific sets. Recall the reformulation of Φ in R9.1.4 as

$$\forall a, b \notin U.\ (a\ b)\cdot x = x.$$

(We implicitly type $a, b, c, n, \ldots$ as atoms in $\mathbb{A}$.)

Recall also from §9.2 that $\mathbf{Supp}(x)$ is the *unique smallest* set such that $\Phi(\mathbf{Supp}(x), x)$. So the following proof-methods suggest themselves:

Remark 9.3.1 (Two proof methods)**.**

- From minimality of **Supp**, if $\Phi(V, x)$ for $V \in pow_{\text{fin}}(\mathbb{A})$ then $\mathbf{Supp}(x) \subseteq (V)$. To show therefore that $a \notin \mathbf{Supp}(x)$ we need only exhibit $V \in pow_{\text{fin}}(\mathbb{A})$ such that $a \notin V$ and for all $b, b' \notin V$, $(b\ b')\cdot x = x$.
- From Item 3 of L9.2.7 if $(a\ b)\cdot x \neq x$ then either a or b is in the support of x. Therefore, to show $a \in \mathbf{Supp}(x)$ it suffices to exhibit some $b \notin \mathbf{Supp}(x)$ such that $(a\ b)\cdot x \neq x$.

$\diamond$

We can now calculate $\mathbf{Supp}(U)$ for $U \in pow_{\text{fin}}(\mathbb{A})$.

Lemma 9.3.2.

$$U \in pow_{\text{fin}}(\mathbb{A}) \implies \boldsymbol{Supp}(U) = U.$$

Proof. Clearly $\Phi(U, U)$ so by the first proof-method of R9.3.1 $\mathbf{Supp}(U) \subseteq U$. Now for the reverse inclusion. Suppose $a \in U, b \notin U$. Then $b \notin \mathbf{Supp}(U) \subseteq U$ and clearly $(a\ b)\cdot U \neq U$. Hence by the second proof-method $U \subseteq \mathbf{Supp}(U)$. $\square$

As an immediate corollary we have

Corollary 9.3.3.

$$\boldsymbol{Supp}(\boldsymbol{Supp}(x)) = \boldsymbol{Supp}(x) \quad \textit{and} \quad \boldsymbol{Supp}(\{a\}) = \boldsymbol{Supp}(a).$$

Now suppose $f\colon \mathcal{V}_{\text{FM}}^n \to \mathcal{V}_{\text{FM}}$ is a function-class defined in the language of FM. Then equivariance (L8.1.12) tells us that

$$(a\ b)\cdot f(x_1, \ldots, x_n) = f((a\ b)\cdot x_1, \ldots, (a\ b)\cdot x_n).$$

Suppose $\Phi(V, x_i)$ for all x_i. The above commutativity result then implies that $\Phi(V, f(\vec{x}))$. By the first proof-method we have

Lemma 9.3.4 (No increase of support)**.**

$$\boldsymbol{Supp}(f(x_1, \ldots, x_n)) \subseteq \bigcup_{x_i} \boldsymbol{Supp}(x_i).$$

In the case $n = 1$ this simplifies to

$$\mathbf{Supp}(f(x)) \subseteq \mathbf{Supp}(x).$$

We can read this as "***support cannot increase***". *Beware!* If g is a function-*set* it need not be equivariant. In this case the appropriate result is $\mathbf{Supp}(g(x)) \subseteq \mathbf{Supp}(g) \cup \mathbf{Supp}(x)$. We can see this holds by taking f to be application $f = \lambda g, x.g(x)$ in L9.3.4.

The relation between support and an *injective* function-class is of particular interest because semantics of term-formers for datatypes are of this type.

Lemma 9.3.5 (Conservation of Support)**.** *Let f be a function-class as in L9.3.4 which is also injective. Then*

$$\boldsymbol{Supp}(f(x_1,\ldots,x_n)) = \bigcup_{x_i} \boldsymbol{Supp}(x_i).$$

Proof. We consider $n = 1$ only, the general case is similar. Write x for x_1. By L9.3.4 $\mathbf{Supp}(f(x)) \subseteq \mathbf{Supp}(x)$.

For $\mathbf{Supp}(x) \subseteq \mathbf{Supp}(f(x))$ we use the second proof-method of R9.3.1. If $\mathbf{Supp}(x) = \emptyset$ then $\mathbf{Supp}(x) \subseteq \mathbf{Supp}(f(x))$. So suppose we have $a \in \mathbf{Supp}(x)$. Pick some $b \notin \mathbf{Supp}(x)$. Clearly

$$\mathbf{Supp}(x) \neq (b\ a)\cdot\mathbf{Supp}(x) \overset{L9.2.6}{=} \mathbf{Supp}((b\ a)\cdot x)$$

and so

$$x \neq (b\ a)\cdot x \overset{f \text{ injective}}{\Longrightarrow} f(x) \neq (b\ a)\cdot f(x).$$

Recall that $b \notin \mathbf{Supp}(x)$ and $\mathbf{Supp}(f(x)) \subseteq \mathbf{Supp}(x)$. So $b \notin \mathbf{Supp}(f(x))$. By the second proof-method above we obtain $a \in \mathbf{Supp}(f(x))$. We have proved

$$a \in \mathbf{Supp}(x) \implies a \in \mathbf{Supp}(f(x)),$$

and this gives us the reverse inclusion as required. □

If f is not injective anything can happen.[18] E.g. take $f = \lambda x.\emptyset$ so $\mathbf{Supp}(f(x)) = \emptyset$ by L9.3.2.

Lemma 9.3.6 (**Supp** of atoms)**.** *For $a \in \mathbb{A}$, $\boldsymbol{Supp}(a) = \{a\}$. In particular for $a, b \in \mathbb{A}$, $b\#a \iff b \neq a$.*

Proof. For the first part combine L9.3.5 and L9.3.2 with $f = \lambda x.\{x\}$. For the second, unfold the definition of # (N9.2.4). □

At the end of §9.2 I claimed that **Supp** provided a synthetic version of the 'free variables of' function `FV` on syntax. We have only built the semantics **lam3** for `lam3` so let us try it out. Using the lemmas above we can prove by induction on the structure of semantic terms in **lam3** that the support of $t \in$ **lam3** is *precisely* the set of atoms appearing in it.

$$\begin{gathered}\mathbf{Supp}(\mathbf{Var3}(a)) = \{a\} \quad \mathbf{Supp}(\mathbf{App3}(t_1,t_2)) = \mathbf{Supp}(t_1) \cup \mathbf{Supp}(t_2) \\ \mathbf{Supp}(\mathbf{Lam3}(a,t)) = \{a\} \cup \mathbf{Supp}(t)\end{gathered} \tag{19}$$

The only problem seems to be with the last fact. Surely it should read $\mathbf{Supp}(t) \setminus \{a\}$? No, this is correct. In `Lam3(a,t)`, `a` *is free* in a syntactic sense; `Lam3(a,Var3(a))` and `Lam3(b,Var3(b))` are *not equal*. We may define equivalence

[18] Within the limits set by L9.3.4.

relations that equate them and functions that satisfy `f(Lam3(a,t))=f(t)-{a}`, and we may write `f` as `FV`, but that is a different matter.

Remark 9.3.7 (Cynicism)**.** The cynical reader may be inclined to ask "so what's the use, seeing as the free-variables function is technically correct but not the one we're interested in?". The answer is that FM set theory will continue to deliver the goods (culminating in D10.7.2 and the subsequent development), but we have to advance one step at a time. ◇

Supp has many more nice properties. Some of them are discussed in §13.

9.4. The И-quantifier. We now show that FM has a notion of "$\phi(\vec{z}, a)$ where a is fresh for $\vec{z}$" which we have already seen in our construction of α-equivalence $\sim$ on **lam3** in $(15)_{33}$.

Notation 9.4.1 (Cofiniteness)**.** *$X \subseteq Y$ is a **cofinite subset** of Y when $Y \setminus X$ is finite. For a set X we define $pow_{\mathrm{cof}}(X)$ to be the set of cofinite subsets of X.*

This is the 'freshness' quantifier promised at the beginning of this subsection. We shall call it ***the И quantifier***, read "the new quantifier".

Definition 9.4.2 (The И quantifier)**.** *For ϕ a predicate in the logic of* FM*, we define*

$$\text{И}a.\ \phi \stackrel{\text{def}}{=} \{a \in \mathbb{A} \mid \phi\} \in pow_{\mathrm{cof}}(\mathbb{A}). \tag{20}$$

We can read this definition as

> "[New atom a, $\phi(a)$] holds precisely when the set of $a \in \mathbb{A}$ such that $\phi(a)$ holds is cofinite."

The reader is warned that he or she will see $\text{И}a.\ \phi(a)$ written not only as "for new a, $\phi(a)$" but also as "for fresh a, $\phi(a)$". We use the two terminologies synonymously.

This new quantifier is of great significance and has very nice logical properties. The following technical lemma is the key to developing them. It generalises L9.3.2.

Lemma 9.4.3 (Support of subsets of $\mathbb{A}$)**.** *If $U \subseteq \mathbb{A}$ then either*

$$U = \boldsymbol{Supp}(U) \quad \textit{or} \quad U = \mathbb{A} \setminus \boldsymbol{Supp}(U).$$

Note that in view of the fact that $\boldsymbol{Supp}(U)$ is always finite (N9.2.4), the two cases above occur when U is finite and cofinite respectively.

Proof. By N9.2.4 we know $U \subseteq \mathbb{A}$ has some $\mathbf{Supp}(U)$ such that $\Phi(\mathbf{Supp}(U), U)$ (D9.1.2). Suppose

$$a \in \mathbb{A} \setminus \mathbf{Supp}(U) \quad \text{and} \quad b \in \mathbf{Supp}(U).$$

Since $\mathbf{Supp}((a\ b)\cdot U) \overset{L9.2.6}{=} (a\ b)\cdot\mathbf{Supp}(U) \neq \mathbf{Supp}(U)$ we know $(a\ b)\cdot U \neq U$. So

$$\text{either}\quad a \in U \wedge b \notin U \quad \text{or} \quad a \notin U \wedge b \in U.$$

We treat only the first alternative, the proof for the second is similar. So $a \in \mathbb{A} \setminus \mathbf{Supp}(U)$, $a \in U$ and $b \notin U$. Since $a \in U$ we can use equivariance of $\in$ to deduce

$$\forall a' \in \mathbb{A} \setminus \mathbf{Supp}(U).\ (a'\ a)\cdot a \in (a'\ a)\cdot U.$$

Now $a, a' \notin \mathbf{Supp}(U)$ so $(a'\ a)\cdot U = U$ (L9.2.7). It follows that this equation is

$$\forall a' \in \mathbb{A} \setminus \mathbf{Supp}(U).\ a' \in U \quad \text{i.e.} \quad \mathbb{A} \setminus \mathbf{Supp}(U) \subseteq U.$$

We can rewrite this to $U \subseteq \mathbf{Supp}(U)$. Since $\Phi(U, U)$ and $\mathbf{Supp}(U)$ is the smallest set such that $\Phi(U, \mathbf{Supp}(U))$ it follows that $U = \mathbf{Supp}(U)$.

The proof proceeds from the other alternative in a similar manner and shows that $\mathbf{Supp}(U) = \mathbb{A} \setminus U$. □

Corollary 9.4.4 (Subsets of $\mathbb{A}$)**.** *All subsets of $\mathbb{A}$ are either finite or cofinite. If $U \subseteq \mathbb{A}$ is finite then $\boldsymbol{Supp}(U) = U$. If $U \subseteq \mathbb{A}$ is cofinite then $\boldsymbol{Supp}(U) = \mathbb{A}\setminus U$.*

Proof. By L9.4.3, if $U \subseteq \mathbb{A}$ then $U = \mathbf{Supp}(U)$ or $\mathbb{A} \setminus U = \mathbf{Supp}(U)$. But $\mathbf{Supp}(U)$ is always finite (N9.2.4) so the result follows. □

Corollary 9.4.5 (И well-behaved)**.** *By C9.4.4 $pow_{\mathrm{cof}}\mathbb{A}$ is an ultrafilter and so $\mathsf{И}a.\ \phi$ commutes with conjunction, disjunction, implication* and *negation, as well as preserving top $\top$ and bottom $\bot$.*

Thus $\mathsf{И}a.\ \top = \top$, $\mathsf{И}a.\ (\neg\phi) \iff \neg\mathsf{И}a.\ \phi$, and so on.

Theorem 9.4.6. *For any formula ϕ and list of distinct variables $\vec{x}$ in the language of* FM*, consider the following formulae.*

$$\forall a \in \mathbb{A}.\ a\#\vec{x} \implies \phi \tag{21}$$

$$\mathsf{И}a.\ \phi \tag{22}$$

$$\exists a \in \mathbb{A}.\ a\#\vec{x} \wedge \phi \tag{23}$$

where $a\#\vec{x}$ is shorthand for $\bigwedge_{x\in\vec{x}} a\#x$. Then in FM*,*

$$(21)_{41} \implies (22)_{41} \implies (23)_{41},$$

and if the free variables of ϕ are contained in $\{\vec{x}, a\}$, then $(23)_{41} \implies (21)_{41}$ so the three formulae are provably equivalent in FM*.*

PROOF. $(21)_{41} \implies (22)_{41}$ is clear, since ϕ is finite, so $\vec{x}$ is finite, so

$$\bigcup \{\mathbf{Supp}(x) \mid x \in \vec{x}\} \quad \text{is finite.}$$

It follows given $(21)_{41}$ that $\{a \in \mathbb{A} \mid \phi\}$ is in $pow_{\mathrm{cof}}(\mathbb{A})$ as required.

$(22)_{41} \implies (23)_{41}$ is even easier. $\mathbb{A}$ is infinite by $(\mathrm{AtmInf})_{26}$. By assumption $\{a \in \mathbb{A} \mid \phi\}$ is in $pow_{\mathrm{cof}}(\mathbb{A})$ and by the finite support property so is $\{a \in \mathbb{A} \mid a\#\vec{x}\}$. So $(23)_{41}$ follows.

$(23)_{41} \implies (21)_{41}$ follows by equivariance (T9.1.6). Suppose

$$\exists a \in \mathbb{A}.\ a\#\vec{x} \wedge \phi(a, \vec{x}).$$

Let a be the a in question.[19] Then equivariance dictates that

$$\forall a' \in \mathbb{A}.\ a'\#\vec{x} \implies a'\#\vec{x} \wedge \phi(a', \vec{x}),$$

which simplifies to $(21)_{41}$ as required. □

Remark 9.4.7 (... furthermore)**.** Using the notation of T9.4.6, we can add dummy variable dependencies $\vec{z}$, so for $\phi(\vec{x}, a)$ a predicate with free variables included in $\vec{x}, a$,

$$\forall a \in \mathbb{A}.\ a\#\vec{x} \wedge a\#\vec{z} \implies \phi \tag{24}$$

$$И a.\ \phi \tag{25}$$

$$\exists a \in \mathbb{A}.\ a\#\vec{x} \wedge a\#\vec{z} \wedge \phi \tag{26}$$

◇

The following result is central. It wraps up the characteristic ∀-∃ duality of И in a convenient package.

Lemma 9.4.8 (Pick a new a)**.** *In particular, when faced by* $И a.\ \phi(\vec{x}, a)$*, if we fix* $\vec{x}$ *we can pick a new* a' *apart from* $\vec{x}$ *and also any other finite collection* $\vec{z}$ *and we know*

$$(И a.\ \phi(\vec{x}, a)) \iff \phi(\vec{x}, a').$$

Note that in the rest of this document I may not bother to give a *and* a' *distinct names.*

PROOF. Suppose $И a.\ \phi(\vec{x}, a)$. We can universally expand И using $(24)_{42}$. By $(\mathrm{AtmInf})_{26}$ $\mathbb{A}$ is infinite and by T9.2.1 the combined supports of $\vec{x}, \vec{z}$ are finite.

[19] We can't add a constant a and appropriate axiom to the signature because that would destroy equivariance. So we add a variable symbol and appropriate axiom instead. Actually, this point is rather important.

Recall from N9.2.4 that $a\#x \iff a \notin \mathbf{Supp}(x)$, so we can choose some particular $a'\#\vec{x},\vec{z}$ and state

$$\phi(\vec{x},a')$$

as required.

Conversely, if we know $\phi(\vec{x},a')$ and $a'\#\vec{x}$ (and possibly some other $\vec{z}$), we may use the existential expansions of И, $(26)_{42}$ or $(23)_{41}$, to deduce $И a.\ \phi(\vec{x},a)$. □

Corollary 9.4.9. *If ϕ does not depend on a then*

$$\Big(И a.\ \phi\Big) \iff \phi.$$

Proof. A corollary of L9.4.8. □

Remark 9.4.10. Thanks to T9.4.6, R9.4.7 and L9.4.8 we can read И as

- "For *all new* a, ... " ,
- "For *all but finitely many* a, ... ", or
- "For *some new* a, ... "

as we please, where 'new' means "is apart from (N9.2.4) the free variables of the formula in the scope of the quantifier". Note that in due course the terminology 'fresh' will also appear in a slightly different context, and can be considered synonymous with 'new'. ◇

Corollary 9.4.11 (И better-behaved)**.** *Continuing C9.4.5, it follows from the discussion above that* И *can extend its scope arbitrarily over conjunction* and *disjunction, etc (so long as no variables are accidentally captured, of course). E.g.*

$$\psi(\vec{x}) \wedge И a.\ \phi(\vec{x},a) \iff И a.\ \big(\psi(\vec{x}) \wedge \phi(\vec{x},a)\big)$$
$$\psi(\vec{x}) \vee И a.\ \phi(\vec{x},a) \iff И a.\ \big(\psi(\vec{x}) \vee \phi(\vec{x},a)\big).$$

Here I have (used R9.4.7 and) bulked out the variable dependencies of ϕ and ψ to some common superset $\vec{x}$.

Remark 9.4.12 (Binding under И)**.** Consider a predicate

$$И a.\ \forall x.\ \phi(a,x).$$

Note that inside ϕ we cannot assume $a\#x$ because x is *not* free in the scope of the И-quantifier. Similarly for $\exists$. T9.4.6 does give us formulae like

$$\forall x.\ И a.\ \phi(a,x) \iff И a.\ \forall x.\ a\#x \to \phi(a,x)$$
$$\exists x.\ И a.\ \phi(a,x) \iff И a.\ \exists x.\ a\#x \wedge \phi(a,x).$$

◇

Recall the terminology 'synthetic' introduced in R8.2.6. The И-quantifier is a synthetic "pick a fresh variable name". Unlike **Supp** and permutation this concept does not correspond to any universally agreed entity in the world of syntax and programming languages. Its effects are usually obtained through something like a choice function (cf. §11.4) for $pow_{\mathrm{cof}}(\mathbb{A})$:

(NOT IN FM) $$f\colon pow_{\mathrm{cof}}(\mathbb{A}) \to \mathbb{A} \quad \text{such that} \quad \forall U.\, f(U) \in U.$$

Note that f is not a function-set in FM, that is the point! It can live quite happily in ZFA (§8) acting perhaps on the model of FM in ZFA (**HFS**, see §11.5).

We can use И to construct a synthetic version of α-equivalence. For sets x, y let (x, y) denote the pair-set. We observe in passing that by L8.1.12 we know $(a\ b)\cdot(x, y) = ((a\ b)\cdot x, (a\ b)\cdot y)$.

Definition 9.4.13 ($\sim$)**.** *Let $\sim$ be the following relation-class on $\mathbb{A} \times \mathrm{FM}$:*

$$(a, x) \sim (b, y) \quad \overset{\text{def}}{\Leftrightarrow} \quad И c.\ (c\ a)\cdot x = (c\ b)\cdot y.$$

Lemma 9.4.14. *$\sim$ is reflexive, symmetric and transitive and hence an equivalence relation.*

Proof. Reflexivity and symmetry are trivial. For transitivity, suppose $(a, x) \sim (b, y) \sim (c, z)$. Then

$$\Big(И n.\ (n\ a)\cdot x = (n\ b)\cdot y\Big) \wedge \Big(И n.\ (n\ b)\cdot y = (n\ c)\cdot z\Big).$$

C9.4.5 tells us that И commutes with conjunction so

$$И n.\ \big((n\ a)\cdot x = (n\ b)\cdot y = (n\ c)\cdot z\big)$$

and so $(a, x) \sim (c, z)$ as required. □

We proved that $\sim$ coincides with traditional α-equivalence for `lam3` in T8.2.5. Now that we have И $(15)_{33}$ is superseded by

$$\frac{И c.\ (c\ a)\cdot t \sim (c\ b)\cdot t'}{\mathbf{Lam3}(a, t) \sim \mathbf{Lam3}(b, t')}.$$

9.5. Atom-abstraction. Recall the definition of $\sim$ in D9.4.13 and the associated (standard) notation (x, y) for pairs.

Definition 9.5.1 ($\mathbb{A}$-abstraction)**.** *For $a \in \mathbb{A}$ write the $\sim$-equivalence class of (a, x) as $a.x$ and call it the $\mathbb{A}$-**abstraction** (read "atom-abstraction" or just "abstraction"), of x by a. This defines a new function-class*

$$\mathbb{A} \times \mathcal{V}_{\mathrm{FM}} \longrightarrow \mathcal{V}_{\mathrm{FM}}$$
$$a, x \longmapsto a.x.$$

Lemma 9.5.2. *For $a \in \mathbb{A}$ and x any set, the class $a.x$ is a set.*

PROOF. $a.x$ is a subclass of

$$\mathbb{A} \times \left\{ (b\ a)\cdot x \;\middle|\; b \in \mathbb{A} \right\},$$

which is a set. By collection $a.x$ is a set. □

Notation 9.5.3 (Class of abstractions)**.** *Write* AbsClass *for the class of $\mathbb{A}$-abstractions, i.e. the sets satisfying $\psi(x)$ where*

$$\psi(x_*) = \exists a \in \mathbb{A}, x.\ x_* = a.x.$$

We shall usually name variable symbols intended to range over AbsClass *$x_*, y_*, z_*, \ldots$ unless we are thinking of them as functions f, see L9.5.4.*

Lemma 9.5.4 (Abstractions functions)**.** *$f \in$ AbsClass is a function-set.*

PROOF. It suffices to show that for all $a \in \mathbb{A}$, if $(a, x) \sim (a, x')$ then $x' = x'$. Suppose for this paragraph we know this. Then know f is a function-class with domain $\mathbb{A} \in \mathcal{V}_{\mathrm{FM}}$. Replacement tells us $\mathbf{Codom}(f) \in \mathcal{V}_{\mathrm{FM}}$ and since $f \subseteq \mathbf{Dom}(f) \times \mathbf{Codom}(f) \in \mathcal{V}_{\mathrm{FM}}$, separation tells us $f \in \mathcal{V}_{\mathrm{FM}}$ and we have the result.

So suppose $(a, x) \sim (a, x')$. We unfold D9.4.13 and obtain

$$\text{⋔}c.\ (c\ a)\cdot x = (c\ a)\cdot x'.$$

By L9.4.8 we can choose $c' \# a, x, x'$ such that

$$(c'\ a)\cdot x = (c'\ a)\cdot x'.$$

But *of course* permutations are bijections so $x = x'$ as required. □

Remark 9.5.5 (Understanding $a.x$)**.** Together D9.4.13 and L9.5.4 allow us to view an abstraction in two ways:

1. As a(n equivalence-class of) pair(s): "$a.x$ is like (a, x) but with the identity of a hidden." This is reminiscent of 'abstraction as information hiding' like that for abstract data types in [**55**]).
2. As a function: "$a.x$ is a function that takes a variable name and substitutes it in the 'hole' as appropriate." This is reminiscent of HOAS (cf. §33.2).

◇

We already know from L9.3.4 and L9.3.6 that $\mathbf{Supp}(a.x) \subseteq \{a\} \cup \mathbf{Supp}(x)$. More is true:

Lemma 9.5.6 (Support destruction)**.** *For $a \in \mathbb{A}$ and $x \in$ FM,*

$$a \# a.x.$$

PROOF. By the technical result L9.5.7 that follows, this amounts to proving

$$\mathsf{N}b.\ (b\ a)\cdot(a.x) = a.x.$$

We use L9.4.8 to pick $b'\#a, x$ and use equivariance to simplify this to

$$(b'.(b'\ a)\cdot x) = a.x.$$

Unpack the definition of $\sim$ (D9.4.13). It now suffices to prove

$$\mathsf{N}c.\ (c\ b')\cdot(b'\ a)\cdot x = (c\ a)\cdot x \quad \textit{so by L9.4.8 we pick } c' \textit{ such that}$$
$$c'\#a, b', x\ \wedge\ (c'\ b')\cdot(b'\ a)\cdot x = (c'\ a)\cdot x.$$

$c'\#b', a, x$ amounts (L9.3.6) to $c' \neq b', a$ and $c'\#x$. We can now apply L9.2.3 to deduce

$$(c'\ a) = (b'\ a)\cdot((b'\ a)\cdot c'\ (b'\ a)\cdot b') = (c'\ b')\cdot(b'\ a)$$

and hence obtain the result. □

The following technical lemma, used in the preceding proof, adds one more (rather useful) proof-method to the two described in R9.3.1. In the proofs of L9.5.4 and L9.5.6 I used L9.4.8, we could do the same here but it seems nicer to go back to T9.4.6 and bring out the $\forall$-$\exists$ duality of N directly:

Lemma 9.5.7 (Extra proof-method for #)**.**

$$a\#x \iff \mathsf{N}b.\ (b\ a)\cdot x = x$$

PROOF. By $(21)_{41}$ reduce RHS to $\forall b\#a, x.\ (b\ a)\cdot x = x$. Then LHS implies RHS from Property 1 of L9.2.7.

By $(23)_{41}$ reduce RHS to $\exists b\#a, x.\ (b\ a)\cdot x = x$. Then RHS implies LHS from Property 3 of L9.2.7. □

Remark 9.5.8 (Different development)**.** In fact we can take L9.5.7 as the definition of # and define $\mathbf{Supp}(x) \stackrel{\text{def}}{=} \{a \in \mathbb{A} \mid \neg(a\#x)\}$. We do this in Isabelle/FM, see §16.8 and R16.8.4). ◇

Corollary 9.5.9 (Support of $\mathbb{A}$-abstraction)**.**

$$\boldsymbol{Supp}(a.x) = \boldsymbol{Supp}(x) \setminus \{a\}$$

PROOF. By L9.5.6 and the comment which precedes it. □

This answers R9.3.7:

Remark 9.5.10 (Is **Supp** equal to **FV**?)**.** Recall that we imagine $a.x$ to be "x with one bound variable". C9.5.9 is a strong hint that we should imagine $\mathbf{Supp}(x)$ to be "the free atoms (variables) of x". We shall make this rigorous in L10.7.7. ◇

It is useful to have a more concrete characterisation of abstractions. The following corresponds precisely to a result first found useful in Isabelle/FM (see §16.9 and the discussion of `Abs_twiddle` on *p.*134).

Lemma 9.5.11 (Concrete version of abstractions)**.** *Write*

$$a.'x = \left\{(b, (b\ a)\cdot x) \;\middle|\; b = a \vee b\#x\right\}. \tag{27}$$

Then $a.'x = a.x$.

Proof. *Left-to-right inclusion.* Observe that $(a, x) \in a.x$ (reflexivity of $\sim$ in L9.4.14). Now C9.5.9 tells us $\mathbf{Supp}(a.x) = \mathbf{Supp}(x) \setminus \{a\}$ and combined with L9.2.7 we know

$$b\#x \implies (b\ a)\cdot(a.x) = a.x.$$

(Repeated use of) R8.1.13 yields

$$(b, (b\ a)\cdot x) = (b\ a)\cdot(a, x) \in (b\ a)\cdot a.x = a.x.$$

Therefore, for $b = a$ and for $b\#x$, $(b, (b\ a)\cdot x) \in a.x$ and therefore

$$a.'x \subseteq a.x.$$

Right-to-left inclusion and equality. Suppose $(b, (b\ a)\cdot x) \in a.'x$ for $b = a \vee b\#x$. We want to show $(b, y) \in a.x$, which amounts to showing that

$$И c.\ (c\ a)\cdot x = (c\ b)\cdot(b\ a)\cdot x.$$

Strip И by choosing some $c\#a, x, b$ by L9.4.8. Now if $b = a$ there is nothing to prove because $(a\ a)\cdot x = x$ and we know as in the previous half of the proof that $(a, x) \in a.x$. So we may safely assume $b \neq a$, which amounts by L9.3.6 to $b\#a$, and also $b\#x$. Now $(c\ b)\cdot(b\ a)\cdot x = (c\ a)\cdot(c\ b)$ by $(18)_{36}$. But $c, b\#x$ so by Item 1 of L9.2.7 we have $(c\ b)\cdot x = x$. Therefore we must show

$$И c.\ (c\ a)\cdot x = (c\ a)\cdot x.$$

This is trivial (case $\top$ of C9.4.5). □

The following corresponds to `Abs_twiddle` in Isabelle/FM, see R16.9.2 and the preceding discussion starting *p.*134. It has a few useful corollaries:

Corollary 9.5.12 (Equality test)**.** *$a.x = b.y$ precisely when [$b\#x$ or $b = a$] and $y = (b\ a)\cdot x$.*

Corollary 9.5.13 (Domain of $x_* \in$ AbsClass)**.** *Let $x_* \in$ AbsClass be an abstraction. Then x_* is a function-set with domain precisely $\mathbb{A} \setminus \boldsymbol{Supp}(x_*)$.*

Proof. By L9.5.4, x_* is a function-set. By N9.5.3 $x_* = a.x$ for some a, x. By L9.5.11 it is a partial function with domain precisely $\mathbf{Supp}(x) \setminus \{a\}$, which by C9.5.9 is equal to $\{n \in \mathbb{A} \mid n\#x_*\}$. □

We can extract a new function-class from C9.5.13:

Definition 9.5.14 (Concretion)**.** *If f is an $\mathbb{A}$-abstraction and $b\#f$ we write $f@b$ for the result of applying f as a function to b. We call this the "**concretion of f at b**".*

Lemma 9.5.15 (Calculate @)**.** $(a.x)@b = (b\ a)\cdot x$ *for* $b \notin \boldsymbol{Supp}(x) \setminus \{a\}$.

Proof. Read directly off $(27)_{47}$ in L9.5.11. □

This lemma has a form of dual:

Lemma 9.5.16. *For $x_* \in$ AbsClass (N9.5.3) an abstraction and $a\#x_*$,*

$$a.(x_*@a) = x_*.$$

Proof. Since $x_* \in$ AbsClass we know $x_* = b.y$ for some b, y. By C9.5.9, Item 1 of L9.2.7, and L8.1.12 for abstraction, $b.y = a.(a\ b)\cdot y$. So WLOG we can take $b = a$. By L9.5.15 we know $x_*@a = (a\ a)\cdot y = y$. This gives us the result. □

This result has two useful corollaries:

Corollary 9.5.17. *For $x_* \in$ AbsClass an abstraction and $a \in \mathbb{A}$,*

$$(\exists x.\ x_* = a.x) \iff a\#x_*.$$

Proof. For the R-L implication we can use L9.5.16 and take $x = x_*@a$. Conversely we use L9.5.6 or the later C9.5.9. □

Corollary 9.5.18. *For $x_*, y_* \in$ AbsClass abstractions*

$$(\mathsf{И}c.\ x_*@c = y_*@c) \implies x_* = y_*.$$

Proof. Suppose $\mathsf{И}c.\ x_*@c = y_*@c$. Then

$$\mathsf{И}c.\ c.(x_*@c) = c.(y_*@c).$$

By L9.5.16 we have

$$\mathsf{И}c.\ x_* = y_*$$

and therefore by C9.4.9 that $x_* = y_*$ as required.[20] □

[20]The reader might wonder what the И quantifier is doing; we do not seem to morally use it. But we do: it guarantees that $c\#x_*, y_*$ so that the concretions $x_*@c$ and $y_*@c$ exist (D9.5.14 and C9.5.13).

It is clear from C9.5.9 that atom-abstraction is a synthetic version of the abstraction term-former `a.t` first suggested in $(4)_{16}$. Now, perhaps for the first time, FM pays a convincing dividend. Concretion @ is clearly a synthetic version of the destructor for the abstraction term-former in $(4)_{16}$, call it `@`, and we can work back to deduce a good typing rule for it. It would look something like this:

$$\frac{\texttt{x':[Atm]X a:Atm a\#x'}}{\texttt{x'@a:X}} \tag{28}$$

where `a#x'` is a syntactic version of #. We need this condition so that $x' = [\![\texttt{x'}]\!]$ is defined at $a = [\![\texttt{a}]\!]$. So notice that our underlying set theory will mean that we cannot concrete an abstraction at `a` if `a` occurs free in it (specifically see 9.5.17). The underlying reason is that abstraction and concretion are implemented using permutations $(b\ a)$ (D9.5.1 and L9.5.15) and not name-for-name substitution $[b/a]$. Cf. §11.1.

It remains to construct the abstraction-set function-class corresponding to the abstraction-type `[Atm]X` for which `a.t` was a term-former.

9.6. Abstraction sets.

Definition 9.6.1 (Abstraction sets)**.** *For a set X we define the **abstraction set** or **abstraction type** of X to be*

$$[\mathbb{A}]X \stackrel{\text{def}}{=} \{a.x \mid a \in \mathbb{A} \wedge x \in X \wedge a\#X\}. \tag{29}$$

$[\mathbb{A}]X$ is intended to capture the idea of

> "Elements $x \in X$ with one distinguished abstracted (bound) atom (variable name)."

We can use T9.4.6 and R9.4.12 to rewrite this definition as

$$[\mathbb{A}]X = \{x' \mid \text{И}a.\ \exists x \in X.\ x' = a.x\}. \tag{30}$$

Looking at $(4)_{16}$, the following **false** definition seems plausible:

$$[\mathbb{A}]'X \stackrel{\text{def}}{=} \{a.x \mid a \in \mathbb{A} \wedge x \in X\}. \tag{FALSE}$$

The problem is, we want $[\mathbb{A}]X$ to provide a semantics for the type `[Atm]X` and in particular for the typing rules $(4)_{16}$ and $(28)_{49}$. Both $[\mathbb{A}]X$ and $[\mathbb{A}]'X$ satisfy the former rule but only $[\mathbb{A}]X$ satisfies the latter. We provide a lemma for $[\mathbb{A}]X$ and a counterexample for $[\mathbb{A}]'X$.

Lemma 9.6.2 ($[\mathbb{A}](-)$ OK)**.** $[\mathbb{A}](-)$ *satisfies* $(28)_{49}$*, i.e. for all X,*

$$\forall x_* \in [\mathbb{A}]X, a \in \mathbb{A}.\ \ a\#x_* \implies x_*@a \in X \tag{31}$$

PROOF. Suppose $a \in \mathbb{A}$, $x_* \in [\mathbb{A}]X$ and $a\#x_*$. We use $(30)_{49}$ to deduce that for $b\#a, x_*, X$ and some $y \in X$, $x_* = b.y$. Then by L9.5.15,

$$x_*@a = (b.y)@a = (a\ b)\cdot y.$$

Now $y \in X$ and $a, b\#X$ so

$$(a\ b)\cdot y \overset{\text{equivariance, } (12)_{30}}{\in} (a\ b)\cdot X \overset{\text{Point 1, L9.2.7}}{=} X.$$

□

Lemma 9.6.3 ($[\mathbb{A}]'(-)$ not OK)**.** $[\mathbb{A}]'X$ *is unsuitable to model* `[Atm]X` *because it does not satisfy* $(31)_{49}$*.*

PROOF. By counterexample. Take $X = \{a\}$. Then

$$[\mathbb{A}]'\{a\} = \left\{b.a \;\middle|\; b \in \mathbb{A}\right\}.$$

C9.5.12 tells us this is a two-element set which we can informally write $\{a.a, b.a\}$. Suppose $b \in \mathbb{A}$ and $b\#X = \{a\}$. Then $b \neq a$ and therefore $b \notin X$, but $(a.a)@b = b$. □

Incidentally, $[\mathbb{A}]\{a\}$ is (by C9.5.12 and writing informally) the singleton $\{b.a\}$.

Remark 9.6.4. These questions are academic because 'normal' types such as `Nat`, `Atm` and `lam3` have equivariant semantics for which $[\mathbb{A}]X$ and $[\mathbb{A}]'X$ coincide because $\mathbf{Supp}(X) = \emptyset$ in that case. In fact $(4)_{16}$ is simplistic and valid only in an equivariant typing system (i.e. such that the denotations of all types are equivariant). With dependent types like `(x:Atm)(x~=a)` we must reconsider the intro-rule. FM pays another dividend. ◇

The following result is *not* needed in the subsequent development, but it ties up the discussion of $[\mathbb{A}]X$ as a set quite nicely:

Theorem 9.6.5. *For a set* X *and* $a\#X$*,*

$$X \cong \left\{x_* \in [\mathbb{A}]X \;\middle|\; a\#x_*\right\}.$$

PROOF. The function $\lambda x_* \in [\mathbb{A}]X.x_*@a$ maps to X by L9.6.2. By construction (see $(29)_{49}$) $\lambda x \in X.(a.x)$ maps to $[\mathbb{A}]X$. We use L9.5.15 and L9.5.16 to show that together they are two halves of a bijection. □

We conclude this section by analysing the function-sets out of $[\mathbb{A}]X$. This will be useful when we extend $[\mathbb{A}](-)$ to a functor on **SetsFM** (D10.1.1) in L10.1.5.

Theorem 9.6.6 (Functions out of abstraction sets)**.** *There is a surjection onto functions* $\bar{f}\colon [\mathbb{A}]X \to Y$ *from functions* $f\colon \mathbb{A} \times X \to Y$ *such that* $\phi(f)$ *where* ϕ *is defined by*

$$\phi(f) \stackrel{\text{def}}{=} И a.\ \forall x \in X.\ a\#f(a,x).$$

In addition there is a bijection between equivariant *(N9.2.8)* $\bar{f}\colon [\mathbb{A}]X \to Y$ *and* equivariant $f\colon \mathbb{A} \times X \to Y$ *such that* $\phi(f)$*.*

Note that by T9.4.6 we can read $\phi(f)$ as

$$\forall a \in \mathbb{A}, x \in X.\ a\#X \implies a\#f(a,x).$$

Proof. To $f\colon \mathbb{A} \times X \to Y$ such that $\phi(f)$ we associate the function

$$\begin{aligned}\bar{f}\colon [\mathbb{A}]X &\to Y \cup \{\emptyset\}\\ x_* &\mapsto \iota y.И a.\ y = f(a, x_*@a)\end{aligned}$$

Here ι is a ***unique choice function*** that returns the unique x such that $\psi(x)$ if this exists and $\emptyset$ otherwise. See R11.4.3.

We must show that $\bar{f}$ maps to Y, not just $Y \cup \{\emptyset\}$, i.e. for all $x_* \in [\mathbb{A}]X$ there exists unique y such that $И a.\ y = f(a, x_*@a)$.

Existence. Choose $a\#f, x_*$ and take $y = f(a, x_*@a)$. Observe from $\phi(f)$ we know $a\#y$. We must verify

$$И b.\ y = f(b, x_*@b) \quad \text{equivalent by T9.4.6 to} \quad \exists b.\ (b\#f, x_*, y)\ \wedge y = f(b, x_*@b),$$

So choose $b = a$.

Uniqueness. Suppose $y, y' \in Y$ satisfy

$$\big(И a.\ y = f(a, x_*@a)\big) \wedge \big(И a.\ y' = f(a, x_*@a)\big).$$

By C9.4.5 $И$ distributes over $\wedge$ and so

$$И a.\ y = f(a, x_*@a) = y'.$$

We can now use L9.4.8 to choose some appropriate $a\#y, f, y'$, and deduce $y = y'$ as required.

Conversely, to $f\colon [\mathbb{A}]X \to Y$ associate

$$\begin{aligned}\hat{f}\colon \mathbb{A} \times X &\to Y\\ (a,x) &\mapsto f(a.x).\end{aligned}$$

For the surjection it suffices to show $\bar{\hat{f}} = f$. We know $\bar{\hat{f}}(x_*)$ is that unique y such that

$$\text{И}a.\ y = \hat{f}(a, x_*@a) \iff \text{И}a.\ y = f(a.x_*@a) \overset{L9.5.16}{\iff} y = f(x_*).$$

But then $y = f(x_*)$ as required.

For the bijection, it suffices to show that $\bar{f_1}$ and $\hat{f_2}$ are equivariant if f_1 and f_2 are, and that $\hat{\bar{f}} = f$. Equivariance follows by L9.3.4 (no increase of support). Suppose now that f is equivariant. $\hat{\bar{f}}(a, x)$ equals $\bar{f}(a.x)$, which is that unique y such that

$$\text{И}b.\ y = f(b, (a.x)@b) \overset{L9.5.15}{\iff} \text{И}b.\ y = f(b, (b\ a)\cdot x)$$

Use L9.4.8 to choose fresh $b\#f, a, x, y$. Since $\mathbf{Supp}(f) = \emptyset$, $\phi(f)$ implies that for *any* a, $a\#f(a, x)$. Thus

$$f(a, x) \overset{L9.2.7}{=} (b\ a)\cdot f(a, x) \overset{equivr.\ L8.1.12}{=} f(b, (b\ a)\cdot x) = y,$$

as required. □

Corollary 9.6.7 (**fresh**). *There is a surjection from partial functions*

$$f\colon \mathbb{A} \longrightarrow Y \quad \textit{such that} \quad \text{И}a.\ a\#f(a) \quad \textit{to elements} \quad \bar{f} \in Y,$$

and a bijection *between equivariant f such that* $\text{И}a.\ a\#f(a)$ *and equivariant elements of Y.*

When we write $\boldsymbol{fresh}\, a.f(a)$ *we say this is a "**legal use of fresh**" to mean that* $\text{И}a.\ a\#f(a)$ *(so* $\boldsymbol{fresh}\, a.f(a)$ *is well-defined).*

We write $\bar{f}$ as

$$\boldsymbol{fresh}\, a.f(a).$$

Very usefully, $\boldsymbol{fresh}\, a.f(a)$ *satisfies the following* ***characteristic equality***:[21]

(32) $\text{И}a.\ \Big(f(a) = \boldsymbol{fresh}\, b.f(b)\Big)$

or use L9.4.8 to pick new a', and $f(a') = \boldsymbol{fresh}\, b.f(b)$.

Proof. In T9.6.6 above set $X = \mathbf{1} = \{\emptyset\}$ so $\mathbb{A} \times X \cong \mathbb{A}$, and model partiality by taking $Y = Y' + \{\emptyset\}$. The results follow by working out the implications of this in the proof of T9.6.6. □

We can read $\mathbf{fresh}\, a.f$ as

[21] ... given in two flavours. In Part 1 of the proof of C9.6.9 I explicitly convert the first flavour into the second, see $(33)_{54}$. Sometimes I favour the first, for complete rigour about what a' is new for (everything free in the scope of the **fresh**/И binder, at least), sometimes I prefer the second, for convenience.

> "Pick a new a and calculate $f(a)$. It doesn't matter which a we choose because $a\#f(a)$."

fresh is an object-level version of the meta-level И-quantifier. It shares many of its nice properties. For example:

Lemma 9.6.8 (**fresh** well-behaved)**.** *For function-sets* $f\colon \mathbb{A} \to X$ *and* $g\colon \mathbb{A} \to Y$*,*

$$\begin{aligned}\boldsymbol{fresh}\, a.\, (f\, a, g\, a) &= (\boldsymbol{fresh}\, a.\, f\, a, \boldsymbol{fresh}\, a.\, g\, a)\\ \boldsymbol{fresh}\, a.\, \boldsymbol{Inl}(f\, a) &= \boldsymbol{Inl}(\boldsymbol{fresh}\, a.\, f\, a)\\ \boldsymbol{fresh}\, a.\, \boldsymbol{Inr}(f\, a) &= \boldsymbol{Inr}(\boldsymbol{fresh}\, a.\, f\, a)\end{aligned}$$

In the case that $X = A \to B$ *and* $Y = B \to C$*,*

$$\boldsymbol{fresh}\, a.\, (f(a) \circ g(a)) = (\boldsymbol{fresh}\, a.\, f(a)) \circ (\boldsymbol{fresh}\, a.\, g(a))$$

Proof. Consider only the last case, the others are similar. By $(32)_{52}$ we have

$$\begin{aligned}И a.\ \Big(f(a) \circ g(a) = \mathbf{fresh}\, b.\, (f(b) \circ g(b))\Big)\ \wedge\\ И a.\ \Big(f(a) = \mathbf{fresh}\, b.\, f(b))\ \wedge\ И a.\ (g(a) = \mathbf{fresh}\, b.\, g(b)\Big).\end{aligned}$$

И distributes over conjunction (C9.4.5) so this implies

$$И a.\ \Big(\mathbf{fresh}\, b.\, (f(b) \circ g(b))\ =\ f(a) \circ g(a)\ =\ \mathbf{fresh}\, b.\, f(b) \circ \mathbf{fresh}\, b.\, g(b)\Big)$$

Now we can convert И to an existential quantifier and pick an appropriate new a (L9.4.8). We then throw away the middle part of the equality and obtain just

$$\mathbf{fresh}\, b.\, (f(b) \circ g(b)) = \mathbf{fresh}\, b.\, f(b) \circ \mathbf{fresh}\, b.\, g(b)$$

as required. □

fresh has many more nice properties, we explore one other in L9.6.10.

$[\mathbb{A}]X$ is well-behaved too:

Corollary 9.6.9. $[\mathbb{A}](-)$ *is remarkably well-behaved:*

$$\begin{aligned}[\mathbb{A}](X \times Y) &\cong [\mathbb{A}]X \times [\mathbb{A}]\, Y\\ [\mathbb{A}](X + Y) &\cong [\mathbb{A}]X + [\mathbb{A}]\, Y\\ [\mathbb{A}](X \to Y) &\cong [\mathbb{A}]X \to [\mathbb{A}]\, Y\end{aligned}$$

Proof. In the first case the bijection is given by

$$\begin{aligned}F = \mathbf{fresh}\, a.\, (a.\pi_X(z_*@a), a.\pi_Y(z_*@a)) &\quad : [\mathbb{A}](X \times Y) \longrightarrow [\mathbb{A}]X \times [\mathbb{A}]\, Y\\ G = \mathbf{fresh}\, a.\, (a.(x_*@a, y_*@a)) &\quad : [\mathbb{A}]X \times [\mathbb{A}]\, Y \longrightarrow [\mathbb{A}](X \times Y).\end{aligned}$$

(where π_X and π_Y are projection functions from $X \times Y$ to X and Y respectively). The second case is similar. I omit the proofs and move on the the third bijection.

The fact that the third bijection exists is really very nice, let me seize the moment and point out just how excellently the equational theory of FM turns out. The bijection is given by

$$\begin{array}{lcl} F\colon [\mathbb{A}](X \to Y) & \longrightarrow & ([\mathbb{A}]X \to [\mathbb{A}]\,Y) \\ \quad f_* & \longmapsto & \lambda x_* : [\mathbb{A}]X.\mathbf{fresh}\; a.\, [a.(f_*@a)(x_*@a)] \\ G\colon ([\mathbb{A}]X \to [\mathbb{A}]\,Y) & \longrightarrow & [\mathbb{A}](X \to Y) \\ \quad g & \longmapsto & \mathbf{fresh}\; a.\, [a.\Big(\lambda x : X.g(a.x)@a\Big)]. \end{array}$$

We now prove that $F(G(g)) = g$ and $G(F(f_*)) = f_*$. I shall do so in a very equational style.

1• *We prove* that $F(G(g))$ and g are extensionally equal, i.e. for all $x_* \in [\mathbb{A}]X$

$$LHS = F(G(g))(x_*) = g(x_*) = RHS,$$

by expanding the definitions of F and G and simplifying. First we expand F and G on the LHS:

$$\begin{aligned} LHS &= \mathbf{fresh}\; a.\; a.(G(g)@a)(x_*@a) \\ &= \mathbf{fresh}\; a.\; a.((\Big(\mathbf{fresh}\; b.\; b.\big(\lambda x : X.g(b.x)@b\big)\Big)@a)(x_*@a)) \end{aligned}$$

We now apply (for demonstration purposes, the first flavour of) the characteristic equality of **fresh** $(32)_{52}$ twice, and deduce

$$\begin{aligned} (33)\quad \mathsf{M}a', b'.\; \mathbf{fresh}\; a.\; a.((\Big(\mathbf{fresh}\; b.\; b.\big(\lambda x : X.g(b.x)@b\big)\Big)@a)(x_*@a)) \;=\; \\ a'.((\Big(b'.\big(\lambda x : X.g(b'.x)@b\big)\Big)@a)(x_*@a')) \end{aligned}$$

We use L9.4.8 to choose new a', b' such that $a'\#b'$ (and so $a' \neq b'$ and $b'\#a'$, see L9.3.6), and also $a', b'\#X, g, x_*$ and indeed a', b' apart from anything else we wish. We know

$$\begin{aligned} \mathbf{fresh}\; a.\; a.((\Big(\mathbf{fresh}\; b.\; b.\big(\lambda x : X.g(b.x)@b\big)\Big)@a)(x_*@a)) \;=\; \\ a'.((\Big(b'.\big(\lambda x : X.g(b'.x)@b'\big)\Big)@a')(x_*@a')) \end{aligned}$$

Now we have something we can simplify.

$$\Big(b'.\big(\lambda x : X.g(b'.x)@b'\big)\Big)@a'$$

$$\begin{aligned}
&\overset{(n.x)@m=(n\ m)\cdot x,\ L9.5.15}{=} && (b'\ a')\cdot\lambda x : X.g(b'.x)@b' \\
&\overset{equivariance,\ L8.1.12}{=} && \lambda x : (b'\ a')\cdot X.((b'\ a')\cdot g)(a'.x)@a' \\
&\overset{a',b'\#X,g,\ L9.2.7}{=} && \lambda x : X.g(a'.x)@a'.
\end{aligned}$$

So we can simplify the LHS as

$$a'.((\lambda x : X.g(a'.x)@a')(x_*@a'))$$

$$\begin{aligned}
&\overset{\beta\text{-reduction}}{=} && a'.(g(a'.(x_*@a'))@a')) \\
&\overset{a.(x_*@a)=x_*,\ L9.5.16}{=} && a'.\big((g(x_*))@a'\big) \\
&\overset{a.(x_*@a)=x_*,\ L9.5.16}{=} && g(x_*),
\end{aligned}$$

so as required $LHS = RHS$.

2• *We prove* that $G(F(f_*)) = f_*$. By C9.5.18 it suffices to show that for new c, $G(F(f_*))@c = f_*@c$, that is,

$$\mathsf{N}c.\ \forall x \in X.\ \Big((GF(f_*)@c)(x) = (f_*@c)(x)\Big).$$

So let us use L9.4.8 to choose new $c\#f_*, X, Y$ and anything else we want, and let us pick some $x \in X$ (note we choose x *after* c so we do *not* know $c\#x$). We want to prove

$$LHS = (GF(f_*)@c)(x) = (f_*@c)(x) = RHS.$$

As before we expand the LHS and obtain

$$LHS =$$

$$((\mathbf{fresh}\ a.\ [a.\Big(\lambda x : X.\big[\Big(\lambda x_* : [\mathbb{A}]X.\mathbf{fresh}\ b.\ [b.(f_*@b)(x_*@b)]\Big)(a.x)\big]@a\Big)])@c)(x)$$

Now there is an ***instructive complication***. We might like to, as in the previous case, eliminate **fresh** using $(32)_{52}$ and L9.4.8 to replace a, b by a', b'. We can do so for a but not for b; it is under the scope of $\lambda x_* : [\mathbb{A}]X.\mathit{stuff}$, which means that for each different x_* we would need to choose a different fresh b'. Since x_* is not fixed within the λ-abstraction, we cannot pick a particular b'. However, we can eliminate $\mathbf{fresh}a$ and β-reduce $\big[\Big(\lambda x_* : [\mathbb{A}]X.\mathbf{fresh}\ b.\ [b.(f_*@b)(x_*@b)]\Big)(a'.x)\big]$, and finally use L9.5.15 to obtain

$$LHS = ([a'.\Big(\lambda x : X.[\mathbf{fresh}\ b.\ b.(f_*@b)((b\ a')\cdot x)]@a\Big)]@c)(x).$$

Now we can reduce $(a'.T)@c$ to $(c\ a')\cdot T$ using L9.5.15;

$$LHS = ((c\ a')\cdot\Big(\lambda x : X.[\mathbf{fresh}\, b.\, b.(f_*@b)((b\ a')\cdot x)]@a\Big))(x).$$

and then pull the permutation in using equivariance L8.1.12. Bear in mind that $a', c\#X, f_*$ so by L9.2.7 $(c\ a')\cdot X = X$ and $(c\ a')\cdot f_* = f_*$. We obtain

$$LHS = (\Big(\lambda x : X.[\mathbf{fresh}\, b.\, b.(f_*@b)((b\ c)\cdot x)]@c\Big))(x).$$

We now β-reduce to obtain

$$LHS = (\mathbf{fresh}\, b.\, b.(f_*@b)((b\ c)\cdot x))@c.$$

Now we can eliminate $\mathbf{fresh}b$ for $b'\#c, x, f_*, X, \ldots$ and simplify using L9.5.15 to obtain

$$LHS = (b'\ c)\cdot((f_*@b')((b'\ c)\cdot x)).$$

We now use equivariance L8.1.12 and the fact that $(b'\ c)\cdot(b'\ c) = Id$ to obtain

$$LHS = (f_*@c)(x)) = RHS$$

as required.

□

We can encapsulate the result which is *not* true in the 'instructive complication' in case 2 of C9.6.9 above in a lemma extending L9.6.8:

Lemma 9.6.10 (**fresh** better-behaved)**.** *For $f(a, \vec{x}, y)$ and $t(b, a, \vec{x})$ terms with free variables contained in (the sets of pairwise distinct variables) $a, y, \vec{x}$ and $b, a, \vec{x}$ (so in particular b is not in the free variables of f), we have*

$$f(a, \vec{x}, \boldsymbol{fresh}\, b.\, (t(b, a, \vec{x}))) = \boldsymbol{fresh}\, b.\, f(a, \vec{x}, t(b, a, \vec{x})).$$

Proof. By using $(32)_{52}$ like in Case 1 of C9.6.9, details omitted. □

L9.6.8 was not applicable before because t was of the form $t(b, a, \vec{x}, y)$ where y occurred bound by λy in f. It is an important result although we do not apply it much in this document. It is used in $(52)_{69}$.

10. Datatypes in FM

10.1. Introduction. In this section we develop a framework for declaring a class of datatypes with variable binding and constructing FM-sets of syntax for them.

Henceforth we shall be less meticulous about N8.2.4 and may refer to "sets of syntax" as "syntax", to "semantic terms" as "terms", etc.

The idea of datatypes as initial algebras goes back to [**20**]. It is a useful way of thinking which is now standard. The set-theoretic universe $\mathcal{V}_{\mathrm{FM}}$ corresponds

naturally to a category of FM-sets **SetsFM** defined below in D10.1.1. Since it is convenient to develop datatypes in a category-theoretic context, we continue the development of FM set theory treating $\mathcal{V}_{\mathrm{FM}}$ as a category **SetsFM**.

Recall the theory of categories **C**, see for example [**43**, §I, p.7] and [**3**, Vol I,Def 1.2.1]. We write the collection of objects of **C** as Obj(**C**) and the collection of arrows from X to Y as $\mathbf{C}(X, Y)$.

Definition 10.1.1 (**SetsFM**)**.** *We introduce a* ***category of*** FM***-sets****, written* **SetsFM**. *Objects of* **SetsFM** *are the* FM*-sets* $\mathcal{V}_{\mathrm{FM}}$*. The arrows from* X *to* Y *are the set-functions* $f\colon X \to Y \in \mathcal{V}_{\mathrm{FM}}$*:*

$$\mathrm{Obj}(\mathbf{SetsFM}) = \mathcal{V}_{\mathrm{FM}} \quad \textit{and} \quad \mathbf{SetsFM}(X, Y) = \left\{ f \in \mathcal{V}_{\mathrm{FM}} \;\middle|\; f\colon X \to Y \right\}.$$

We *shall* adhere to the convention of N4.5 of writing syntax `blah` and the corresponding **SetsFM**-semantics as $[\![\texttt{blah}]\!]$ or **blah**. We also follow the convention of naming variables of abstraction type with starred names as in x_*, or primed names `x'` in typewriter font.

Definition 10.1.2 (Basic types, strings, variables)**.** *We introduce a set of* ***basic type symbols*** *written* ***BTypeSymb****,*

$$\boldsymbol{BTypeSymb} \stackrel{\mathrm{def}}{=} \{\mathtt{U}, \mathtt{N}, \mathtt{A}\},$$

and a set of ***basic types*** *written* ***BType***

$$\boldsymbol{BType} \stackrel{\mathrm{def}}{=} \{\mathbb{U}, \mathbb{N}, \mathbb{A}\}.$$

Here $\mathbb{U} = \{*\}$ *is a unit set with one element* $*$ *and* `U` *is intended to represent some unit type with one element* `*`*. It is useful to define a function*

$$\begin{aligned} \beta\colon \boldsymbol{BTypeSymb} &\longrightarrow \boldsymbol{BType} \\ \mathtt{U}, \mathtt{N}, \mathtt{A} &\longmapsto \mathbb{U}, \mathbb{N}, \mathbb{A} \quad \textit{respectively.} \end{aligned} \tag{34}$$

We introduce a set of strings `str` $\in$ ***Strings****, which we assume for the sake of argument consist of ASCII text strings such as* `Var2` *and* `Lam3`*. Finally, we hypothesise an infinite collection of* ***binding signature variables*** $X, Y, Z, \ldots \in$ ***BindVar****.*

Definition 10.1.3 (Binding signatures)**.** *The grammar of Fig.*4_{58} *defines* ***Binding signatures****.*[22]

We have developed no theory yet, but here are a few binding signatures we might like to write:

[22] The idea is from [**17**, §2, p.5].

$N_X ::= \texttt{nil} \mid [\texttt{A}]^n X \texttt{ x } N \mid C \texttt{ x } N$

$B_X ::= \texttt{nil} \mid (\texttt{str of } N_X) \texttt{ + } B_X.$

$n \in \mathbb{N}, C \in \mathbf{BTypeSymb}, X \in \mathbf{BindVar}$

Brackets may be dropped where convenient.

FIGURE 4. D10.1.3 - Binding Signatures

1. The untyped λ-calculus not up to α-equivalence.

$$M_X \stackrel{\text{def}}{=} \texttt{Var of A + App of } X \texttt{ x } X \texttt{ + Lam of A x } X.$$

2. The untyped λ-calculus up to α-equivalence.

$$L_X \stackrel{\text{def}}{=} \texttt{Var of A + App of } X \texttt{ x } X \texttt{ + Lam of } [\texttt{A}]X.$$

3. Terms of a simple π-calculus up to α-equivalence (see for example [**7**, §2, p.4] or the original [**53**, Part I, p.10]).

$$\begin{aligned} P_X \stackrel{\text{def}}{=}\ & \texttt{Zero of U + Out of A x A x } X \texttt{ + In of A x } [\texttt{A}]X \texttt{ +} \\ & \texttt{Silent of } X \texttt{ + Nu of } [\texttt{A}]X \texttt{ + Par of } X \texttt{ x } X \texttt{ + Sum of } X \texttt{ x } X \texttt{ +} \\ & \texttt{Match of A x A x } X \texttt{ + Mismatch of A x A x } X. \end{aligned} \tag{35}$$

Clearly binding signatures are designed to allow us to build sets of syntax in FM, we shall do this in §10.2.

The standard development of algebras in categories demands that we associate to B_X a functor $\mathbf{SetsFM} \to \mathbf{SetsFM}$. We abuse notation and write it $\lambda X.B_X$. As our notation suggests $\lambda X.B_X$ will be 'bolted together' out of disjoint sum, product, and n-ary abstraction sets, see D10.1.3. First we must develop the theory.

Notation 10.1.4 ($[\mathbb{A}]^n X$)**.** *We write*

$$[\mathbb{A}]^n X \quad \textit{as shorthand for} \quad \overbrace{[\mathbb{A}][\mathbb{A}]\ldots[\mathbb{A}]}^{n \textit{ times}}(X)$$

$$\textit{and} \quad \mathbb{A}^n \quad \textit{as shorthand for} \quad \overbrace{\mathbb{A} \times \mathbb{A} \ldots \times \mathbb{A}}^{n \textit{ times}}.$$

The functorial actions of disjoint sum + and product $\times$ are derived from the $\mathcal{V}_{\mathrm{FM}}$ function-classes in a standard way given in Fig.5$_{59}$. The functorial action of $[\mathbb{A}](-)$ is also given and proved well-defined and functorial in the following lemma:

$$\times : \mathbf{SetsFM} \times \mathbf{SetsFM} \to \mathbf{SetsFM}$$
$$X, Y \quad \mapsto (X, Y)$$
$$f\colon X \to X', g\colon Y \to Y' \mapsto f \times g \stackrel{\text{def}}{=} \lambda(x, y).(f(x), g(y)) : X \times Y \to X' \times Y'.$$

$$+\colon \mathbf{SetsFM} \times \mathbf{SetsFM} \to \mathbf{SetsFM}$$
$$X, Y \quad \mapsto X + Y$$
$$f\colon X \to X', g\colon Y \to Y' \mapsto f + g \stackrel{\text{def}}{=} \lambda z. \begin{cases} \mathbf{Inl}(f(z')) & z = \mathbf{Inl}(z') \\ \mathbf{Inr}(g(z')) & z = \mathbf{Inr}(z') \end{cases} : X + Y \to X' + Y'$$

$$[\mathbb{A}](-)\colon \mathbf{SetsFM} \to \mathbf{SetsFM}$$
$$X \mapsto [\mathbb{A}]X$$
$$f\colon X \to Y \mapsto [\mathbb{A}]f \stackrel{\text{def}}{=} \lambda x_*.\mathbf{fresh}\, a.\, (a.f(x_*@a)) : [\mathbb{A}]X \to [\mathbb{A}]\, Y.$$

We have dropped typing conditions in λ-terms for clarity (e.g. in the last clause $\lambda x_*.\mathbf{fresh}\ldots$ should read $\lambda x_* \in [\mathbb{A}]X.\mathbf{fresh}\ldots$).
fresh is introduced in C9.6.7.

FIGURE 5. Functorial action of $\times$, $+$ and $[\mathbb{A}](-)$

Lemma 10.1.5 ($[\mathbb{A}](-)$ functor)**.** *We can extend* $[\mathbb{A}](-)\colon \mathcal{V}_{\mathrm{FM}} \to \mathcal{V}_{\mathrm{FM}}$ *to a functor on* **SetsFM** *defined as in Fig.5*$_{59}$.

PROOF. To verify this is a functor we must show four things:

- $a\#\big(\lambda a.a.f(x_*@a)\big) \implies a\#a.f(x_*@a)$
  But $a\#a.f(x_*@a)$ always (L9.5.6). Thus, the **fresh** in Fig.5$_{59}$ is legal (C9.6.7).
- $f \in \mathbf{SetsFM}(X, Y) \implies [\mathbb{A}]f \in \mathbf{SetsFM}([\mathbb{A}]X, [\mathbb{A}]\, Y)$
  We just have to show that $a.f(x_*@a) \in [\mathbb{A}]\, Y$, i.e. that $f(x_*@a) \in Y$. But this is certainly the case if $x_*@a$ is well-defined and in X, i.e. if $a\#x_* \in [\mathbb{A}]X$. But this holds because a was chosen fresh for all variables outside the scope of $\mathbf{fresh}\, a.\, a.f(x_*@a)$, which include x_*, X.

$$\Phi_N : N_X \mapsto \lambda X.N_X \qquad \begin{array}{rcl} \texttt{nil} & \mapsto & \lambda X.1 \\ [\texttt{A}]^n X \texttt{ x } N_X & \mapsto & \lambda X.\, [\mathbb{A}]^n X \times \Phi_N(N_X)(X) \\ C \texttt{ x } N_X & \mapsto & \lambda X.\, \beta(C) \times \Phi(N_X)(X) \end{array}$$

$$\Phi_B : B_X \mapsto \lambda X.B_X \qquad \begin{array}{rcl} \texttt{nil} & \mapsto & \lambda X.\emptyset \\ (\texttt{str of } N_X) \texttt{ + } B_X & \mapsto & \lambda X.\Phi_N(N_X)(X) + \Phi_B(B_X)(X) \end{array}$$

See N10.2.1 for $\lambda X.\emptyset$ and $\lambda X.1$. See $(34)_{57}$ for the definition for β used in $\beta(C)$ above.

FIGURE 6. D10.1.6 - Functor associated to a binding signature

---

- $[\mathbb{A}]\mathbf{id}_X = \mathbf{id}_{[\mathbb{A}]X}$
  Unpack the definition of the functorial action and of **fresh** (C9.6.7).
- $[\mathbb{A}](f \circ g) = [\mathbb{A}]f \circ [\mathbb{A}]g$
  Unpack the definition of the functorial action, use the fact that И distributes over logical connectives (C9.4.5), simplify with L9.5.15.

□

We are now in a position to associate to a binding signature B_X an endofunctor on **SetsFM**.

Definition 10.1.6 (Functor associated to B_X)**.** *To B_X a binding signature (D10.1.3) associate an endofunctor $\lambda X.B_X : \mathbf{SetsFM} \to \mathbf{SetsFM}$ defined by induction on the structure of N_X and B_X as shown in Fig.6$_{60}$.*

Warning! We employ the shorthand given in N10.1.7.

Notation 10.1.7 (Shorthand)**. For Z a set we may write $B(Z)$ for $(\lambda X.B_X)(Z)$ and $N(Z)$ for $(\lambda X.N_X)(Z)$.**

The construction of $\lambda X.B_X$ means it is full of leading $1\times$ and $0+$. We shall drop them for convenience. We shall also write $[\mathbb{A}]^0(X)$ and $[\texttt{A}]^0(X)$ as X, $[\mathbb{A}]^1(X)$ as $[\mathbb{A}]X$, and $[\texttt{A}]^1X$ as $[\texttt{A}]X$.

So, to choose an *entirely random* example, the functor associated to

$$B_X = \texttt{Var3 of A + (App3 of } X \texttt{ x } X \texttt{ + (Lam3 of } [\texttt{A}]X))$$

has an action on objects given by

$$X \longmapsto 0 + (1 \times \mathbb{A} + (1 \times (X \times X) + 1 \times [\mathbb{A}]^1 X)).$$

written in shorthand as (something like)

$$X \longmapsto \mathbb{A} + X \times X + [\mathbb{A}]X.$$

The syntax captured by B_X is more general than it first appears: B_X does not allow a single binder to capture two arguments as in $[\mathbb{A}](X \times Y)$, or a choice of argument as in $[\mathbb{A}](X + Y)$. But C9.6.9 tells us that $[\mathbb{A}](-)$ distributes over $\times$ and $+$, so this should not matter. The only issue is whether out intuition agrees that binding should display this behaviour. We just consider $[\mathbb{A}](X \times Y)$. This is intended to model

> "Elements $x \in X$ and $y \in Y$ with one common distinguished bound variable".

Of course we can give this distinguished bound variable a different name in x and y if we wish using α-conversion, so this should indeed be equivalent to

> "Elements $x \in X$ with a distinguished bound variable and $y \in Y$ with a distinguished bound variable".

10.2. Initial algebras for binding signatures.

Notation 10.2.1. *Write* ***End***(**SetsFM**) *for the* ***category of endofunctors and natural transformations of*** **SetsFM** ***expressible in the language of*** FM. *Thus functors and natural transformations in* ***End***(**SetsFM**) *are assumed given by (possibly parameterised) function-classes in the language of* FM.

End(**SetsFM**) *has an initial object* ***0****, which is* $\lambda X.\emptyset$ *the constant functor taking objects to* $\emptyset$ *and arrows to the empty function* $\emptyset\colon \emptyset \to \emptyset$*. It has a terminal object* ***1*** *which is* $\lambda X.1$ *the constant functor taking objects to* $\{\emptyset\} = 1$ *and arrows to the singleton function* $\{(1,1)\} : 1 \to 1$.

So $\lambda X.B_X$ are elements of **End**(**SetsFM**). We now consider a subcategory of **End**(**SetsFM**) which has some attractive properties and then show that $\lambda X.B_X$ is in this subcategory (and enjoys its properties).

Notation 10.2.2. *We call* $F \in$ ***End***(**SetsFM**) ***pre-syntactic*** *when*

1. *F maps inclusions to inclusions and inclusion maps to inclusion maps. That is, if $X \subseteq Y$ and $\hookrightarrow_{X,Y} \in$ **SetsFM**(X, Y) denotes the inclusion map then $F(X) \subseteq F(Y)$ and $F(\hookrightarrow_{X,Y}) = \hookrightarrow_{FX,FY}$.*
2. *F preserves unions of countably ascending chains. That is, for h some function-set out of $\mathbb{N}$ such that $a \leq b$ implies $h(a) \subseteq h(b)$,*

$$F(\bigcup_{n\in\mathbb{N}} h(n)) = \bigcup_{n\in\mathbb{N}} F(h(n)).$$

Write ***PreSyn*** *for the full subcategory of* ***End***(**SetsFM**) *with objects the pre-syntactic endofunctors.*

Definition 10.2.3 (F-algebras)**.** *Recall that for a category* $\mathbf{C}$ *and an endofunctor* F *on* $\mathbf{C}$ *the* ***category of*** F***-algebras*** ***Alg***(F) *is the following category: Objects* $f, g \in \mathrm{Obj}(\boldsymbol{Alg}(F))$ *are arrows* $F(X) \xrightarrow{f} X$ *and* $F(Y) \xrightarrow{g} Y$ *for* $X, Y \in \mathrm{Obj}(\mathbf{C})$*. Arrows* $h \in \boldsymbol{Alg}(F)(f, g)$ *are arrows* $h \in \mathbf{C}(X, Y)$ *such that the following square commutes*

$$\begin{array}{ccc} F(X) & \xrightarrow{f} & X \\ {\scriptstyle F(h)}\downarrow & & \downarrow{\scriptstyle h} \\ F(Y) & \xrightarrow[g]{} & Y \end{array}$$

The standard definition of an abstract datatype described by B_X is the (underlying set of the) ***initial object*** of $\mathbf{Alg}(\lambda X.B_X)$.[23]

This motivates us to prove the following general theorem, adapted from [**60**, §2,§3]:

Theorem 10.2.4. *If* $F \in$ ***PreSyn*** *then* ***Alg***(F) *has an initial object the identity function* $\boldsymbol{id}_{\boldsymbol{init}(F)}$ *on a set* ***init***(F) *such that* $F(\boldsymbol{init}(F)) = \boldsymbol{init}(F)$*. Furthermore the set* ***init***(F) *is equal to* $\bigcup_{n\in\mathbb{N}} F^n(\emptyset)$*.*[24]

Proof. To save space we jump the gun a little and write

$$\mathbf{init}(F) \quad \text{for} \quad \bigcup_{n\in\mathbb{N}} F^n(\emptyset).$$

How do we know this really is an initial algebra and fixedpoint of F?

Property 1 guarantees that $\mathbf{init}(F)$ is a colimit of the diagram

$$\emptyset \hookrightarrow F(\emptyset) \xrightarrow{F(\hookrightarrow)} F^2(\emptyset) \xrightarrow{F^2(\hookrightarrow)} F^3(\emptyset) \ldots \tag{36}$$

with cone the trivial subset inclusion functions.

Property 2 guarantees that $\mathbf{init}(F)$ is itself a fixedpoint of F.

[23] Initial objects are unique only up to isomorphism. This is not an issue, we shall treat initial objects as 'morally unique'.

[24] The paranoid reader might worry how we know that the *class* $\bigcup_{n\in\mathbb{N}} F^n(\emptyset)$ is a *set*. F restricted to `Nat` is clearly a function-set (proof omitted), we can now apply $(\text{Replacement})_{26}$ and take $\bigcup$. Let us not worry about this issue.

Now take X and some $f\colon F(X) \to X$. Observe $\emptyset \subseteq X$ so there is an arrow, call it $\iota\colon \emptyset \hookrightarrow X$. We can use all this to build another cone over $(36)_{62}$, into X:

$$\begin{array}{ccccccccc} \emptyset & \hookrightarrow & F(\emptyset) & \xrightarrow{F(\hookrightarrow)} & F^2(\emptyset) & \xrightarrow{F^2(\hookrightarrow)} & F^3(\emptyset) & \dots \\ \downarrow{\scriptstyle \iota} & & \downarrow{\scriptstyle F(\iota)} & & \downarrow{\scriptstyle F^2(\iota)} & & \downarrow{\scriptstyle F^3(\iota)} & \\ X & \xleftarrow{f} & F(X) & \xleftarrow{F(f)} & F^2(X) & \xleftarrow{F^2(f)} & F^3(X) & \dots \end{array} \tag{37}$$

So there is a unique $\bar{f}\colon \mathbf{init}(t) \to X$ such that the colimit diagram commutes. This is equivalent to the initial algebra property:

$$\begin{array}{ccc} F(\mathbf{init}(F)) & = & \mathbf{init}(F) \\ \downarrow{\scriptstyle F(\bar{f})} & & \downarrow{\scriptstyle \exists!\bar{f}} \\ F(X) & \xrightarrow[f]{} & X \end{array}$$

□

Now we set about proving C10.2.7 ($[\mathbb{A}](-)$ is pre-syntactic).

Lemma 10.2.5 ($[\mathbb{A}](-)$ monotone)**.** $[\mathbb{A}](-)$ *satisfies N10.2.2 property 1:* $[\mathbb{A}](-)$ *is monotone and*

$$[\mathbb{A}](\hookrightarrow_{X,Y}) \;=\; \hookrightarrow_{[\mathbb{A}]X,[\mathbb{A}]Y}.$$

Proof. For the first part we must show

$$X \subseteq Y \implies [\mathbb{A}]X \subseteq [\mathbb{A}]Y.$$

We unpack the definition of $[\mathbb{A}](-)$ using the rewritten version of $(30)_{49}$. Suppose $X \subseteq Y$. We must show

$$\{x' \mid \mathit{И}a.\ \exists x \in X.\ x' = a.x\} \subseteq \{x' \mid \mathit{И}a.\ \exists x \in Y.\ x' = a.x\}$$

Since $X \subseteq Y$ we know

$$\mathit{И}a.\ \Big(\exists x \in X.\ x' = a.x \implies \exists x \in Y.\ x' = a.x\Big).$$

From C9.4.5 we know that $\mathit{И}$ distributes over implication and hence that

$$\mathit{И}a.\ \big(\exists x \in X.\ x' = a.x\big) \implies \mathit{И}a.\ \big(\exists x \in Y.\ x' = a.x\big),$$

and this gives us monotonicity.

For the second part, we unpack the functorial action on an inclusion map. Let $f\colon X \to Y$ be the inclusion map $\hookrightarrow_{X,Y}$. Then from L10.1.5,

$$[\mathbb{A}]f = \mathbf{fresh}\, a.\, \lambda x_*.(a.f(x_*@a)) \overset{L9.5.16}{=} \mathbf{fresh}\, a.\, \lambda x_*.x_*.$$

From the definition C9.6.7 we see **fresh** $a.\,\lambda x_*.x_*$ is just the inclusion $\lambda x_*.x_*$, as required. □

Corollary 10.2.6. $[\mathbb{A}](-)$ *satisfies N10.2.2 property 2: for* h *some function-set out of* $\mathbb{N}$ *such that* $a \leq b$ *implies* $h(a) \subseteq h(b)$*,*

$$[\mathbb{A}] \bigcup_{n\in\mathbb{N}} h(n) = \bigcup_{n\in\mathbb{N}} [\mathbb{A}]h(n).$$

Proof. Monotonicity (L10.2.5) gives us one half of the inclusion

$$\bigcup_{n\in\mathbb{N}} [\mathbb{A}]h(n) \subseteq [\mathbb{A}](\bigcup_{n\in\mathbb{N}} (h(n)))$$

because for each m, $h(m) \subseteq \bigcup_{n\in\mathbb{N}} h(n)$.

Now suppose $x_* \in \bigcup_{n\in\mathbb{N}} [\mathbb{A}]h(n)$. h is a function-set and has finite support so choose $a \in \mathbb{A}$ with $a\#h, x_*$. By C9.5.17 there exists an x such that $x_* = a.x$. Also, there exists an $m \in \mathbb{N}$ such that $x_* \in F(m)$. But then

$$a.x \in [\mathbb{A}]h(i) \overset{monotonicity}{\subseteq} [\mathbb{A}] \bigcup_{n\in\mathbb{N}} h(n).$$

and this gives us the reverse inclusion

$$[\mathbb{A}] \bigcup_{n\in\mathbb{N}} h(n) \subseteq \bigcup_{n\in\mathbb{N}} [\mathbb{A}]h(n).$$

□

Corollary 10.2.7. $[\mathbb{A}](-)$ *is pre-syntactic.*

Proof. From L10.2.5 and C10.2.6, which show that $[\mathbb{A}]^1(-)$ satisfies the two conditions of N10.2.2. □

We conclude this subsection by showing that for a binding signature B_X (D10.1.3), the endofunctor associated to it $\lambda X.B_X$ (D10.1.6) is pre-syntactic (N10.2.2).

Lemma 10.2.8. *For all sets* C*, the constant functors* $\lambda X.C$ *that take objects to* C *and arrows to the identity on* C *are in* ***PreSyn****.*

Proof. We verify the conditions of N10.2.2 individually. □

Lemma 10.2.9. ***PreSyn*** *has finite products and finite sums.*

Proof. For the first part it suffices to verify that the initial object $\mathbf{0} = \lambda x.\emptyset$ is pre-syntactic, and that for $F, G \in$ **PreSyn** their product in **End**(**SetsFM**) (which is $\lambda X, Y.FX \times FY$) is pre-syntactic. Proof omitted.

For the second part it suffices to verify that the terminal object $\mathbf{1} = \lambda x.1$ (N10.2.1) is pre-syntactic, and that for $F, G \in$ **PreSyn**, their sum in **End**(**SetsFM**) (which is $\lambda X, Y.FX + FY$) is pre-syntactic. Proof omitted. □

Lemma 10.2.10. ***PreSyn*** *is closed under functor-composition: if* $F, G \in$ ***PreSyn*** *then* $F \circ G \in$ ***PreSyn****.*

Proof. We inspect the two conditions of N10.2.2 individually and see they are preserved by functor-composition. □

Lemma 10.2.11. $[\mathbb{A}]^n(-) \in$ ***PreSyn****.*

Proof. A corollary of C10.2.7 and L10.2.10. □

Corollary 10.2.12. *For any binding signature* B_X *(D10.1.3), the associated endofunctor* $\lambda X.B(X)$ *(D10.1.6) is pre-syntactic (N10.2.2).*

Proof. $\lambda X.B_X$ is inductively generated using operations under which, as we have just proved, **PreSyn** is closed. □

Definition 10.2.13 (Datatypes)**.** *C10.2.12 and T10.2.4 imply* $\lambda X.B_X$ *has an initial object for all* B_X *(D10.1.3). Since the initial object we construct is the identity function on an underlying set we identify the initial object* with *its underlying set (and shall not distinguish the two from now on), and write it*

$$\mu X.B_X \quad \textit{or just} \quad \boldsymbol{B}. \tag{38}$$

Recall from T10.2.4 that

$$\mathbf{B} = \bigcup_{n\in\mathbb{N}} B^n(\emptyset). \tag{39}$$

(Here we use the shorthand of N10.1.7 and write $B^n(\emptyset)$ for $(\lambda X.B_X)^n(\emptyset)$.)

We shall briefly work through a small example of a set built using a particular binding signature in §10.3 below. In D10.3.2 and R10.3.3 we shall consider what a typical element of some **B** looks like as a set.

10.3. Iteration. We shall use the notation of N10.1.7 and D10.2.13 mostly without comment.

The universal property of $\mathbf{B} = \mu X.B_X$ gives rise directly (and in a standard way) to an iterative scheme for defining functions out of it. Suppose

$$B_X = N_X^1 + \ldots + N_X^k.$$

Then a function $f\colon B(Z) \to Z$ is equivalent to k functions

$$f_i\colon N^i(Z) \to Z \qquad i = 1, \ldots, k$$

and by initiality of $\mathbf{B}$, for each such k-tuple there exists a unique $\bar{f}\colon \mathbf{B} \to Z$ such that the following diagram commutes:

$$\begin{array}{ccc} B(\mathbf{B}) & = & \mathbf{B} \\ {\scriptstyle B(\bar{f})}\downarrow & & \downarrow{\scriptstyle \bar{f}} \\ B(Z) & \xrightarrow[f_1+\ldots+f_k]{} & Z \end{array} \qquad \text{i.e.} \qquad \bar{f} \;=\; (f_1 + \ldots + f_k) \circ B(\bar{f}). \tag{40}$$

The nonstandard part is that the N^i_X may now contain $[\mathbb{A}]^n X$. T9.6.6 and C9.6.7 characterise all functions out of $[\mathbb{A}]X$ but in practice—as in the latter half of this subsection—we find ourselves mostly concerned with 'normal' functions $f\colon X \to Y$ lifted using the using the functorial action of $[\mathbb{A}](-)$ (L10.1.5) to $[\mathbb{A}]f\colon [\mathbb{A}]X \to [\mathbb{A}]\,Y$.

Notation 10.3.1 (Injections)**.** *A typical B_X is $B_X = \mathtt{str}_1$ of N^1_X + ... + $\mathtt{str}_k$ of N^k_X. We write the injection functions from $N^i(\boldsymbol{B})$ to $\boldsymbol{B}$ as*

$$\boldsymbol{str}_i\colon N^i(\boldsymbol{B}) \longrightarrow \boldsymbol{B}.$$

Write the injection into the i-th component of a k-fold disjoint sum

$$\boldsymbol{In}_i \stackrel{\text{def}}{=} \boldsymbol{Inl}(\boldsymbol{Inr}^{i-1})\colon X_i \;\hookrightarrow\; X_1 + (X_2 + (\ldots + X_n)\ldots).$$

So in fact $\boldsymbol{str}_i = \boldsymbol{In}_i$ and (using N10.1.7)

$$B(X) = \boldsymbol{In}_1(N^1(X)) + \ldots + \boldsymbol{In}_k(N^k(X)) \tag{41}$$

For those who appreciate maths you can see, touch and taste, we combine $(39)_{65}$ and $(41)_{66}$ to characterise the t such that there exists a binding signature B_X with $t \in \mathbf{B}$:

Definition 10.3.2 (Syntactic Sets)**.** *Let the set of* ***pre-syntactic sets PreStx*** *be generated by the following inductive rules (expressed in BNF grammar style, but this is sets,* not *syntax):*

$$(\mathbf{PreStx}) \qquad ps ::= c \mid \boldsymbol{In}_i(ps) \mid (ps, ps) \mid a.ps \qquad i \in \mathbb{N},\; c \in \bigcup \boldsymbol{BType}$$

We define the class of ***syntactic sets Stx*** *(a subset of* ***PreStx****) in a similar style as follows:*

$$\begin{aligned} s &::= \boldsymbol{In}_i(f) && i \in \mathbb{N} \\ f &::= (t, f) \mid (c, f) && c \in \bigcup \boldsymbol{BType} \\ t &::= \vec{a}.s \end{aligned} \qquad (\mathbf{Stx})$$

(The notation $\mathbf{In}_i$ is introduced in N10.3.1. **BType** is defined in D10.1.2. Nested abstractions $\vec{a}.s$ are defined in N10.5.1.)

Remark 10.3.3 (Typical $t \in \mathbf{B}$)**.** An informal reading of D10.3.2 is that $t \in \mathbf{B}$ is of the form

$$\mathbf{In}_i(t') = \mathbf{Inl}(\mathbf{Inr}^n(t'))$$

where $t' \in N^i(\mathbf{B})$ is of the form

$$(\vec{a}_1.s_1, c, a, (\vec{a}_2.s_2, (\dots \vec{a}_m.s_m)\dots)$$

for s_i itself of the form of t. Here c is some constant in a type $T \in \mathbf{BType}$ and a is a similar constant for the special case of $\mathbb{A}$. $\diamond$

We could now try defining some general algorithm for manufacturing iterative schemes out of a B_X. I choose not to. I propose it would be more enlightening to work through a small example, so we devote the rest of the subsection to working out the iterative scheme for the untyped λ-calculus.

Recall the binding signature for the λ-calculus up to α-equivalence given on $p.57$:

Definition 10.3.4 (λ-terms)**.**

(42) $L_X \stackrel{\text{def}}{=} \texttt{Var}\ \textit{of}\ \texttt{A + App}\ \textit{of}\ X\ \texttt{x}\ X\ \texttt{+ Lam}\ \textit{of}\ [\texttt{A}]X \quad \textit{so in this case}$

$$\boldsymbol{Var}(\boldsymbol{L}) = \boldsymbol{In}_1(\mathbb{A}),\ \ \boldsymbol{App}(\boldsymbol{L}) = \boldsymbol{In}_2(\boldsymbol{L} \times \boldsymbol{L}),\ \ \textit{and}\ \ \boldsymbol{Lam}(\boldsymbol{L}) = \boldsymbol{In}_3([\mathbb{A}]\boldsymbol{L}).$$

This implements in FM the datatype `lam2` declared informally in D4.8. So $\mathbf{L}$ is the set inductively constructed by the three rules

$$(43) \qquad \frac{a \in \mathbb{A}}{\mathbf{Var}(a) \in \mathbf{L}} \quad \frac{x_1 \in \mathbf{L} \quad x_2 \in \mathbf{L}}{\mathbf{App}(x_1, x_2) \in \mathbf{L}} \quad \frac{x_* \in [\mathbb{A}]\mathbf{L}}{\mathbf{Lam}(x_*) \in \mathbf{L}}.$$

These visibly correspond to the three rules of $(5)_{17}$ and as promised terms 'quotient' themselves by α-equivalence because of the binding-like behaviour of the abstraction-set.

To obtain the iterative scheme for $\mathbf{L}$ we consult $(40)_{66}$ and see it splits into the following three diagrams:

$$\begin{array}{ccc}
\mathbb{A} & \xrightarrow{\mathbf{Var}} & \mathbf{L} \\
{\scriptstyle \mathbf{id}}\downarrow & & \downarrow{\scriptstyle \bar{S}} \\
\mathbb{A} & \xrightarrow[f_{\mathbf{Var}}]{} & \mathbf{Z}
\end{array}
\qquad
\begin{array}{ccc}
\mathbf{L} \times \mathbf{L} & \xrightarrow{\mathbf{App}} & \mathbf{L} \\
{\scriptstyle \bar{f} \times \bar{f}}\downarrow & & \downarrow{\scriptstyle \bar{f}} \\
\mathbf{Z} \times \mathbf{Z} & \xrightarrow[f_{\mathbf{App}}]{} & \mathbf{Z}
\end{array}
\qquad
\begin{array}{ccc}
[\mathbb{A}]\mathbf{L} & \xrightarrow{\mathbf{Lam}} & \mathbf{L} \\
{\scriptstyle [\mathbb{A}]\bar{f}}\downarrow & & \downarrow{\scriptstyle \bar{f}} \\
[\mathbb{A}]\mathbf{Z} & \xrightarrow[f_{\mathbf{Lam}}]{} & \mathbf{Z}
\end{array}$$

which together express the following predicate in the language of FM:

$$\begin{aligned}
&\forall Z.\ \forall f_{\mathbf{Var}}\colon \mathbb{A} \to Z,\ f_{\mathbf{App}}\colon Z \times Z,\ f_{\mathbf{Lam}}\colon [\mathbb{A}]Z \to Z.\\
&\quad \exists! \bar{f}\colon \mathbf{L} \to Z.\\
&\qquad \forall x \in \mathbb{A}.\ \bar{f}(\mathbf{Var}(x)) = f_{\mathbf{Var}}(x) \wedge\\
&\qquad \forall t_1, t_2 \in \mathbf{L}.\ \bar{f}(\mathbf{App}(t_1, t_2)) = f_{\mathbf{App}}(\bar{f}(t_1), \bar{f}(t_2)) \wedge\\
&\qquad \forall t_* \in [\mathbb{A}]\mathbf{L}.\ \bar{f}(\mathbf{Lam}(t_*)) = f_{\mathbf{Lam}}(\mathbf{fresh}\, n.\, n.\bar{f}(t_*@n))
\end{aligned} \tag{44}$$

The final clause is direct from the functorial action of $[\mathbb{A}](-)$ in L10.1.5. There is one more manipulation we could carry out if we wished. We could apply $(32)_{52}$ to rewrite the last clause and obtain

$$\begin{aligned}
&\forall Z.\ \forall f_{\mathbf{Var}}\colon \mathbb{A} \to Z,\ f_{\mathbf{App}}\colon Z \times Z,\ f_{\mathbf{Lam}}\colon [\mathbb{A}]Z \to Z.\\
&\quad \exists! \bar{f}\colon \mathbf{L} \to Z.\\
&\qquad \forall x \in \mathbb{A}.\ \bar{f}(\mathbf{Var}(x)) = f_{\mathbf{Var}}(x) \wedge\\
&\qquad \forall t_1, t_2 \in \mathbf{L}.\ \bar{f}(\mathbf{App}(t_1, t_2)) = f_{\mathbf{App}}(\bar{f}(t_1), \bar{f}(t_2)) \wedge\\
&\qquad \forall t_* \in [\mathbb{A}]\mathbf{L}.\ И n.\ \bar{f}(\mathbf{Lam}(t_*)) = f_{\mathbf{Lam}}(n.\bar{f}(t_*@n)).
\end{aligned} \tag{45}$$

After D4.8 we declared a function `subst2` on `lam2`. We can now iteratively define a substitution function **subst** on **L**. For $t \in \mathbf{L}$, $a \in \mathbb{A}$ let $\mathbf{subst}(t, a) \stackrel{\text{def}}{=} \bar{S}$ where $S\colon L(\mathbf{L}) \to \mathbf{L}$ is defined by the three functions

(46)

$$S_{\mathbf{Var}}\colon \mathbb{A} \to \mathbf{L} \qquad S_{\mathbf{App}}\colon \mathbf{L} \times \mathbf{L} \to \mathbf{L} \qquad S_{\mathbf{Lam}}\colon [\mathbb{A}]\mathbf{L} \to \mathbf{L}$$

$$b \mapsto \begin{cases} \mathbf{Var}(b) & b \neq a \\ t & b = a \end{cases} \qquad s_1, s_2 \mapsto \mathbf{App}(s_1, s_2) \qquad s_* \mapsto \mathbf{Lam}(s_*).$$

So $S_{\mathbf{App}} = \mathbf{App}$ and $S_{\mathbf{Lam}} = \mathbf{Lam}$, but $S_{\mathbf{Var}}$ is something new. By its definition $\bar{S}$ fits in a diagram like $(40)_{66}$, so $S_{\mathbf{Var}}$, $S_{\mathbf{App}}$ and $S_{\mathbf{Lam}}$ make the following diagrams commute:

$$\begin{array}{ccc}
\mathbb{A} & \xrightarrow{\mathbf{Var}} & \mathbf{L} \\
{\scriptstyle \mathbf{id}}\downarrow & & \downarrow{\scriptstyle \bar{S}} \\
\mathbb{A} & \xrightarrow[S_{\mathbf{Var}}]{} & \mathbf{L}
\end{array}
\qquad
\begin{array}{ccc}
\mathbf{L} \times \mathbf{L} & \xrightarrow{\mathbf{App}} & \mathbf{L} \\
{\scriptstyle \bar{S} \times \bar{S}}\downarrow & & \downarrow{\scriptstyle \bar{S}} \\
\mathbf{L} \times \mathbf{L} & \xrightarrow[\mathbf{App}]{} & \mathbf{L}
\end{array}
\qquad
\begin{array}{ccc}
[\mathbb{A}]\mathbf{L} & \xrightarrow{\mathbf{Lam}} & \mathbf{L} \\
{\scriptstyle [\mathbb{A}]\bar{S}}\downarrow & & \downarrow{\scriptstyle \bar{S}} \\
[\mathbb{A}]\mathbf{L} & \xrightarrow[\mathbf{Lam}]{} & \mathbf{L}
\end{array}$$

Accordingly, or just using the iterative scheme $(44)_{68}$ directly, we deduce that $\bar{S}$ satisfies the following iterative equations:

$$\bar{S}(\mathbf{Var}(a)) = \begin{cases} \mathbf{Var}(b) & b \neq a \\ t & b = a \end{cases}, \quad \bar{S}(\mathbf{App}(s_1, s_2)) = \mathbf{App}(\bar{S}s_1, \bar{S}s_2) \quad \text{and} \tag{47}$$

$$\bar{S}(\mathbf{Lam}(s_*)) = \mathbf{Lam}(\mathbf{fresh}\, b.\, (b.\bar{S}(s_*@b))) \quad \text{or using } (32)_{52} \tag{48}$$

$$\mathit{И} b.\, \Big(\bar{S}(\mathbf{Lam}(s_*)) = \mathbf{Lam}(b.\bar{S}(s_*@b)) \Big) \tag{49}$$

Since b is chosen fresh it does not occur in t and is not equal to a: the rule for **Lam** is just the rule for substitution

$$\mathbf{subst}(t, a, \mathbf{Lam}(b.s)) = \mathbf{Lam}(b.(\mathbf{subst}(t, a, s)))$$

from D4.11.

10.4. Induction. Now let us derive a principle of logical induction for **L**, the datatype constructed in §10.3 above. We continue the notation of that subsection and in particular the commuting diagram $(40)_{66}$. Consider the special case $Z = \mathbb{B}$ where $\mathbb{B}$ is the (any) two-element set of truth values $\{\top, \bot\} \in \mathcal{V}_{\mathrm{FM}}$ in the FM universe, say $\top = \{\emptyset\}$ and $\bot = \emptyset$. Note that $\top$ and $\bot$ are equivariant; $a\#\top, \bot$ for all $a \in \mathbb{A}$ (N9.2.8).

We define $f\colon B(\mathbb{B}) = \mathbb{A} + \mathbb{B} \times \mathbb{B} + [\mathbb{A}]\mathbb{B} \longrightarrow \mathbb{B}$ as follows:

$$\begin{aligned} f_{\mathbf{Atm}}(x) &= \top \\ f_{\mathbf{App}}(b_1, b_2) &= b_1 \wedge b_2 \\ f_{\mathbf{Lam}}(b_*) &= \mathbf{fresh}\, a.\, b_*@a \end{aligned} \tag{50}$$

Since $b_*@a \in \mathbb{B}$ and $n\#\top, \bot$ for all $n \in \mathbb{A}$ we know this is a legal use of **fresh** (C9.6.7).

Direct from $(44)_{68}$ we obtain

$$\begin{aligned} &\exists!\bar{f}\colon \mathbf{L} \to \mathbb{B}. \\ &\quad \forall x \in \mathbb{A}.\, \bar{f}(\mathbf{Var}(x)) = f_{\mathbf{Var}}(x) = \top \wedge \\ &\quad \forall t_1, t_2 \in \mathbf{L}.\, \bar{f}(\mathbf{App}(t_1, t_2)) = \big(\bar{f}(t_1) \wedge \bar{f}(t_2)\big) \wedge \\ &\quad \forall t_* \in [\mathbb{A}]\mathbf{L}.\, \bar{f}(\mathbf{Lam}(t_*)) = \mathbf{fresh}\, a.\, \big(\mathbf{fresh}\, n.\, n.\bar{f}(t_*@n)\big)@a \end{aligned} \tag{51}$$

This last clause looks complicated but it simplifies. I shall work through the details of how, though I fear that by doing so I risk making something easy look hard. The main flow of discussion resumes at $(53)_{70}$ (and culminates in $(55)_{71}$).

We can apply L9.6.10 to the last clause of $(51)_{69}$.

$$\mathbf{fresh}\, a.\, \big(\mathbf{fresh}\, n.\, n.\bar{f}(t_*@n)\big)@a = \mathbf{fresh}\, a.\, \mathbf{fresh}\, n.\, (n.\bar{f}(t_*@n))@a, \tag{52}$$

and use L9.5.15.

$$\textbf{fresh}\, a.\, \textbf{fresh}\, n.\, (n.\bar{f}(t_*@n))@a = \textbf{fresh}\, a.\, \textbf{fresh}\, n.\, (((a\ n)\cdot\bar{f})(((a\ n)\cdot t_*)@a))$$

But $\bar{f}, t_*$ are free under the scope of the **fresh** so we may assume (technically by $(32)_{52}$) that $a, n\#\bar{f}, t_*$. By L9.2.7 we know

$$\textbf{fresh}\, a.\, \textbf{fresh}\, n.\, (((a\ n)\cdot\bar{f})(((a\ n)\cdot t_*)@a)) = \textbf{fresh}\, a.\, \textbf{fresh}\, n.\, (\bar{f}(t_*@a)).$$

the **fresh** $n.$ is dummy (use $(32)_{52}$) and we have

$$\textbf{fresh}\, a.\, \big(\textbf{fresh}\, n.\, n.\bar{f}(t_*@n)\big)@a = \textbf{fresh}\, a.\, \bar{f}(t_*@a).$$

We can now rewrite $(51)_{69}$ as follows:

$$\begin{array}{rll} & \exists!\bar{f}\colon \mathbf{L}\to\mathbb{B}.\ \Phi(f) & \\ \Phi(\bar{f}) \stackrel{\text{def}}{=} & \forall x\in\mathbb{A}.\ \bar{f}(\mathbf{Var}(x)) = \top \wedge & \\ & \forall t_1, t_2\in\mathbf{L}.\ \bar{f}(\mathbf{App}(t_1,t_2)) = \big(\bar{f}(t_1)\wedge\bar{f}(t_2)\big) \wedge & \\ & \forall t_*\in[\mathbb{A}]\mathbf{L}.\ \bar{f}(\mathbf{Lam}(t_*)) = \textbf{fresh}\, a.\, \bar{f}(t_*@a) & \end{array} \tag{53}$$

Observe that $\Phi(\lambda x.\top)$ so by uniqueness of $\bar{f}$ above,

$$\bar{f} = \lambda x.\top. \tag{54}$$

This observation is the core of an inductive principle for **L**. Now for a little more theory.

Lemma 10.4.1 (**fresh** and И)**.** *For $\phi\colon X\to\mathbb{B}$ a predicate-set in $\mathcal{V}_{\mathrm{FM}}$,*

$$(\textbf{\textit{fresh}}\, a.\, \phi(a)) = \top \iff И a.\ (\phi(a) = \top)$$

Proof. Since $a\#\top, \bot$ the LHS is a legal use of **fresh** and well-defined. The proof itself is by $(32)_{52}$. □

We anticipated the following notation when we wrote "$b_1\wedge b_2$" in $(50)_{69}$ and "$\bar{f}(t_1)\wedge\bar{f}(t_2)$" in $(53)_{70}$.

Notation 10.4.2. *For $\phi\colon X\to\mathbb{B}$, $\phi(x)$ is not a predicate but a set—one of $\top$ or $\bot$—in $\mathcal{V}_{\mathrm{FM}}$. Nevertheless we may abuse notation and write '$\phi(x)$', read "$\phi(x)$* ***holds****", to mean the predicate "$\phi(x)=\top$" and similarly '$\neg\phi(x)$' for the predicate "$\phi(x)=\bot$".*

In this notation the statement of L10.4.1 above becomes

$$\textbf{fresh}\, a.\, \phi(a) \iff И a.\ \phi(a).$$

and we can combine $(53)_{70}$ and $(54)_{70}$ to see that $(51)_{69}$ gives a normal-looking inductive principle:

$$\begin{array}{ll} & \forall \phi\colon \mathbf{L} \to \mathbb{B}. \\ & \Big(\forall x \in \mathbb{A}.\ \phi(\mathbf{Var}(x)) \quad \wedge \\ (55) & \forall t_1, t_2 \in \mathbf{L}.\ \phi(t_1) \wedge \phi(t_2) \Rightarrow \phi(\mathbf{App}(t_1, t_2)) \quad \wedge \\ & \forall t_* \in [\mathbb{A}]\mathbf{L}.\ \mathit{И} a.\ \phi(t_* @ a) \Rightarrow \phi(\mathbf{Lam}(t_*))\Big) \qquad \Longrightarrow\ \forall x \in \mathbf{L}.\ \phi(x) \end{array}$$

10.5. Adequacy. Recall the examples of binding signatures on *p.*57. As promised they now have associated initial algebras. In the first two cases they rigorously construct a set of syntax for datatypes declared in our informal programming language in §4. Recall that the boldface notation **B** for initial algebras of B_X was introduced in D10.2.13.

- The untyped λ-calculus not up to α-equivalence.

  $$M_X \stackrel{\text{def}}{=} \texttt{Var of A + App of } X \texttt{ x } X \texttt{ + Lam of A x } X.$$

  The set **M** is **lam3**, the set of syntax for the datatype `lam3` (see D8.2.3 and D8.2.2 respectively).
- The untyped λ-calculus up to α-equivalence.

  $$L_X \stackrel{\text{def}}{=} \texttt{Var of A + App of } X \texttt{ x } X \texttt{ + Lam of [A]} X.$$

  The set **L** is the set of syntax for `lam2` (see D4.8).
- Terms of a simple π-calculus up to α-equivalence.

$$\begin{aligned} (56)\quad P_X \stackrel{\text{def}}{=}\ & \texttt{Zero of U + Out of A x A x } X \texttt{ + In of A x [A]} X \texttt{ +} \\ & \texttt{Silent of } X \texttt{ + Nu of [A]} X \texttt{ + Par of } X \texttt{ x } X \texttt{ + Sum of } X \texttt{ x } X \texttt{ +} \\ & \texttt{Match of A x A x } X \texttt{ + Mismatch of A x A x } X. \end{aligned}$$

This may look quite good, but the question is

> "How do we know the sets **M,L,P** really *are* what we said they are, i.e. syntax possibly up to α-equivalence?"

We need an ***adequacy result*** relating sets of syntax built 'naïvely' with $\mathbb{A} \times (-)$ to those built using $[\mathbb{A}](-)$.

Notation 10.5.1 (Shorthand)**.** *Let $\vec{a} = (a_1, \ldots, a_n)$ be a list of atoms of length n and $x \in X$. We continue N10.1.7 and write $\vec{a}.x$ for*

$$a_1.(\ldots(a_n.x)\ldots) \in [\mathbb{A}]^n X,$$

and $x @ \vec{a}$ for

$$(\ldots(x @ a_1)\ldots @ a_n).$$

We also write

$$\mathsf{И}\vec{a}.\ \phi(\vec{a},\vec{x}) \quad \textit{for} \quad \mathsf{И}a_1.\ \mathsf{И}a_2.\ \ldots \mathsf{И}a_n.\ \phi(\vec{a},\vec{x}),$$

and

$$\boldsymbol{fresh}\,\vec{a}.f \quad \textit{for} \quad \boldsymbol{fresh}\,a_1.\,\boldsymbol{fresh}\,a_2.\ \ldots \boldsymbol{fresh}\,a_n.f.$$

For the compound И *and* ***fresh*** *quantifiers the ordering of* $\vec{a}$ *does not logically matter and the* a_i *may be assumed distinct (proof omitted). It may occur that* $\vec{a}$ *is the empty list. In that case* $\vec{a}.x$ *denotes* x*, and similarly for* И *and* ***fresh****.*

Definition 10.5.2 (Naïve signatures)**.** *We relate to a binding signature* B_X *(D10.1.3) a* ***naïve signature*** B'_X *obtained from* B_X *by "replacing every* $[\mathtt{A}]^n(-)$ *by* $\mathtt{A}^n\ \mathtt{x}\ (-)$*" (N10.1.4). The naïve signature gives rise to a* ***naïve datatype*** $\mu X.B'_X$ *(or just* $\boldsymbol{B'}$*, see* $(38)_{65}$*).*

For example for M_X, L_X as above, $M_X = L'_X$. A rigorous inductive definition of the map $B_X \mapsto B'_X$ is easy but I hope not needed.

Definition 10.5.3 (Canonical $\alpha(B_X)$)**.** *Associated to binding signatures* B_X *we build a canonical function*

$$\alpha(B_X)\colon Z \in \mathrm{Obj}(\mathbf{SetsFM}) \longmapsto \alpha(B_X)(Z)\colon B'(Z) \to B(Z),$$

by induction on N_X *and* B_X *as shown in Fig.*γ_{73}*.*[25]

Lemma 10.5.4. $\alpha(B_X)(Z)$ *as constructed in Fig.*γ_{73} *really is a function* $B'(Z) \to B(Z)$*, and* $\alpha(N_X)(Z)$ *really is a function from* $N'(Z) \to N(Z)$*.*

Proof. By induction on the syntax of B_X and N_X, details omitted. □

Corollary 10.5.5. *In particular*

$$\alpha(B_X)(\boldsymbol{B})\colon B'(\boldsymbol{B}) \longrightarrow B(\boldsymbol{B}) = \boldsymbol{B},$$

so $\alpha(B_X)(\boldsymbol{B})$ *is a* B'_X*-algebra (D10.2.3).*

Now fix B_X **and write**

$$\alpha \stackrel{\mathrm{def}}{=} \alpha(B_X)(\mathbf{B}).$$

We now show that $\bar{\alpha}$ (§10.3) generates an equivalence relation on the naïve datatype that is recognizable as α-equivalence. The only real difficulty is not $\bar{\alpha}$ itself but the fact that there is no conveniently available general account of α-equivalence with which to compare it, besides itself!

[25] Recall that Obj(**SetsFM**) is the FM sets $\mathcal{V}_{\mathrm{FM}}$ and $\mathbf{SetsFM}(B'(Z), B(Z))$ is the collection of function-sets from $B'(Z)$ to $B(Z)$.

$$
\begin{array}{ll}
\alpha(\texttt{nil})(Z) & = \lambda x \in 1.* \\
 & \quad : \mathbb{U} \to \mathbb{U} \\
\alpha([\texttt{A}]^n X\ \texttt{x}\ N_X)(Z) & = (\lambda \vec{a}, x \in Z.\vec{a}.x) \times \alpha(N_X)(Z) \\
 & \quad : (\mathbb{A}^n \times X) \times \mathbf{Dom}(\alpha(N_X)(Z)) \to \\
 & \qquad\qquad [\mathbb{A}]^n X \times \mathbf{Rng}(\alpha(N_X)(Z)) \\
\alpha(C\ \texttt{x}\ N_X)(Z) & = (\lambda x \in C.x) \times \alpha(N_X)(Z) \\
 & \quad : C \times \mathbf{Dom}(\alpha(N_X)(Z)) \to C \times \mathbf{Rng}(\alpha(N_X)(Z)) \\
\alpha(\texttt{nil})(Z) & = \lambda x \in \emptyset.\emptyset \\
 & \quad : \emptyset \to \emptyset \\
\alpha((\texttt{str of}\ N_X)\ \texttt{+}\ B_X) & = \alpha(N_X)(Z) + \alpha(B_X)(Z) \\
 & \quad : \mathbf{Dom}(\alpha(N_X)(Z)) \times \mathbf{Dom}(\alpha(B_X)(Z)) \to \\
 & \qquad\qquad \mathbf{Rng}(\alpha(N_X)(Z)) \times \mathbf{Rng}(\alpha(B_X)(Z))
\end{array}
$$

The action of $\times$ and $+$ on functions is described in Fig.5_{59}.

FIGURE 7. D10.5.3 - Scheme for canonical α out of naïve datatype

By the initial algebra property of the naïve datatype $\mathbf{B}'$ ('naïve datatype' D10.5.2, 'initial algebra' D10.2.13), there is a unique $\bar{\alpha}\colon \mathbf{B}' \to \mathbf{B}$ such that the following diagram commutes:

(57)
$$
\begin{array}{ccc}
B'(\mathbf{B}') & = & \mathbf{B}' \\
{\scriptstyle B'(\bar{\alpha})}\downarrow & & \downarrow{\scriptstyle \bar{\alpha}} \\
B'(\mathbf{B}) & \xrightarrow[\alpha]{} & \mathbf{B}
\end{array}
$$

Notation 10.5.6 ($\mathbf{Ker}(f)$)**.** *For $f\colon A \to B$ a set-function recall that the **kernel of** f written **Ker**(f), is an equivalence class on A. As a relation we write it $\approx_f$ and define it by*

$$a \approx_f b \quad \stackrel{\text{def}}{\Leftrightarrow} \quad f(a) = f(b)$$

From the commuting square $(57)_{73}$ above, $\mathbf{Ker}(\bar{\alpha})$ is the ***smallest congruence on $\mathbf{B}'$ containing Ker***(α). So what is $\mathbf{Ker}(\alpha)$? It is the identity except on $N(X)$ for $N_X = [\texttt{A}]^n X$ a clause in in B_X. There $N'_X = \texttt{A}^n\ \texttt{x}\ X$ and on

$N'(X) = \mathbb{A}^n \times X$,

$$(\vec{a}, x) \approx_\alpha (\vec{b}, y) \iff \vec{a}.x = \vec{b}.y. \tag{58}$$

Lemma 10.5.7. *Let B_X be a binding signature and $\alpha = \alpha(B_X)(\boldsymbol{B})$ as above (D10.5.3). As discussed $\boldsymbol{Ker}(\bar{\alpha})$ is the smallest congruence containing $\approx_\alpha$ (see $(58)_{74}$). Then we claim this is precisely what we would normally call α-equivalence on $\boldsymbol{B}$.*

Sketch proof. Unpack abstractions $\vec{a}.x$ as equivalence-classes (D9.5.1) and observe that $\vec{a}.x = \vec{b}.y$ precisely when for $\mathsf{M}\,\vec{c}$ (see N10.5.1),

$$(\vec{c}\ \vec{a})\cdot x = (\vec{c}\ \vec{b})\cdot y$$

(for an appropriate definition of $(\vec{x}\ \vec{y})$ as $(x_1\ y_1)\cdot\ldots\cdot(x_n\ y_n)$, given that all elements occuring in both lists are pairwise distinct). Then observe that name-permutation and name-substitution for c a *new* name coincide for syntax, and hence that this condition is precisely

$$[\vec{c}/\vec{a}]x \ = \ [\vec{c}/\vec{b}]y.$$

This brings us back to a more traditional construction of α-equivalence. □

We have seen this before in T8.2.5 in the special case of naïve binding signature for the untyped λ-calculus. We have the following result:

Theorem 10.5.8 (Adequacy of FM syntax)**.** *For any binding signature B_X and corresponding naïve signature B'_X (D10.5.2), the datatypes $\boldsymbol{B}'$ and $\boldsymbol{B}$ are related by*

$$\boldsymbol{B}'/\approx_{\bar{\alpha}} \ \cong \ \boldsymbol{B}$$

where $\approx_{\bar{\alpha}}$ is α-equivalence on the naïve datatype of terms.

Proof. From the results above. □

We have now given a "general account of α-equivalence". This means

> "Binding signatures B_X are a framework within which we can declare and construct a nontrivial class of syntax in FM. Given a B_X we can automatically generates α-equivalence on the corresponding naïve datatype B'_X for free."

10.6. Taking Stock. Now α-equivalence is no mystery to me or to the reader. Given a specific grammar we know precisely what it should be on the datatype of naïve terms (D10.5.2), and we can define it quite rigorously—on a case-by-case basis. §10.5 presented a general recipe (based on FM) which could be used, say, as the underlying theory of an automated procedure for constructing datatypes up to α-equivalence. I know of at least two contexts where we would want to do this:

1. Design and implementation of programming languages intended for manipulating syntax.
2. Theorem-proving environments (such as Isabelle, Chapter III).

We can program general procedures and operations involving α-equivalence once-and-for-all to be instantiated as the user declares new datatypes. Similarly, we can prove theorems about α-equivalence once-and-for-all to be instantiated for particular binding signatures.

I know Isabelle best, so let us imagine an extended datatypes package (see [**59**, §2.8 p47]) that takes a binding signature and returns, say,

1. The datatype (D10.2.13).
2. An [iterative/recursive] [proof/function] scheme on it (§10.3).
3. A corresponding naïve datatype with an associated "α-equivalence" map onto the binding datatype plus useful theorems and results—perhaps to model pretty-printing and parsing (T10.5.8).

We can easily imagine a corresponding package for a programming language.

This is not earth-shattering. The first two points on the list above are already ubiquitous among theorem-proving environments and virtually universal in programming languages. To my knowledge the third item is not usually provided (it certainly does not exist in Isabelle)—but there is no reason it should not be. The datatypes-up-to-binding would have to be implemented in ZF, perhaps using de Bruijn (§33.1), but it is feasible.

FM scores in this third point because it has the $\mathcal{N}$-quantifier and the **fresh**-term former (D9.4.2 and C9.6.7), which enable us to *express* these programs and theorems in an *easy* and *natural* manner.[26]

[26]Cf. `oNat` on *p.*13. Examples permeate the text. For informal examples see D4.8 and D4.11. The mathematics leading up to T10.5.8 is clean and slick, in ZF that would be absolutely impossible. See also §22.2. For a moment of particular drama see R23.1.13.

$$
\begin{array}{ll}
\phi(\texttt{nil}) & = \lambda x \in 1.\emptyset \\
 & \quad : \mathbb{U} \to pow_{\text{fin}}(\mathbb{A}) \\
\phi([\texttt{A}]^n X\ \texttt{x}\ N_X) & = \lambda x \in [\mathbb{A}]^n pow_{\text{fin}}(\mathbb{A}), y \in \mathbf{Dom}(\phi(N_X)). \\
 & \quad \big(\mathbf{fresh}\ \vec{a}.\ (x@\vec{a}) \setminus \vec{a}\big) \cup \phi(N_X)(y) \\
 & \quad : [\mathbb{A}]^n pow_{\text{fin}}(\mathbb{A}) \times \mathbf{Dom}(\phi(N_X)) \to pow_{\text{fin}}(\mathbb{A}) \\
\phi(C\ \texttt{x}\ N_X) & = \lambda x \in \beta(C), y \in \mathbf{Dom}(\phi(N_X)).\mathbf{Supp}(x) \cup \phi(N_X)(y) \\
 & \quad : \beta(C) \times \mathbf{Dom}(\phi(N_X)) \to pow_{\text{fin}}(\mathbb{A}) \\
\phi(\texttt{nil}) & = \lambda x \in \emptyset.\emptyset \\
 & \quad : \emptyset \to pow_{\text{fin}}(\mathbb{A}) \\
\phi((\texttt{str of}\ N_X)\ \texttt{+}\ B_X) & = \begin{cases} \phi(N_X)(z) & \mathbf{Inl}(z) \\ \phi(B_X)(z) & \mathbf{Inr}(z) \end{cases} \\
 & \quad : \mathbf{Dom}(\phi(N_X)) + \mathbf{Dom}(\phi(B_X)) \to pow_{\text{fin}}(\mathbb{A})
\end{array}
$$

$X \setminus \vec{a}$ defined in N10.7.1, $\beta(C)$ defined in $(34)_{57}$.

Syntactic Construction

$$
\mathbf{fv} : \mathbf{PreStx} \longrightarrow pow_{\text{fin}}(\mathbb{A})
$$

$$
z \longmapsto \begin{cases} \mathbf{Supp}(z) & z \in \bigcup \mathbf{BType} \\ \mathbf{fv}(x) & z = \mathbf{In}_i(x) \\ \mathbf{fv}(x) \cup \mathbf{fv}(y) & z = (x, y) \\ \mathbf{fresh}\ n.\ \mathbf{fv}(f@n) \setminus \{n\} & z = f \in \text{AbsClass} \end{cases}
$$

$\mathbf{In}_i$ defined in N10.3.1.

Synthetic Construction

Figure 8. D10.7.2 - Free Variables **FV** For Free

10.7. Everything for free. In this subsection we demonstrate FM set theory and FM binding signatures by constructing some ubiquitous classes of functions on datatypes. This could conceivably be the underlying theory of a datatypes package for an automated proof environment or a programming language.

Notation 10.7.1. *For $\vec{a}$ a list of sets and X a set, we write*

$$
X \setminus \vec{a} \stackrel{\text{def}}{=} \big\{ x \in X \ \big|\ x \text{ does not occur in } \vec{a} \big\}
$$

Definition 10.7.2 (**FV** for free)**.** *To binding signatures B_X (D10.1.3) we inductively define an associated map $\phi(B_X)\colon B(pow_{\mathrm{fin}}(\mathbb{A})) \to pow_{\mathrm{fin}}(\mathbb{A})$ in Fig.8$_{76}$.*

*By L10.7.4 below, $pow_{\mathrm{fin}}(\mathbb{A})$ is (the underlying set) of a B_X-algebra and gives rise to a function (§10.3) $\overline{\phi(B_X)}\colon \boldsymbol{B} \to pow_{\mathrm{fin}}(\mathbb{A})$ which we write $\boldsymbol{FV}$ and call "**the free variables function**".*

*We also construct **fv** by $\in$-induction on **PreStx** (D10.3.2) as shown in Fig.8$_{76}$.*

The use of **fresh** in Fig.8$_{76}$ is legal by following lemma:

Lemma 10.7.3. 1. $X \in pow_{\mathrm{fin}}(\mathbb{A}) \wedge a \in \mathbb{A} \implies a\#X \setminus \{a\}$.
2. $X \in pow_{\mathrm{cof}}(\mathbb{A}) \wedge a \in \mathbb{A} \implies \neg(a\#X \setminus \{a\})$.

Proof. Direct from C9.4.4 (recall that $b\#Y \Leftrightarrow b \notin \mathbf{Supp}(Y)$ by N9.2.4). □

Recall the notation of D10.1.6.

Lemma 10.7.4. *$\phi(B_X)$ is a $\lambda X.B_X$-algebra over $pow_{\mathrm{fin}}(\mathbb{A})$. That is, for all binding signatures B_X,*

$$\phi(B_X)\colon B(pow_{\mathrm{fin}}(\mathbb{A})) \longrightarrow pow_{\mathrm{fin}}(\mathbb{A}).$$

Proof. By induction on the syntax of B_X. □

Lemma 10.7.5. *For any B_X, **FV** is the iteratively defined function we would normally write to construct the free variables function out of $t \in \boldsymbol{B}$.*

Proof. By observing that the iterative definition of **FV** on any particular **B** coincides with our expected notion of 'free variables'. We just need L9.3.6 to tell us $\mathbf{Supp}(a) = \{a\}$ for $a \in \mathbb{A}$. □

This purely syntactic notion coincides with a visibly corresponding function on **PreStx**:

Lemma 10.7.6. *For any B_X, **FV** defined on **B** coincides with **fv**.*

Proof. Observe by comparing Fig.9$_{79}$ with R10.3.3 that the iterative definition of **FV** on **B** precisely coincides with the inductive definition of **fv**. □

fv itself coincides with **Supp** on its domain of definition **PreStx**.

Lemma 10.7.7. ***fv** coincides with **Supp**.*

Proof. By induction on **PreStx**. The details of the induction depend on our particular realisation of syntax ($\mathbf{In}_i$, $(-,-)$ as function-classes on $\mathcal{V}_{\mathrm{FM}}$).

A more elegant though less rigorous proof is this: any realisation of syntax must use injective disjoint functions (see R4.3), which conserve support by L9.3.5. So an induction on any appropriate **PreStx** set *must* work. □

This is not merely satisfying, it is useful. We now have no less than three proof-methods for **FV**: iteration, $\in$-induction on **fv**, and theorems about **Supp**. In the language of R8.2.6, ***fv and Supp are synthetic versions of FV***.

Definition 10.7.8 (Closed Terms for Free)**.** *For B_X a binding signature let the **closed terms of $\boldsymbol{B}$**, written **Closed**$(\boldsymbol{B})$, be the set*

$$\boldsymbol{Closed}(\boldsymbol{B}) \stackrel{\text{def}}{=} \left\{ x \in \boldsymbol{B} \;\middle|\; \boldsymbol{FV}(x) = \emptyset \right\}.$$

The following lemma would be useful in a theorem-proving environment for porting results about free variables directly from **Supp**, so we do not even have to consider binding signatures.

Lemma 10.7.9.

$$\boldsymbol{Closed}(\boldsymbol{B}) = \left\{ x \in \boldsymbol{B} \;\middle|\; \boldsymbol{Supp}(x) = \emptyset \right\}$$

Proof. Trivial by L10.7.7. □

Definition 10.7.10 (Name-for-Name for Free)**.** *For B_X a binding signature (D10.1.3) and $a, b \in \mathbb{A}$ we inductively define*

$$\eta(B_X)\colon B(\boldsymbol{B}) \to B(\boldsymbol{B}) = \boldsymbol{B}$$

*in Fig.9$_{79}$. Just as in D10.7.2 this gives rise to a map $\overline{\eta(B_X)}\colon \boldsymbol{B} \to \boldsymbol{B}$ which we write $[b/a]$ and call "**the name-for-name substitution function**".*

Just as **Supp** is defined on the universe and **fv** is defined on **PreStx** and both coincide with each other and coincide with **FV** for particular datatypes (see R8.2.6), so we can define functions $[\![b/a]\!]'$ on the universe and $[\![b/a]\!]$ on **PreStx** (D10.3.2) that coincide with $[b/a]$:

Definition 10.7.11 (Synthetic Version)**.** *The function $[\![b/a]\!]$ is defined by $\in$-induction on **PreStx** as in Fig.9$_{79}$.*

Lemma 10.7.12. *For any binding signature B_X, the functions $[\![b/a]\!]$ and $[b/a]$ coincide on **B**.*

Proof. Fix a binding signature B_X and its initial datatype **B**. We work by induction on **B** with inductive hypothesis

$$[b/a]t = [\![b/a]\!]t.$$

Now we do not know what B_X is but we know any t is of the form

$$\mathbf{fresh}\, \vec{a_1}, \ldots, \vec{a_n}.\;\; \mathbf{In}_i(\vec{a_1}.s_1, \ldots, \vec{a_n}.s_n).$$

$$
\begin{aligned}
\eta(\texttt{nil}) &= \lambda x \in \mathbb{U}.x \\
&\quad : \mathbb{U} \to \mathbb{U} \\
\eta([\texttt{A}]^n X\ \texttt{x}\ N_X) &= \lambda x \in [\mathbb{A}]^n X \times \mathbf{Dom}(\eta(N_X)).x \\
&\quad : [\mathbb{A}]^n X \times \mathbf{Dom}(\eta(N_X)) \to [\mathbb{A}]^n X \times \mathbf{Dom}(\eta(N_X)) \\
\eta(C\ \texttt{x}\ N_X)\ , C \neq \texttt{A} &= \lambda x \in \beta(C) \times \mathbf{Dom}(\eta(N_X)).x \\
&\quad : \beta(C) \times \mathbf{Dom}(\eta(N_X)) \to C \times \mathbf{Dom}(\eta(N_X)) \\
\eta(\texttt{A}\ \texttt{x}\ N_X) &= \lambda x \in \mathbb{A}, y \in \mathbf{Dom}(\eta(N_X)). \begin{cases} (x,y) & x \neq a \\ (b,y) & x = a \end{cases} \\
&\quad : \mathbb{A} \times \mathbf{Dom}(\eta(N_X)) \to \mathbb{A} \times \mathbf{Dom}(\eta(N_X)) \\
\eta(\texttt{nil}) &= \lambda x \in \emptyset.\emptyset \\
&\quad : \emptyset \to \emptyset \\
\eta((\texttt{str of}\ N_X)\ \texttt{+}\ B_X) &= \begin{cases} \mathbf{Inl}(\eta(N_X)(z)) & \mathbf{Inl}(z) \\ \mathbf{Inr}(\eta(B_X)(z)) & \mathbf{Inr}(z) \end{cases} \\
&\quad : \mathbf{Dom}(\eta(N_X)) + \mathbf{Dom}(\eta(B_X)) \to \\
&\qquad \mathbf{Dom}(\eta(N_X)) + \mathbf{Dom}(\eta(B_X))
\end{aligned}
$$

$\beta(C)$ defined in $(34)_{57}$.

Syntactic Construction

$$[\![b/a]\!] : \mathbf{PreStx} \longrightarrow \mathbf{PreStx}$$

$$
z \longmapsto \begin{cases} b & z = a \in \mathbb{A} \\ z & z \in \mathbb{A},\ z \neq a \\ \mathbf{In}_i([\![b/a]\!]x) & z = \mathbf{In}_i(x) \\ ([\![b/a]\!]x, [\![b/a]\!]y) & z = (x,y) \\ \mathbf{fresh}\, n.\, n.[\![b/a]\!](f@n) & z = f \in \text{AbsClass} \end{cases}
$$

$\mathbf{In}_i$ defined in N10.3.1.

Synthetic Construction

Figure 9. D10.7.10 - Name-for-Name Substitution $[b/a]$ For Free

---

Here we use N10.5.1 to write $\vec{a_i}.$. The $\vec{a_i}$ are finite lists of variables of type $\mathbb{A}$ which are chosen fresh. By induction hypothesis we also know

$$\mathit{M}\vec{a_i}.\ [b/a]s_i = [\![b/a]\!]s_i$$

(see N10.5.1). Now if $[\![b/a]\!]$ and $[b/a]$ act on t in the same way given that they act the same way on the s_i, we are done. So we refer to Fig.9_{79} and see that both functions satisfy

$$t \stackrel{f}{\longmapsto} (\vec{a_1}.f(s_1), \ldots, \vec{a_n}.f(s_n)).$$

We consider $[\![b/a]\!]$ and omit the case $[b/a]$ (which is very fiddly, we use L10.1.5). Write $f = [\![b/a]\!]$. From Fig.9_{79} we see

$$f(t) = (f(\vec{a_1}.s_1), \ldots, f(\vec{a_n}.s_n)).$$

Now the $\vec{a_i}$ were chosen fresh for f, t so by Fig.9_{79} we can pull f inside the abstraction and we are done. □

We can now prove theorems about $[b/a]$ either by iteration on $\mathbf{B}$ or $\in$-induction on **PreStx**.

$[b/a]$ is defined for particular datatypes and $[\![b/a]\!]$ is defined on **PreStx**. The obvious definition of a function on the entire universe, we wrote it $[\![b/a]\!]'$, is this:

$$[\![b/a]\!]'x \stackrel{\text{def}}{=} \begin{cases} \{[\![b/a]\!]'y \mid y \in x\} & x \notin \mathbb{A} \\ x & x \neq a,\ x \in \mathbb{A} \\ b & x = b \end{cases} \tag{59}$$

This is wrong, it does not coincide with $[b/a]$ and $[\![b/a]\!]$ defined above. The problem is discussed in §11.1.

Now the last item on our list is "substitution for free". It is much harder to inductively construct some σ by induction on B_X in the style of Fig.8_{76} or Fig.9_{79} because the parameter $t \in \mathbf{B}$ depends on a particular B_X: if we consider the clause for $\eta(\mathbb{A} \times N_X)$ in Fig.9_{79} we might imagine a corresponding

$$\sigma(\mathtt{A\ x}\ N_X) = \lambda x \in \mathbb{A}, y \in \mathbf{Dom}(\eta(N_X)).\ \begin{cases} (x, y) & x \neq a \\ (t, y) & a \end{cases}$$
$$: \mathbb{A} \times \mathbf{Dom}(\sigma(N_X)) \to \mathbb{A} \times \mathbf{Dom}(\sigma(N_X))$$

—but this is nonsense. Firstly, $t \notin \mathbb{A}$. Secondly we cannot replace $\mathbb{A} \times \mathbf{Dom}(\sigma(N_X))$ on the right by $\mathbf{B} \times \mathbf{Dom}(\sigma(N_X))$ because (t, y) will in due course be wrapped in $\mathbf{In}_i$ by the final clause for $\sigma(\mathtt{str}$ of $N_X + B_X)$.[27]

[27] Incidentally, $\bar{\alpha}$ in *p.*72 can have no *synthetic* counterpart since different binding signatures can share a common naïve signature.

$$[t/a] : \mathbf{PreStx} \longrightarrow \mathbf{PreStx}$$

$$z \longmapsto \begin{cases} t & z = \mathbf{In}_i(a) \\ \mathbf{In}_i([t/a]z)) & z = \mathbf{In}_i(x),\ x \neq a \\ ([t/a]x, [t/a]y) & z = (x, y) \\ \mathbf{fresh}\, n.\, n.[t/a](f@n) & z = f \in \text{AbsClass} \end{cases}$$

PreStx defined in D10.3.2.

FIGURE 10. D10.7.13 - Substitution $[t/a]$ For Free

For the sake of argument, let us just take the synthetic version as primitive and carry on.

Definition 10.7.13 (Substitution for free)**.** *For t a set and $a \in \mathbb{A}$ we define a function-class written*

$$[t/a] \colon \boldsymbol{PreStx} \longrightarrow \boldsymbol{PreStx}$$

as in Fig.10$_{81}$*.*

There are many results about this function we might like to have. We only consider one, which is useful in §12.6. The reader will easily recognise this as a standard result:

Lemma 10.7.14. *For $x, t \in \boldsymbol{PreStx}$ and $a \in \mathbb{A}$*

$$a\#t \implies a\#[t/a]s.$$

PROOF. By induction on $s \in \mathbf{PreStx}$ using the inductive structure of **PreStx** in D10.3.2. □

See D12.6.2 for a brief continuation of this development.

Lemma 10.7.15. *For a binding signature of the form*

$$B_X = (\mathtt{str}_1\ \mathit{of}\ N_1)\ \mathtt{+}\ \ldots (\mathtt{str}_k\ \mathit{of}\ N_k)$$

such that each N_i is either equal to `A` *or does not mention the symbol* `A` *at all, and for $b \in \mathbb{A}$ and $s, t \in \boldsymbol{B}$, $[t/b]s$ is in $\boldsymbol{B}$ and is precisely what we understand by 'substitution of t for b in s'.*

PROOF. Similar to the proof of L10.7.12. □

The syntactic condition on B_X makes sure it is of a form for which we can sensibly talk about substitution. For example the result makes no sense for P_X defined in $(56)_{71}$.[28]

An interesting question is how we might extend substitution $[t/a]$ to a function on the universe just as **Supp** (N9.2.4) extends **fv** (D10.7.2). We may get some useful mileage out of a suitably modified version of $(61)_{83}$ but like that equation, this is an idea for future work. In the meantime we have more than enough to automate a lot of standard syntactic functions, and have shown how we might do others if necessary.

11. Questions

11.1. Name-for-name substitution. Rather than $(b\ a)$ why not base FM on name-for-name substitution $[b/a]$ (see $(59)_{80}$, there it is written $[\![b/a]\!]'$):

$$[b/a]x \stackrel{\text{def}}{=} \begin{cases} \{[b/a]y \mid y \in x\} & x \notin \mathbb{A} \\ x & x \neq a,\ x \in \mathbb{A} \\ b & x = b \end{cases} \tag{60}$$

Recall from T8.2.5 that α-equivalence $=_\alpha$ on syntax is usually defined in terms of a "name-for-name substitution" operator usually written $[b/a]$. We hinted in T8.2.5 and then proved in some generality in T10.5.8 that we can just as well construct $=_\alpha$ using the permutation action of FM set theory. But given that tradition uses $[b/a]$, is it possible to carry out a programme of extending ZFA to FM' where FM' is like FM only based on $[b/a]$ instead of $(b\ a)$?

The obvious approach is to duplicate the development of $\mathsf{И}$, **Supp** and $\#$ based on $[b/a]$. So we have

$$\mathsf{И}'a.\ \phi(a,\vec{x}) \stackrel{\text{def}}{=} \{a \in \mathbb{A} \mid \phi(a,\vec{x})\} \in pow_{\text{cof}}(\mathbb{A}). \tag{$\mathsf{И}'$}$$

[28] Incidentally, we see a related issue in [**49**, Cor 11.3, p.128 and §11.1,§14] in HOAS. One form of HOAS binding is to interpret it by a function space like $X \to X$. We get name-for-term substitution 'for free'—but on the other hand we *have* to have name-for-term substitution for free. In the case of the π-calculus (the example in [**49**] is 'Dynamic Logic' in Chapter 7) this is not what we need.

This problem does not arise in FM. We do have 'name-for-term' substitution for free on the datatype of processes, because $[t/a]$ (D10.7.13) is defined on all of **PreStx**. However, nothing says the sets so obtained must be in our datatype or in any other way significant, and in the case of terms of the π-calculus as declared in the binding signature on $(35)_{58}$, they are not.

This is identical to $(20)_{40}$ and we proceed. It is simplest to use the alternative characterisation of $\#$ in terms of N given in R9.5.8. $\#'$ is defined by

$$(\#',\mathbf{Supp}') \quad a\#'x \iff \mathsf{N}b.\ [b/a]x = x \quad \text{and} \quad \mathbf{Supp}'(x) = \left\{ a \in \mathbb{A} \;\middle|\; \neg(a\#'\mathbb{A}) \right\}.$$

But observe that

$$[b/a]\mathbb{A} = \mathbb{A} \setminus \{a\} \neq \mathbb{A}.$$

In fact for all $a \in \mathbb{A}$, $\neg(a\#'\mathbb{A})$ and so $\mathbf{Supp}'(\mathbb{A}) = \mathbb{A} \notin pow_{\mathrm{fin}}(\mathbb{A})$. Disaster! We lose the finite support property and therefore T9.4.6. This theorem is more-or-less the heart of FM, without it we have nothing.

Let us labour the point a little. Consider $\sim$ and $a.x$ (D9.4.13 and D9.5.1) reformulated:

$$(\sim') \qquad (a,x)\sim'(b,y) \stackrel{\text{def}}{\Leftrightarrow} \mathsf{N}'c.\ [c/a]x = [c/b]y$$

Take $x = y = \mathbb{A}$. Then $(a,\mathbb{A})\sim'(b,\mathbb{A})$ precisely when $a = b$. $a.'x$ is the $\sim'$-equivalence class of (a,x), so $a.'\mathbb{A} = \{(a,\mathbb{A})\}$.

Consider this more sophisticated alternative to $[b/a]$:

$$(61) \qquad [b/a]'x \stackrel{\text{def}}{=} \begin{cases} \left\{ [b/a]'y \;\middle|\; y \in x \right\} & a \in \mathbf{Supp}(x), x \notin \mathbb{A} \\ x & a \notin \mathbf{Supp}(x) \\ x & x \neq a,\ x \in \mathbb{A} \\ b & x = b \end{cases}$$

This is no good for building FM from scratch because it uses **Supp**, but it does satisfy $[b/a]'\mathbb{A} = \mathbb{A}$ and does coincide with the iterative $[b/a]$ and $\in$-recursive $[\![b/a]\!]$ of D10.7.10. I invite the reader to imagine automating this definition and using it as the basis for a general theory of name-for-name substitution without syntax. I leave an investigation of this to future work.

11.2. Significance of 'for free'. We spent §10.7 and all of §11 until now (we change the subject after this subsection) building various versions of well-known syntactic operations. We considered systematic inductive definitions out of datatypes, inductive definitions on a useful class of sets **PreStx** (D10.3.2) which coincide with the datatype definitions for all datatypes, and also pure set-function classes defined on the entire universe which coincide with the previous two where they are defined.

Sometimes, we even defined these function classes two or three times in different contexts, for example **FV**, **fv** and **Supp** (proved identical on their domains

in L10.7.6 and L10.7.7). I have said why but it bears repeating: when we define a function by induction on a datatype our *only* method of proof is inductive reasoning on the structure of that datatype. This is a hassle for at least two reasons. First, the datatype may be complex. Second, with each new datatype we must prove our theorems again using its new inductive principle. i.e. We may have proved $\mathbf{FV}(\lambda x.t) = \mathbf{FV}(t) \setminus \{x\}$ for one λ-calculus, but if we define another calculus, or start work on the π-calculus, we have to prove the result all over again.

If we have automated the construction of a class of functions across datatypes and proved once-and-for-all that they coincide with some function-class defined on the set-universe (as we did for **FV** in §10.7, equating it with **Supp**), we can reason just by the properties of the function-class.

This is excellent news, especially for those using theorem-proving environments.

11.3. Restricted set of permutations. When we constructed FM set theory in §9.1 we introduced the notion of 'supporting set'. We defined '$U \subseteq \mathbb{A}$ supports x' in Fig.3_{35} to mean $\Phi(U, x)$ where

$$\Phi(U, x) \stackrel{\text{def}}{=} \forall \pi \in F_{\mathbb{A}}.\ \Big(\forall u \in U.\ \pi{\cdot}u = u\Big) \implies \pi{\cdot}x = x. \tag{62}$$

Recall that $\Sigma_{\mathbb{A}}$ is the full set of permutations on $\mathbb{A}$ and $F_{\mathbb{A}}$ is the restricted set generated by the transpositions.

Why did we restrict to $F_{\mathbb{A}}$ and does it make a difference? Well, $F_{\mathbb{A}}$ is generated by transpositions and for $\Sigma_{\mathbb{A}}$ this is not necessarily true. If we build support using $F_{\mathbb{A}}$ we can effectively assume all permutations are transpositions. If the reader examines Chapter II he or she will see that we do just this. Using $F_{\mathbb{A}}$ is the 'elementary' option and it makes our lives easier.

The extra axiom that turns FM into ZFA is $(\text{Fresh})_{35}$, which insists that all sets x have some finite supporting set (and hence, by T9.2.1, have a least finite supporting set $\mathbf{Supp}(x)$). In particular, the $\pi \in \Sigma_{\mathbb{A}} \subseteq \mathcal{V}_{\text{FM}}$ must have finite support, so $\Sigma_{\mathbb{A}}$ in FM is *smaller* than in ZFA. In fact $F_{\mathbb{A}}$ and $\Sigma_{\mathbb{A}}$ are provably equal in FM (proof omitted). So using $F_{\mathbb{A}}$ does not actually make any difference once we are *in* FM.

But what about building the theory from scratch? Suppose we use $\Sigma_{\mathbb{A}}$ in $(62)_{84}$ to build a Φ'

$$\Phi'(U, x) \stackrel{\text{def}}{=} \forall \pi \in \Sigma_{\mathbb{A}}.\ \Big(\forall u \in U.\ \pi{\cdot}u = u\Big) \implies \pi{\cdot}x = x, \tag{63}$$

a corresponding $(\text{Fresh})_{35}$', and hence a theory FM$'$. Then:

1. Because $\Sigma_{\mathbb{A}} = F_{\mathbb{A}}$ in FM, any model of FM is a model of FM'. This gives us a relative consistency result.

2. $\Phi'(U,x)$ implies $\Phi(U,x)$ provably in FM' and so $(\text{Fresh})_{35}$' implies $(\text{Fresh})_{35}$ provably in FM'.
3. We can duplicate the entire development of support based on Φ only in FM' and can be certain that it all makes sense from the previous two points. We conclude by duplicating the proof that $\Sigma_{\mathbb{A}} = F_{\mathbb{A}}$.
4. Therefore $\Sigma_{\mathbb{A}}$ and $F_{\mathbb{A}}$ are provably equal in FM', so

$$\Phi(U,x) \iff \Phi'(U,x)$$

is provable in FM'. So FM and FM' are identical logical theories.

I find this rather amusing. If we do not restrict to $F_{\mathbb{A}}$, we can prove that we *could* have by pretending that we *had*.

11.4. FM and AC. Does $(\text{Fresh})_{35}$ contradict the axiom of choice AC? Yes:

Theorem 11.4.1 ($(\text{Fresh})_{35}$ implies $\neg$AC)**.** *There is no function-set*

$$\xi\colon pow_{\text{fin}}(\mathbb{A}) \longrightarrow \mathbb{A}$$

such that

$$\forall A \in pow_{\text{fin}}(\mathbb{A}).\ A \neq \emptyset \implies \xi(A) \in A.$$

Proof. $\xi \in \mathcal{V}_{\text{FM}}$ is a set and therefore has finite support. Choose $a, b\#\xi$ such that $a \neq b$ and set $A = \{a, b\}$. WLOG assume $\xi(A) = a$. Clearly $(a\ b)\cdot A = A$. Also by L9.2.7 $(a\ b)\cdot\xi = \xi$. By $(14)_{31}$ and the above we know

$$b = (a\ b)\cdot(\xi(A)) = \xi((a\ b)\cdot A) = \xi(A) = a.$$

This contradicts the choice $a \neq b$. □

This should not surprise us; FM is named after Fraenkel and Mostowski, who devised ZFA in the '20s and '30s and used their 'permutation models of set theory' to prove the independence of the Axiom of Choice (AC) from the other axioms of ZFA (and three decades later Cohen proved the harder result of independence of AC from set theory without atoms (ZF), via his celebrated forcing method; see [**38**, Section 6] for a brief survey). Here, we have turned their constructions to an entirely new end.

Remark 11.4.2 (FM $\to \neg$AC)**.** Quite often, careful formulations of the definition of capture-avoiding substitution use a choice function for picking out fresh variables, e.g. [**75**, Section 2]. The vague feeling that such concrete choices should be irrelevant crystallises here into the fact that such choice functions are *inconsistent* as we have just shown in T11.4.1.

In particular Hilbert's choice operator $\varepsilon x.\phi$ cannot be added to FM because it can be used to build choice functions. Proof assistants based on set theory or

higher order logic often use ε to provide anonymous notations for terms defined by formulas (see [**41**, Section 2.1]). ◇

Remark 11.4.3 (ι OK)**.** A ***unique choice*** or ***definite description*** operator ι is consistent with FM (proof omitted). For $\phi_{\vec{z}}$ a predicate on $\mathcal{V}_{\rm FM}$ in the logic of FM (possibly with parameters in $\vec{z}$)

$$\exists! y.\ \phi_{\vec{z}}(y) \implies \forall y.\ \Big(\phi_{\vec{z}}(y) \Rightarrow \iota x.\phi_{\vec{z}}(x) = y\Big),$$

i.e. $\iota x.\phi_{\vec{z}}(x)$ is the unique y such that ϕ if such a unique y exists.

Isabelle/FM inherits one from Isabelle/ZF (it is called `The`, see [**59**]). Because Isabelle functions have to be total `The` returns $\emptyset$ in the case that a unique y does not exist. I use ι in the proof of T9.6.6 and find it convenient to adopt this influence in the paper proof and let ι return $\emptyset$ when no unique choice exists (traditionally we might leave it undefined). Cf. R15.2.4 and R16.1.3. ◇

11.5. Consistency of FM**.** ***This subsection is conducted in*** ZFA ***set theory.*** Suppose ZFA is consistent. Choose *some particular* model $\mathcal{V}_{\rm ZFA}$ (cf. R8.1.7).

In this section we use ZFA to construct a subclass of $\mathcal{V}_{\rm ZFA}$ which satisfies the FM axioms. This proves relative consistency of FM wrt ZFA.

Definition 11.5.1. *Let the* ***class of hereditarily finitely supported sets*** *in* ZFA *be written* ***HFS*** *and defined by*

$$\boldsymbol{HFS}(x) \stackrel{\rm def}{=} \exists U \in pow_{\rm fin}(\mathbb{A}).\ \Phi(x,U)\ \wedge\ \forall y \in x.\ \boldsymbol{HFS}(y).$$

Recall that $\Phi(x,U)$ means "U supports x", see D9.1.2. In accordance with standard practice we shall abuse $\in$ and write $x \in \mathbf{HFS}$ for $\mathbf{HFS}(x)$.

Remark 11.5.2. We immediately have the following ***proof-method*** for **HFS**:

> $x \in \mathbf{HFS}$ precisely when x is finitely supported and for all $y \in x$, $y \in \mathbf{HFS}$.

We shall use it often, usually **without** referencing this remark explicitly. ◇

Corollary 11.5.3 (Downwards $\in$-closed)**.** *R11.5.2 has the useful corollary that* ***HFS*** *is* ***downwards $\in$-closed****: $x \in \boldsymbol{HFS}$ and $y \in x$ implies $y \in \boldsymbol{HFS}$.*

Lemma 11.5.4. *If f is a $\mathcal{V}_{\rm ZFA}$ function-class on $x_1,\ldots,x_n$ (and no other variables) and each x_i is finitely supported then $f(x_1,\ldots,x_n)$ is finitely supported.*

Proof. The proof is really just the proof of L9.3.4. ZFA is equivariant so by L8.1.12

$$(a\ b)\cdot f(x_1,\ldots,x_n) = f((a\ b)\cdot x_1,\ldots,(a\ b)\cdot x_n).$$

We assume each x_i is finitely supported so let $U_i \in pow_{\text{fin}}(\mathbb{A})$ support it. Then from this commutativity equation above it is clear that $U = \bigcup_i U_i$ supports $f(x_1, \ldots, x_n)$. Of course U is finite so we are done. □

Definition 11.5.5. *Define a new function-class on* ZFA *as*

$$\begin{aligned} pow_{fs} \colon \mathcal{V}_{\text{ZFA}} &\longrightarrow \mathcal{V}_{\text{ZFA}} \\ x &\longmapsto \big\{ y \subseteq x \;\big|\; y \text{ finitely supported} \big\} \end{aligned} \tag{64}$$

Lemma 11.5.6. *pow_{fs} restricted to **HFS** maps to **HFS**. i.e.*

$$x \in \boldsymbol{HFS} \implies pow_{fs}(x) \in \boldsymbol{HFS}.$$

Proof. We use R11.5.2. Suppose $x \in \mathbf{HFS}$. Then x is finitely supported and all $z \in x$ are in **HFS**. But if $y \subseteq x$ is finitely supported then all z in y are in x and so in **HFS** and hence $y \in \mathbf{HFS}$. We have just proved that all $y \in pow_{fs}(x)$ are in **HFS** and L11.5.4 says $pow_{fs}(x)$ is finitely supported, so again by R11.5.2 we know $pow_{fs}(x) \in \mathbf{HFS}$ as required. □

Corollary 11.5.7. *From L11.5.6 it immediately follows that if $x \in \boldsymbol{HFS}$ then $pow_{fs}(x) = \big\{ y \subseteq x \;\big|\; y \in \boldsymbol{HFS} \big\}$.*

Lemma 11.5.8. *$a \in \mathbb{A}$ is supported by $\{a\}$ and is empty, $\mathbb{A}$ is supported by $\emptyset$ and has atoms as elements, so*

$$a \in \boldsymbol{HFS} \quad \text{and} \quad \mathbb{A} \in \boldsymbol{HFS}.$$

Theorem 11.5.9. *The class $\boldsymbol{HFS} \subseteq \mathcal{V}_{\text{ZFA}}$ contains $\mathbb{A}$ and is closed under all the rules of* FM *set theory for the* ZFA *interpretations of $\mathbb{A}$ and $\in$. Thus **HFS** is a model of* FM *set theory in $\mathcal{V}_{\text{ZFA}}$.*

Proof. We verify **HFS** contains $\mathbb{A}$ and is closed under the rules of Fig.2_{26} and Fig.3_{35}.

For the reader not familiar with such proofs, the general idea is "if you put **HFS** in, you get **HFS** out", hence the word 'closed' in the statement of the result. Thus, variables and parameters will usually range over **HFS**, not $\mathcal{V}_{\text{ZFA}}$. However—and this can cause confusion—when we use ZFA function-classes to build sets which we then prove are in **HFS** , e.g. $\bigcup$, any universally or existentially quantified variables in the ZFA function-classes' definition range over $\mathcal{V}_{\text{ZFA}}$.

For a concrete example consider $(\text{Powerset})_{26}$. The statement "**HFS** is closed under $(\text{Powerset})_{26}$" actually means

$$\forall x \in \mathbf{HFS}.\ \exists y \in \mathbf{HFS}.\ \forall z \in \mathbf{HFS}.\ z \in y \leftrightarrow \forall w \in z.\ w \in x \tag{65}$$

—so in principle a proof of this from ZFA axioms is complicated by the possibility that might be $z \in \mathcal{V}_{\text{ZFA}}$ with $z \notin \mathbf{HFS}$ such that, say, $z \in y$ but not $\forall w \in z.\ w \in$

x. However because **HFS** is downwards $\in$-closed (C11.5.3) this cannot happen. $pow(x)$, the ZFA powerset function-class, does not satisfy $(65)_{87}$, we use $pow_{fs}(x)$ (D11.5.5) instead.

I shall *not* go through these details in the rest of the proof.

1. $\mathbb{A} \in \boldsymbol{HFS}$ by L11.5.8.

2. ***HFS** is closed under* $(\text{Fresh})_{35}$ by construction.

3. ***HFS** is closed under* $(\text{Sets})_{26}$, $(\text{Extensionality})_{26}$, $(\text{AtmInf})_{26}$. Follow using C11.5.3.

4. ***HFS** is closed under* $(\text{Infinity})_{26}$. $\mathbb{N} \in \mathcal{V}_{\text{ZFA}}$ is equivariant (N9.2.8, 'equivariant' means 'has empty support') and hereditarily equivariant for the obvious meaning of this terminology by analogy with D11.5.1. A set with empty support certainly has finite support, so $\mathbb{N} \in$ **HFS**, and this validates $(\text{Infinity})_{26}$ of **HFS** .

5. ***HFS** is closed under* $(\text{Union})_{26}$. Suppose $x \in$ **HFS**. Then x is finitely supported and all $y \in x$ are in **HFS**. So all $z \in y$ for $y \in x$ are in **HFS**. By L11.5.4 $\bigcup(x)$ is finitely supported. All $z \in \bigcup(x)$ are in **HFS** by our previous observation. Thus $\bigcup(x) \in$ **HFS**.

6. ***HFS** is closed under* $(\text{Pairset})_{26}$. The standard pairset function in $\mathcal{V}_{\text{ZFA}}$ is $x, y \mapsto (x, y) = \{\{x, y\}, \{x\}\}$. Suppose $x, y \in$ **HFS**. Then x, y are finitely supported. By L11.5.4 so are (x, y), $\{x\}$ and $\{x, y\}$.[29] So $\{x\}, \{x, y\}$ are in **HFS**. Now $(x, y) = \{\{x, y\}, \{x\}\}$ so $(x, y) \in$ **HFS**.

7. ***HFS** is closed under* $(\text{Powerset})_{26}$. By L11.5.6, C11.5.7, and $(65)_{87}$ above.

8. ***HFS** is closed under* $(\text{Collection})_{26}$. For ϕ a predicate let f be the function-class on $\mathcal{V}_{\text{ZFA}}$ taking x to $\{y \in x \mid \phi(y)\}$. If ϕ is parameterised we fix the parameters and assume them to be in **HFS** and therefore finitely supported.[30] Suppose $x \in$ **HFS**. Then every $y \in x$ is in **HFS** and x is finitely supported. L11.5.4 implies that $f(x)$ is finitely supported. Furthermore as already observed, $y \in f(x)$ is in x and so in **HFS**. Therefore $f(x) \in$ **HFS**.

9. ***HFS** is closed under* $(\text{Replacement})_{26}$. Let F be a function-class from **HFS** to **HFS**. As in the case of $(\text{Collection})_{26}$ if it is parameterised we assume the parameters are in **HFS**. For $x \in$ **HFS** every $y \in x$ is in **HFS**. By assumption

[29] We don't need L11.5.4 to see this. If U supports x ($\Phi(U, x)$) and V supports y ($\Phi(V, y)$) then from D9.1.2 it is evident that $\Phi(U \cup V, z)$ for $z = (x, y), \{x\}, \{x, y\}$.

[30] Recall from the beginning of the proof that we're trying to prove "**HFS** is closed under $(\text{Collection})_{26}$", so we assume all free variables and parameters range over **HFS** and prove what we get out is also in **HFS**.

$F(y) \in \mathbf{HFS}$. By L11.5.4 the $\mathcal{V}_{\mathrm{ZFA}}$ set $\{F(y) \mid y \in x\}$ is finitely supported. Therefore it is in **HFS**.

10. ***HFS*** *satisfies* $(\in\text{-Induction})_{26}$. By L11.5.11 below **HFS** is equal to the cumulative hierarchy $\mathcal{M}_{\mathrm{FM}}$ (D11.5.10 below) built up in ZFA from $\mathbb{A}$ using pow_{fs} $((64)_{87})$. Such a cumulative hierarchy clearly satisfies $\in$-induction.

This concludes the proof. □

Definition 11.5.10. *Write* $\mathcal{M}_{\mathrm{FM}} \subseteq \mathcal{V}_{\mathrm{ZFA}}$ *for the cumulative hierarchy built up from* $\mathbb{A}$ *using* pow_{fs} *instead of* pow*. 'M' stands for 'model'.*

Lemma 11.5.11. $\mathcal{M}_{\mathrm{FM}} = \boldsymbol{HFS}$.

Proof. $\mathbb{A} \in \mathbf{HFS}$ and as proved in Case 7 above, **HFS** is closed under pow_{fs}. Thus $\mathcal{M}_{\mathrm{FM}}$ is a subclass of **HFS**.

Conversely we must show that $\mathbf{HFS} \subseteq \mathcal{M}_{\mathrm{FM}}$. This is proved by ZFA $\in$-induction on $\mathcal{V}_{\mathrm{ZFA}}$ with inductive hypothesis

$$\phi(x) \stackrel{\text{def}}{=} \big(x \in \mathbf{HFS} \to x \in \mathcal{M}_{\mathrm{FM}}\big).$$

The proof is elementary but omitted. □

Corollary 11.5.12. FM *set theory is consistent wrt* ZFA *set theory.*

Proof. A corollary of T11.5.9. Given a model $\mathcal{V}_{\mathrm{ZFA}}$ of ZFA, $\mathbf{HFS} = \mathcal{M}_{\mathrm{FM}} \subseteq \mathcal{V}_{\mathrm{ZFA}}$ provides a model of FM. □

12. Inductive reasoning in FM

Now we change direction and consider a simple but prototypical case study of inductive reasoning on inductively defined sets with α-equivalence.

Chapter IV contains a long proof of an interesting result (T21.9) about a programming language FreshML. The proof is in FM; the datatype of terms of FreshML is defined up to α-equivalence using the tools of §10 and reasoning is carried out in the theory of FM, including the M-quantifier. So Chapter IV is a huge case study. Our prototypical example is FML$_{tiny}$, a drastically simplified version of FreshML. We shall construct its syntax and typing and evaluation relations for it, all in FM, and prove a few results about them. We shall dot our i's and cross our t's here and take this as a license not to in Chapter IV.

12.1. The syntax of FML$_{tiny}$.

Definition 12.1.1 (FML$_{tiny}$ syntax)**.** FML$_{tiny}$ *is a typed* λ*-calculus, its terms, types, and values are defined in Fig.11*$_{90}$.

$\tau ::=$	$\texttt{Unit}$	Unit Type
	$\mid \tau \to \tau$	Function type

TYPES OF FML*tiny*

$t ::=$	$\texttt{TheUnit}$	The Unit
	$\mid x$	Variable Symbols
	$\mid \texttt{Fix}(\tau, t_{**})$	Function fixedpoints
	$\mid t\ t$	Application

TERMS OF FML*tiny*

$V ::=$	$\texttt{TheUnit}$	
	$\mid x$	Variable Symbols
	$\mid \texttt{Fix}(\tau, t_{**})$	Function fixedpoints

VALUES OF FML*tiny*

Consistent with N9.5.3, t_{**} denotes an atom-abstraction of an atom-abstraction of a term t. Corresponding binding signatures are

$$Ty_X \stackrel{\text{def}}{=} \overset{\texttt{Unit}}{\texttt{Unit}} \text{ of } \texttt{U} + \overset{\tau \to \tau}{\texttt{Func}} \text{ of } X\ \texttt{x}\ X$$

$$Tm_Y \stackrel{\text{def}}{=} \overset{\texttt{TheUnit}}{\texttt{TheUnit}} \text{ of } \texttt{U} + \overset{x, y, z, \ldots}{\texttt{Var}} \text{ of } \texttt{A} + \overset{\texttt{Fix}(\tau, t_{**})}{\texttt{Fix}} \text{ of } X\ \texttt{x}\ [\texttt{A}]^2\, Y + \overset{t\ t}{\texttt{App}} \text{ of } Y\ \texttt{x}\ Y$$

$$Vl_Z \stackrel{\text{def}}{=} \overset{\texttt{TheUnit}}{\texttt{TheUnit}} \text{ of } \texttt{U} + \overset{x, y, z, \ldots}{\texttt{Var}} \text{ of } \texttt{A} + \overset{\texttt{Fix}(\tau, t_{**})}{\texttt{Fix}} \text{ of } X\ \texttt{x}\ [\texttt{A}]^2\, Y$$

FIGURE 11. D12.1.1 - Types, Terms, and Variables of FML*tiny*

Remark 12.1.2 (Technical points)**.** 1. No sooner do I define a notation than I abuse it. Fig.11_{90} is naughty because the initial algebras $\mathbf{Ty}, \mathbf{Tm}$ and $\mathbf{Vl}$ respectively defined by Ty_X, Tm_Y and Vl_Z are constructed by

mutual induction, which we did not formally develop in §10. I had mutual induction in mind when I set up B_X labelled with a variable symbol X. I gloss over the issue, quoting Bekić's theorem which shows how mutual inductive definitions can be encoded in ordinary inductive definitions (see [**78**, §10.1, p.162]).

2. It is unclear whether **Vl** and **Tm** are separate datatypes, or whether **Vl** is an inductively defined subset of **Tm**. I gloss over this too.

◇

This is a more interesting point:

Remark 12.1.3 (*Not* ZF grammars)**.** Observe from Tm_Y in Fig.11_{90} that the term-former for (fixedpoint) function abstraction `Fix` takes as arguments a type τ and a double abstraction t_{**} of a term. A more traditional writing of this is

$$\texttt{fix}\,\underline{f}(\underline{x}:\tau)\,\texttt{in}\ t,$$

where underlined variables are bound.[31] I shall call this '***nameful style***'; though bound variables have no names, we present them as if they did. I shall call the style of Fig.11_{90} '***nameless style***'. This follows the terminology first mentioned in R4.14.

The nameful presentation is more familiar and I use it to define FreshML in Fig.31_{164}. There is nothing wrong with explicit variable names so long as we understand that they are bound. So a 'nameful' presentation of the grammar for terms of Fig.11_{90} is this:

$$t ::= \texttt{TheUnit} \mid x \mid \texttt{fix}\,\underline{f}(\underline{x}:\tau)\,\texttt{in}\ t \mid t\ t. \tag{66}$$

They are *identical declarations*, the difference is stylistic. We shall see more about nameful reasoning in FM in the course of this section, see for example R12.4.1.

◇

12.2. Inductive reasoning on the syntax of FML*tiny***.** What *is* induction on the syntax of terms? We consult §10.4 and in particular $(55)_{71}$ and (details

[31] This notation is not definitive because it does not specify the scope of the binding, cf. ft.75_{163}.

omitted) deduce the following induction scheme:

$$
\begin{aligned}
&\forall \phi\colon \mathbf{Tm} \to \mathbb{B}.\\
&\Big(\ \forall x \in \mathtt{Unit}.\ \phi(\mathtt{TheUnit}(x)) \quad \wedge\\
&\ \ \forall x \in \mathbb{A}.\ \phi(\mathtt{Var}(x)) \quad \wedge\\
&\ \ \forall t_1, t_2 \in \mathbf{Tm}.\ \phi(t_1) \wedge \phi(t_2) \Rightarrow \phi(\mathtt{App}(t_1, t_2)) \quad \wedge\\
&\ \ \forall \tau \in \mathbf{Ty}, t_{**} \in [\mathbb{A}]^2\mathbf{Tm}.\ \mathit{M} f, u.\ \phi((t_{**}@f)@u) \Rightarrow \phi(\mathtt{Fix}(\tau, t_{**}))\ \Big)\\
&\qquad \Longrightarrow\ \forall x \in \mathbf{Tm}.\ \phi(x)
\end{aligned}
\tag{67}
$$

In N4.5 we set up a notation that $\mathtt{Var}, \mathtt{App}$ and $\mathtt{Fix}$ are pure syntax term-formers modelled in FM by functions $\mathbf{Var}, \mathbf{App}$ and $\mathbf{Fix}$. Now that we have binding signatures (D10.1.3), adequacy for syntax (T10.5.8) and all the other apparatus of §10, we can be comfortable that our FM syntactic sets are an accurate reflection of abstract syntax 'in nature'. So we just write $\mathtt{Var}$, $\mathtt{App}$, $\mathtt{Lam}$ for the set-function versions of the term-formers too, as in $(67)_{92}$ above. Cf. R21.11.

Theorem 12.2.1 (Substitution Property)**.** *If t is a term, $x \in \mathbb{A}$ a variable symbol and V a value then $t[V/x]$ (where $[-/-]$ denotes substitution as in D10.7.13 but written postfix instead of prefix[32]) is a well-formed term. Furthermore, if U is a value then $U[V/x]$ is also a value.*

Proof. We prove the first assertion only, by FM-induction on t with inductive hypothesis

$$\forall V \in \mathbf{Vl}.\ t[V/x] \in \mathbf{Tm}.$$

1• *Suppose* $t = \mathtt{TheUnit}$. Then $t[V/x] = t$ and the result is trivial.

2• *Suppose* $t = x$. Then $t[V/x] = V$. Observe that values V are terms (cf. Item 2 of R12.1.2). So the result follows.

3• *Suppose* $t = t_1\, t_2$. Then $t[V/x] = t_1[V/x]\, t_2[V/x]$. We work by induction on the syntax so we have the inductive hypothesis for t_1 and t_2. Therefore $t_1[V/x]$ and $t_2[V/x]$ are both terms and so is their application.

4• *Suppose* $t = \mathtt{Fix}(\tau, t_{**})$. This is the interesting case. By the clause for $\mathtt{Fix}$ of $(67)_{92}$ it suffices to choose atoms $f, u \in \mathbb{A}$ fresh for the current context, so in particular $f, u\#t_{**}, x, V$, and verify the inductive hypothesis of t assuming the inductive hypothesis of $t' \stackrel{\text{def}}{=} (t_{**}@f)@u$ (so $t = \mathtt{fix}\, f(u:\tau)\, \mathtt{in}\ t'$).

[32] ... for no particular reason.

(68) $\Gamma \vdash \texttt{TheUnit} : \texttt{Unit}$

(69) $\Gamma \vdash x : \tau \quad (\Gamma(x) = \tau)$

(70) $$\frac{\Gamma \vdash t_1 : \tau \to \tau' \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 t_2 : \tau'}$$

(71) $$\mathit{И} f, u.\ \frac{\Gamma, f : \tau \to \tau', u : \tau \vdash (t_{**} @ f) @ u : \tau'}{\Gamma \vdash \texttt{Fix}(\tau, t_{**}) : \tau \to \tau'}$$

FIGURE 12. D12.3.1 - Typing Judgements of FML*tiny*

We therefore assume $t'[V/x]$ is a term. Now from $f, u \# x$ it follows by L9.3.6 that $f, u \neq x$. Also from $f, u \# V$ it follows by L10.7.7 that $f, u \notin \mathbf{FV}(V)$. So there are no problems with variable capture and

$$(\texttt{fix}\, f(u : \tau)\, \texttt{in}\ t')[V/x] = (\texttt{fix}\, f(u : \tau)\, \texttt{in}\ t'[V/x])$$

Since the RHS is a term, so is the LHS. This gives us the result.

□

12.3. Typing of FML*tiny*.

Definition 12.3.1 (FML*tiny* Typing)**.** *Typing contexts are partial functions with finite domain from* $\mathbb{A}$ *to types* ***Ty****, written in standard style as a finite list* Γ*. Write the typing contexts* $\boldsymbol{Ctx}_{typ}$*.*

Typing judgements are a subset of $\boldsymbol{Ctx}_{typ} \times \boldsymbol{Tm} \times \boldsymbol{Ty}$ *inductively defined by the rules of Fig.12*$_{93}$*. We write* $\boldsymbol{Judge}_{typ}$ *for the* ***valid typing judgements*** *of* FML*tiny*.

Recall that the inductive rule

$$\frac{\phi(\vec{x})}{\psi(\vec{x})} \quad \text{is shorthand for} \quad \forall \vec{x}.\ \phi(\vec{x}) \Rightarrow \psi(\vec{x})$$

in ZF. The language of FM has the $\mathit{И}$-quantifier, so for $\vec{x} = \vec{y}, \vec{x'}$ we may now write inductive rules

$$\mathit{И}\vec{y}.\ \frac{\phi(\vec{x})}{\psi(\vec{x})} \quad \text{as shorthand for} \quad \forall \vec{x'}.\ \mathit{И}\vec{y}.\ \phi(\vec{x}) \Rightarrow \psi(\vec{x}).$$

The rule for `Fix` in Fig.12$_{93}$ written in traditional *ZF*-style is

$$\frac{\Gamma, f:\tau\to\tau', u:\tau \vdash t:\tau'}{\Gamma \vdash \mathtt{fix}\, f(x:\tau)\,\mathtt{in}\; t:\tau\to\tau'}.$$

Now $\Gamma, f:\tau\to\tau', u:\tau$ is only well-formed when $f, u \notin \mathbf{Dom}(\Gamma)$. Traditionally we insist that we only ever write $\Gamma, x:\tau$ when $x \notin \mathbf{Dom}(\Gamma)$. This amounts to adding a side-condition $x \notin \mathbf{Dom}(\Gamma)$.

We can imitate this in FM by writing out t_{**} in nameful style (R12.1.3) with explicit variable names:

$$\frac{\Gamma, f:\tau\to\tau', u:\tau \vdash t:\tau'}{\Gamma \vdash \mathtt{fix}\, f(x:\tau)\,\mathtt{in}\; t:\tau\to\tau'}$$

We do this in Chapter IV, for example in (126)$_{180}$, (125)$_{180}$ and (133)$_{181}$. It looks just the same as normal rules. But FM is different. The understanding is we are just writing out t_{**} with explicit variable names *which may as well be chosen new!* So we really mean:

$$(72)\qquad \mathit{И} f, u.\ \forall t.\ \frac{\Gamma, f:\tau\to\tau', u:\tau \vdash t:\tau'}{\Gamma \vdash \mathtt{fix}\, f(x:\tau)\,\mathtt{in}\; t:\tau\to\tau'}$$

f, u are new for everything except t (see R9.4.12). So $f, u\#\Gamma$ and by L12.4.2 this implies that $f, u \notin \mathbf{Dom}(\Gamma)$ and the side condition is taken care of.

Remark 12.3.2. The slogan is:

> **Unless otherwise indicated, bound variable names explicitly appearing in an inductive rule are assumed to be created new for variables outside the scope of the binding for that variable,** cf. R4.12.

◇

Remark 12.3.3 (Monster proof nameful)**.** In Fig.12$_{93}$ we do it 'properly' with `Fix`(τ, t_{**}). In Chapter IV everything is nameful, in accordance with ordinary practice, but the slogan above is observed throughout, the difference is merely stylistic. It is implicitly used (without any comment) wherever binders occur, for example in (125)$_{180}$, (126)$_{180}$, (133)$_{181}$, or (148)$_{188}$. ◇

12.4. Type uniqueness.

Remark 12.4.1 (Nameful reasoning on $\mathbf{Judge}_{typ}$)**.** $\mathbf{Judge}_{typ}$ (D12.3.1) is an inductively defined set, a least-fixedpoint of an appropriate monotone operator. *How do we reason about such a set?* By induction over its rules given in Fig.12$_{93}$, just as we normally would. The $\mathit{И}$-quantifier in the rule for `Fix` makes absolutely no odds besides allowing us to assume f and u are new when we reason

about them.[33] It is at precisely at this point that we first see how FM allows us to manipulate nameless terms in a nameful way. See R4.14 and R12.1.3. ◇

Lemma 12.4.2. *If $\Gamma \in \mathcal{V}_{\mathrm{FM}}$ is a function-set with finite domain (the obvious candidate being of course an* FML*tiny typing context) and $a\#\Gamma$ then $a \notin \boldsymbol{Dom}(\Gamma)$.*

Proof. $\mathbf{Dom}(\Gamma)$ is finite so $\mathbf{Supp}(\mathbf{Dom}(\Gamma)) = \mathbf{Dom}(\Gamma)$ by L9.3.2. By L9.3.4 we also have $\mathbf{Supp}(\mathbf{Dom}(\Gamma)) \subseteq \mathbf{Supp}(\Gamma)$ and a is not in the RHS. Put together this gives us $a \notin \mathbf{Dom}(\Gamma)$ as required. □

This result is often applicable because we often have $x\#\Gamma$, where x was chosen new by a rule such as $(72)_{94}$. For an example consider the case of `Fix` in T12.4.3 below. Cf. also §13.2.

Theorem 12.4.3 (Type Uniqueness)**.** *For Γ, t and τ such that $\Gamma \vdash t : \tau$,*

$$\forall \tau'.\ \Gamma \vdash t : \tau' \implies \tau' = \tau.$$

Proof. By induction on the typing rules using inductive hypothesis

$$\Gamma \vdash t : \tau \implies \forall \tau'.\ \Gamma \vdash t : \tau' \implies \tau' = \tau.$$

1• *Suppose $\Gamma \vdash t : \tau$ is deduced using $(68)_{93}$.* Trivially $t = \texttt{TheUnit}$ and $\tau = \texttt{Unit}$. By pattern-matching against the rules for any other $\Gamma \vdash \texttt{TheUnit} : \tau'$ it is the case that $\tau' = \texttt{Unit}$. We skip to the last typing rule.

2• *Suppose $\Gamma \vdash t : \tau$ is deduced using $(71)_{93}$.* Then $t = \texttt{Fix}(\sigma, t'_{**})$ and $\tau = \sigma_1 \to \sigma_2$. We now choose new $f, u\#t'_{**}, t, \Gamma, \tau$ and write $t'_{**} = f.(u.t')$ so that

$$t = \texttt{fix}\, f(u : \sigma_1)\ \texttt{in}\ t'.$$

We have the inductive hypothesis for the judgement

$$\Gamma, f : \sigma_1 \to \sigma_2, u : \sigma \vdash t' : \sigma_2.$$

This is well-formed because $f, u\#\Gamma$ so by L12.4.2, $f, u \notin \mathbf{Dom}(\Gamma)$.

[33] If this does not convince the reader, consider that for any rule

$$И x.\ \frac{\phi(x,y)}{\psi(x,y)}$$

by T9.4.6 it is *actually logically equivalent* to the rule

$$\frac{(x\#\phi, \psi, y) \quad \phi(x,y)}{\psi(x,y)}.$$

This version is less convenient but it is traditional ZF in the sense that the inductive rule is of standard nameful form $\forall \mathit{FreeVars}.\ \mathit{Assumptions} \Rightarrow \mathit{Conclusions}.$ $x\#\phi, \psi, y$ are FM predicates but that has nothing to do with the induction rule itself.

Now suppose $\Gamma \vdash t : \tau'$. Because of the syntactic form of t we know this was deduced by an application of $(71)_{93}$. We follow back the deduction and apply the inductive hypotheses in the standard way. This gives the result.

□

12.5. What FM **gives us.** What has FM given us? Not that much, but that should not surprise us since if FML*tiny* got any simpler it would cease to exist. FM has simply delivered what it always promised: proofs by pure structural induction and general tidying up of the treatment of bound variables. Thus T12.2.1 would normally be proved by induction on term length, here we have seen it proved by pure structural induction. In T12.4.3, itself by structural induction, newness of f and u ensured well-formedness of the typing context. There are more complex results out there as well. For example in a type-substitution property

$$(\Gamma, x : \tau \vdash t : \sigma) \ \wedge \ (\Gamma \vdash U : \tau) \implies \Gamma \vdash t[U/x] : \sigma$$

we would find both the qualities above useful. The industrial-strength application of FM is in Chapter IV and it contains precisely this result, twice for two different judgements, and proved by pure structural induction on the derivation (T23.1.14 and L24.1.8, also C24.1.9). We see other and more sophisticated uses of $\#$ as well, e.g. $\equiv^{se}$ in D26.3.1 and its interaction with, say, evaluation in T26.4.7.

Returning to FML*tiny*, the fact that we got what we wanted in a prosaic way is part of the basis for my claim that FM is 'simple' and 'natural'.

Remark 12.5.1. Now is a good opportunity to mention that FM does deliver a little more than just structural induction. We have equivariance (L8.1.12) for the datatypes and functions and predicates on them (modulo parameterisation of course) as well. FML*tiny* is too simple for this to matter, but in a more complex language it may give us a few more results 'for free' than we expected. For example in a language with dynamically allocated local state, where new locations would be modelled by new atoms, equivariance could come in jolly useful. Consider for example [**67**, p.5, Lemma 1], which is a correctness result for evaluation with respect to different choices for new locations. ◇

12.6. Evaluation of FML*tiny***.** This subsection ostensibly studies evaluation on FML*tiny*, but I really use it to study the interaction of FM-abstractions with standard deduction rules.

Definition 12.6.1. *We define* ***Closed terms and values****,* $\boldsymbol{CTm}$ *and* $\boldsymbol{CVl}$ *respectively, by*

$$\boldsymbol{CTm} \stackrel{\text{def}}{=} \left\{ t \in \boldsymbol{Tm} \;\middle|\; \boldsymbol{Supp}(t) = \emptyset \right\} \quad \boldsymbol{CVl} \stackrel{\text{def}}{=} \left\{ V \in \boldsymbol{Vl} \;\middle|\; \boldsymbol{Supp}(V) = \emptyset \right\}.$$

(74) $\texttt{TheUnit} \Downarrow \texttt{TheUnit}$

(75) $$\frac{t_1 \Downarrow \texttt{Fix}(\tau, t_{**}) \quad t_2 \Downarrow V_2 \quad (t_{**}@@\texttt{Fix}(\tau, t_{**}))@@V_2 \Downarrow V}{t_1\, t_2 \;\Downarrow\; V}$$

(76) $\texttt{Fix}(\tau, t_{**}) \;\Downarrow\; \texttt{Fix}(\tau, t_{**})$

FIGURE 13. D12.6.4 - Evaluation Relation of FML$_{tiny}$

Here we use "**FV** for free" (see L10.7.7 and the preceding discussion), which allows us to use **Supp** directly without bothering to inductively define a "free variables" function.

To define evaluation we use $[t/a]$ the "substitution for free" defined in D10.7.13. However, it is convenient to use a slightly different flavour of this function:

Definition 12.6.2 (Name-free substitution)**.** *We define* ***name-free substitution*** *using* $[t/a]$ *(D10.7.13) by*

(73) $$@@ : [\mathbb{A}]\boldsymbol{PreStx} \times \boldsymbol{PreStx} \longrightarrow \boldsymbol{PreStx}$$
$$x_*@@t = \boldsymbol{fresh}\, a.\, [t/a](x_*@a).$$

Because a is fresh, $a\#t$ and by L10.7.14 we deduce $a\#[t/a](x_*@a)$. So the use of **fresh** is legal (C9.6.7) and $(73)_{97}$ is well-defined. The variable to be substituted for is 'marked' by being bound (cf. `subst2`, D4.11).

We see why we prefer @@ to $[t/a]$ in $(75)_{97}$ and R12.6.5. It behaves well:

Lemma 12.6.3. *For* $s, t \in \boldsymbol{PreStx}$ *and* $a \in \mathbb{A}$*,*

$$(a.s)@@t = [t/a]s.$$

PROOF. By induction over $s \in \mathbf{PreStx}$ using the inductive structure of the set (D10.3.2). We use the inductive hypothesis

$$\forall t \in \mathbf{PreStx}, a \in \mathbb{A}.\ (a.s)@@t = [t/a]s.$$

The proof is a bit messy and I give no details. □

Definition 12.6.4 (FML$_{tiny}$ Evaluation)**.** ***The evaluation relation****, written* $\boldsymbol{Eval}_{\text{FML}tiny}$*, is a subset of* $\boldsymbol{CTm} \times \boldsymbol{CVl}$ *(D12.6.1) inductively defined by the rules of Fig.13*$_{97}$*.*

Remark 12.6.5. In $(75)_{97}$ we use FM dialect but we can rephrase it in nameful ZF style (R4.14) using L12.6.3 to rewrite @@ in terms of substitution and the convention of R12.3.2 to expand abstractions with implicitly fresh atoms f, x;

$$\frac{t_1 \Downarrow \mathtt{fix}\, f(x:\tau)\,\mathtt{in}\ t \quad t_2 \Downarrow V_2 \quad [\mathtt{fix}\, f(x:\tau)\,\mathtt{in}\ t/f, V_2/x]t \Downarrow V}{t_1\ t_2\ \Downarrow\ V}$$

meaning

(77) $\mathit{N} f, x.\ \forall t.$

$$\frac{t_1 \Downarrow \mathtt{fix}\, f(x:\tau)\,\mathtt{in}\ t \quad t_2 \Downarrow V_2 \quad [\mathtt{fix}\, f(x:\tau)\,\mathtt{in}\ t/f, V_2/x]t \Downarrow V}{t_1\ t_2\ \Downarrow\ V},$$

cf. R9.4.12 (binding under N). This is what is happening in $(147)_{188}$ and many other rules of FreshML, cf. R25.5. ◇

13. More set theory

I finish Chapter II by clearing up a few loose ends. We start in §13.1 with L13.1.1, which relates the support of x to the supports of its elements (always a useful sort of equality to know in set theory). §13.2 uses L13.1.1 to develop the theory of the support of syntactic sets.

13.1. Advanced theory of Supp.

Lemma 13.1.1. *For all x,*

$$\boldsymbol{Supp}(x) = \begin{cases} \{x\} & x \in \mathbb{A} \\ \boldsymbol{Supp}(\bigcup_{y\in x} \boldsymbol{Supp}(y)) & x \notin \mathbb{A}. \end{cases}$$

Proof. The base case is L9.3.6. Concerning the second clause, the right-to-left inclusion $\mathbf{Supp}(\bigcup_{y\in x} \mathbf{Supp}(y)) \subseteq \mathbf{Supp}(x)$ is L9.3.4 for $f = \lambda x. \bigcup_{y\in x} \mathbf{Supp}(y)$.

The left-to-right inclusion $\mathbf{Supp}(x) \subseteq \mathbf{Supp}(\bigcup_{y\in x} \mathbf{Supp}(y))$ is as follows. Let us write

$$\mathcal{S} \quad \text{for} \quad \bigcup_{y\in x} \mathbf{Supp}(y).$$

Suppose there exists some $a \in \mathbb{A}$ such that

$$a \in \mathbf{Supp}(x) \quad \text{and} \quad a \notin \mathbf{Supp}(\mathcal{S}) \subseteq \mathbf{Supp}(x).$$

Choose some $b \notin \mathbf{Supp}(x)$. Then (by L9.2.7)

$$(a\ b)\cdot x \neq x \quad \text{and} \quad (a\ b)\cdot \mathcal{S} = \mathcal{S}.$$

Since $(a\ b)\cdot x = \{(a\ b)\cdot y \mid y \in x\}$ (D8.1.8) we can choose $y \in x$ such that $(a\ b)\cdot y \notin x$ and therefore $(a\ b)\cdot y \neq y$. Also by construction of $\mathcal{S}$ we know $\mathbf{Supp}(y) \subseteq \mathcal{S}$ and since $a, b \notin \mathbf{Supp}(\mathcal{S})$ we have (using $(13)_{30}$ for $\in$)

$$(a\ b)\cdot\mathbf{Supp}(y) \subseteq \mathcal{S}.$$

Now $\mathcal{S}$ is a subset of $\mathbb{A}$ so by L9.4.3 either

$$\mathbf{Supp}(\mathcal{S}) = \mathcal{S} \quad \text{or} \quad \mathbf{Supp}(\mathcal{S}) = \mathbb{A} \setminus \mathcal{S}.$$

We consider only the first case, the second case is similar. Because of this,

$$\mathbf{Supp}(y) \subseteq \mathbf{Supp}(\mathcal{S}) \quad \text{and} \quad (a\ b)\cdot\mathbf{Supp}(y) \subseteq \mathbf{Supp}(\mathcal{S}).$$

Now because $(a\ b)\cdot y \neq y$ we know at least one of a and b is in $\mathbf{Supp}(y)$, and therefore b or a is in $(a\ b)\cdot\mathbf{Supp}(y)$. Hence

$$b, a \in \mathbf{Supp}(\mathcal{S}).$$

However, $\mathbf{Supp}(\mathcal{S}) \subseteq \mathbf{Supp}(x)$ and $b \notin \mathbf{Supp}(x)$, so we have a contradiction. □

This is nearly but *not quite* suitable as a definition by $\in$-induction, although if we define support first on atoms and subsets of $\mathbb{A}$—call this preliminary function $\mathbf{Supp}_p$—then we can actually make the definition

(Possible but **unused** def)

$$\mathbf{Supp}(x) \stackrel{\text{def}}{=} \begin{cases} \mathbf{Supp}_p(x) & x \in \mathbb{A} \vee x \subseteq \mathbb{A} \\ \mathbf{Supp}_p(\bigcup_{y\in x} \mathbf{Supp}(y)) & \text{otherwise} \end{cases}$$

This is *not* the definition of **Supp**, see T9.2.1.

13.2. Theory of finite sets. L13.1.1 has consequences for the theory of finite sets:

Corollary 13.2.1. *For F a finite set,*

$$a\#F \iff \forall x \in F.\ a\#F \tag{78}$$

Proof. A corollary of (Possible but **unused** def)$_{99}$, the fact that a finite union of finite sets is finite, and the fact (L9.3.2) that for X a finite subset of $\mathbb{A}$, $\mathbf{Supp}_p(X) = X$. □

And why is this interesting?

It often happens where we use variable binding that we need to "choose a fresh atom (i.e. variable name) a". The precise meaning of "choose a fresh atom a" in our new theory FM is "choose a apart from Γ, t, X, Y, Z and everything else in the context" (i.e. $a\#\Gamma, t, X, Y, Z$, see T9.4.6 or L9.4.8). We saw this in L12.4.2 for example.

C13.2.1 allows us to translate $a\#\Gamma$ to something more concrete. For example, if Γ is a finite set of pairs (x,τ) then by C13.2.1 $a\#\Gamma$ precisely when $a\#(x,\tau)$ for all $(x,\tau)\in\Gamma$. Pairset is an injective function-class so by L9.3.5 this is the case precisely when $a\#x$ and $a\#\tau$. Now variable symbols are interpreted by atoms, so x is an atom, and by L9.3.6 $a\#x$ iff $a\neq x$. Suppose for the sake of argument that the typing system is not dependent and satisfies $\mathbf{Supp}(\tau)=\emptyset$ for all types τ (as was the case in FML*tiny* in §12). Then we have "$a\#\Gamma$ iff a does not occur in Γ".

This is nothing new, we proved it all in L12.4.2. But typing contexts are more complex in Chapter IV and we use the extra power of C13.2.1 to extend the results. See C23.1.10 and L24.1.5.

We can argue similarly about "a apart from t" (i.e. $a\#t$) for t a term. Abstract syntax in FM is built up using injective function-classes[34] so for $t\in\mathcal{V}_{\mathrm{FM}}$ a semantic term (N8.2.4) we can use L9.3.5 and C9.5.9 to relate $a\#t$ to "a does not occur free in t".

Thus we may start in the sparkling springs of FM, leave for the fuming fumaroles of traditional practice, and then return, all safe in the knowledge that our mathematical rigour will neither capsize or be eaten out from beneath us.

[34]Except, note, for atom-abstraction $a.x$ (D9.5.1). This is 'injective up to α-equivalence'. We designed it to be.

Chapter III

Implementation: Isabelle/FM

14. Introduction

Having developed FM in Chapter II, there are at least two ways we may proceed.

1. We can implement an established programming language extended with facilities for handling datatypes of syntax with binding. Any programmer manipulating syntax-like structures would benefit.
2. We can implement FM set theory or a derivative of it in a theorem-proving environment, at once formally 'verifying' the mathematics and allowing formal methods to be applied to datatypes of syntax with binding in a practical setting.

The first option is big and requires collaboration between Pitts and me and practitioners requiring specific features. Dr Pitts and I have written a paper on the subject, see [**66**]. See also Chapter IV.

I have carried out this second option, the results are presented here. We shall see FM set theory implemented inside Isabelle, we call it ***Isabelle/FM***. We shall present the mathematics of the implementation (subtly different from the FM presented in Chapter II) and some of the technical considerations that shaped its design.[35]

Remark 14.2. Isabelle/FM differs theoretically from FM set theory as presented in §8 and §9, for various reasons listed below with cross-references to more extensive discussions.

1. Isabelle/FM is a little stronger than FM; it has many types of atoms. This makes it possible to attack state of the art case studies. See R16.1.2.
2. Some hacking was used. For an example see R15.5.1.
3. Isabelle/FM uses Quine atoms. See §15.2.
4. Isabelle/FM has many more constant symbols than FM. See R16.1.1.

◇

Remark 14.3 (Crude structure of Isabelle/FM)**.** We constructed FM set theory as ZFA+New (§8+§9 respectively), where ZFA is ZF set theory with an infinite set of atoms and New is the axiom $(\text{Fresh})_{35}$, which give FM its unique power and might be called the "theory of the N-quantifier".

The implementation mirrors this. Isabelle/FM was created by reengineering Isabelle/ZF to be Isabelle/ZFQA, ZF with Quine atoms (see §15.2 for a discussion

[35] Isabelle experts will probably find Chapter III slow in places. I urge tolerance. This is a feature not a bug; I want this chapter accessible to the non-expert.

of the nature of Quine atoms and why we used them), and later extending ZFQA with `New`, the theory of the $\forall$ quantifier. ◇

Remark 14.4 (Underlying system)**.** Note that as R14.3 mentions, Isabelle/FM is based on a reengineered, extended version of an existing theory Isabelle/ZF (one of the theories of the Isabelle distribution, [**37**]). The version of Isabelle/ZF from which I derived Isabelle/FM was Isabelle98-1. All the work described took place in the context of that particular release.

If Isabelle is installed on the reader's system the Isabelle/ZF files are in `$ISABELLE_HOME/src/ZF`. The precise value of `$ISABELLE_HOME` depends on the installation. It will be a directory called `Isabelle-`*version*, probably in one of the `/usr/local`, `/usr/local/share` or `/usr/share` paths. ◇

Remark 14.5 (Just-in-Time vs All-at-Once)**.** Isabelle scripts are a sequential development cycle of declaring new types and constants (a `Foo.thy` 'theory declaration' file) and proving results about them (a corresponding `Foo.ML` 'proof script' file). We must decide: shall we declare all types and constants in some `Head.thy` file and then develop the theory of the various constants in appropriate `Script01.ML` to `Script`*ab*`.ML` files? Or shall we declare types and constants as late as possible in individual `Script`*ab*`.thy` files? Call the former strategy '***All-at-once***' and the latter '***Just-in-time***'.

Concerning Isabelle/ZF, the type declarations of all standard constructs of set theory ($\bigcup$, $\emptyset$, etc) are made 'all at once' in `ZF.thy` although the axioms controlling them and more out-of-the way technical constants (such as 'less than' on ordinals or the internalised relation-set version of $\in$), are declared 'just in time'. This has the advantage of laying out all the familiar type and constant declarations in a single initial file.

In contrast Isabelle/FM follows a strict 'just in time' strategy. Constants and their controlling axioms are declared in separate `Script.thy` files and their theory developed in a corresponding `Script.ML` file. This leads to neater code and makes theory dependencies clearer (e.g. we have a rule of thumb that a result belongs in the earliest script where all its constants have been declared—and if placed it earlier, it won't compile). ◇

Remark 14.6 (Local and non-local changes)**.** We start by reengineering Isabelle/ZF to Isabelle/ZFQA so we should consider some of the issues involved when we modify a line in a `.thy` file of an existing (and extensive) implementation. This will change one or both of the statement of and proofs of results which mention the constants or axioms we modify, creating work rewriting the proof scripts which tends, fatally, to propagate exponentially through the scripts. Once a result is

changed subsequent proof-script changes and a domino effect sets in. This is to be avoided, sometimes we can:

Although Isabelle proof scripts are very technical they do tend to mirror a paper development. This manifests itself in their structure, and a script usually builds up to a fairly limited number of crucial mostly semantically significant results (e.g. introduction/elimination or unfolding rules). Although there is no real provision for 'local theorems' scripts usually observe a discipline of only using these results in the subsequent development. We shall call them ***interface results***, as opposed to ***non-interface results***, which are used locally but not in other scripts.

A classic example is `pair.ML`, the Isabelle/ZF script developing the theory of ordered pairs. Technical details of the set-theoretic implementation of ordered pairs (see `Pair_def` in Fig.15_{108} for the Isabelle/Quine version, slightly different from the Isabelle/ZF version) are completely hidden. Only implementation-independent results such as $\mathbf{fst}(a, b) = a$

```
qed_goalw "fst_conv" thy [fst_def] "fst(<a,b>) = a"
 (fn _=> [ (Blast_tac 1) ]);
```

are of interest. The implementation of Isabelle/ZFQA ordered pairs differs from that of Isabelle/ZF and this provoked work in `pair.ML` but the interfacing results were unchanged and the disturbance did not propagate.

When we modify an Isabelle theory, one of two things will occur:

1. A change propagates exponentially to all parts of the script logically 'downstream' of the original modification (usually by getting into the interface results). We call this a ***non-local*** modification.
2. A change propagates exponentially at first, but then dies out. We call this a ***local*** modification.

No change is best, but of the two, local is clearly better. I have found that a semantically innocuous change need not be local, nor need a semantically radical change be non-local. We now consider some types of modification: ◇

Remark 14.7 (Changes to avoid)**.** Experience shows that a semantically innocuous but very non-local change is the addition of an extra condition or indeed any modification to an existing condition, in an interface result.[36] Because proof in Isabelle is by resolution this extra condition provokes an extra or modified subgoal each time the modified result is used. In the best (but still very unpleasant)

[36] For example, if we add atoms to ZF set theory the axiom of extensionality *changes* from $A = B \iff A \subseteq B \wedge B \subseteq A$ to

$$(A \notin \mathbb{A} \wedge B \notin \mathbb{A}) \implies (A = B \iff A \subseteq B \wedge B \subseteq A) \ldots$$

case we must write a little bit of extra script each time to eliminate the subgoal.[37] In the worst case the extra subgoal makes clever applications of the automated theorem proving tools fail[38] and forces the user to think very hard about how these complicated applications, which he or she did not design, worked and what went wrong. If in addition this extra condition appears in one of the interface results, or *worst of all* systematically propagates to all the interface results, it is liable to spread uncontrollably and provoke a complete code rewrite.[39] Instances of this kind of problem, thankfully mostly averted, arose for example in foundation (see §15.4 and in particular R15.4.2), Quine vs Empty atoms (see R15.2.3 and R15.2.3), `wf_def` (see *p.*115), and `Memrel` (see the discussion in and around R15.5.1). My only really bad 'domino-rewrite-everything-from-scratch' incident was `rank`, discussed on *p.*119. ◇

Further examples will arise as we discuss the development.

15. Isabelle/ZFQA

Isabelle/ZFQA is Isabelle/ZF ([**59**]) re-engineered with Quine atoms (D15.2.1). We now discuss its design.

15.1. Axioms and Constants of Isabelle/ZFQA. We start by considering the constants and axioms of ZFQA, the simple theory underlying FM. Its constants are illustrated in Fig.15_{108} and Fig.16_{109}.[40] This table is adapted from the original version for ZF copied by kind permission of the author from [**59**, Section 2.2, Fig 2.1, P.22]. The significance of the various constants is discussed there. Changes and additions appear underlined.

Remark 15.1.1 (`Atm` collected)**.** The differences between ZFQA and the original ZF file are slight. We just add a constant symbol `Atm` of type i, which declares a set of atoms. A weaker declaration with `Atm` a predicate of type $i \Rightarrow o$

[37] ... so the proof of the result

$$(A, B) = (C, D) \implies \big(A = C \wedge B = D\big)$$

becomes more complicated, because we first have to verify that $(A, B) \notin \mathbb{A}$ and $(C, D) \notin \mathbb{A}$. Of course this should be easy to prove, but if five hundred results are involved it becomes a burden.

[38] For example, in Isabelle/ZF we can prove that

$$\emptyset = \big\{a \in \emptyset \;\big|\; \mathbf{False}\big\}$$

automatically. This is implicitly used in a highly automated proof of `non_mem_empty` in `ZF.ML`, right at the beginning of the implementation. It caused me endless trouble because the proof kept on breaking with no indication of the hows and whys ...

[39] ... reimplementation can become impractical with astonishing rapidity.

[40] In the actual implementation files the name of the theory has not been changed from the original 'ZF'.

ZFQA = FOL + Let +

name	*meta-type*	*description*
`Let`	$[\alpha, \alpha \Rightarrow \beta] \Rightarrow \beta$	let binder
`0`	i	empty set
`cons`	$[i, i] \Rightarrow i$	finite set constructor
`Upair`	$[i, i] \Rightarrow i$	unordered pairing
`Pair`	$[i, i] \Rightarrow i$	ordered pairing
`Inf`	i	infinite set
`Pow`	$i \Rightarrow i$	powerset
`Union Inter`	$i \Rightarrow i$	set union/intersection
`split`	$[[i, i] \Rightarrow i, i] \Rightarrow i$	generalized projection
`fst snd`	$i \Rightarrow i$	projections
`converse`	$i \Rightarrow i$	converse of a relation
`succ`	$i \Rightarrow i$	successor
`Collect`	$[i, i \Rightarrow o] \Rightarrow i$	separation
`Replace`	$[i, [i, i] \Rightarrow o] \Rightarrow i$	replacement
`PrimReplace`	$[i, [i, i] \Rightarrow o] \Rightarrow i$	primitive replacement
`RepFun`	$[i, i \Rightarrow i] \Rightarrow i$	functional replacement
`Pi Sigma`	$[i, i \Rightarrow i] \Rightarrow i$	general product/sum
`domain`	$i \Rightarrow i$	domain of a relation
`range`	$i \Rightarrow i$	range of a relation
`field`	$i \Rightarrow i$	field of a relation
`Lambda`	$[i, i \Rightarrow i] \Rightarrow i$	λ-abstraction
`restrict`	$[i, i] \Rightarrow i$	restriction of a function
`The`	$[i \Rightarrow o] \Rightarrow i$	definite description
`if`	$[o, i, i] \Rightarrow i$	conditional
`Ball Bex`	$[i, i \Rightarrow o] \Rightarrow o$	bounded quantifiers
`Atm`	i	set of atoms

CONSTANTS

symbol	*meta-type*	*priority*	*description*
`` `` ``	$[i, i] \Rightarrow i$	Left 90	image
`-`` ``	$[i, i] \Rightarrow i$	Left 90	inverse image
`` ` ``	$[i, i] \Rightarrow i$	Left 90	application
`Int`	$[i, i] \Rightarrow i$	Left 70	intersection ($\cap$)
`Un`	$[i, i] \Rightarrow i$	Left 65	union ($\cup$)
`-`	$[i, i] \Rightarrow i$	Left 65	set difference ($-$)
`:`	$[i, i] \Rightarrow o$	Left 50	membership ($\in$)
`<=`	$[i, i] \Rightarrow o$	Left 50	subset ($\subseteq$)

INFIXES

FIGURE 14. Constants of ZFQA

would allow a proper class of atoms but there are good reasons to wish `Atm` to be a set. For example it lets us define `Supp` in the way we do, see Fig.20_{122} and §16.8. ◇

Remark 15.1.2 (Safe under extension)**.** Adding a new constant is innocuous in practice, though it may be devastating in theory. E.g. in the presence of an anti-choice axiom the mere declaration of a properly axiomatised choice function makes the theory inconsistent. But previously existing proofs will not mention this new addition to the theory and will compile as before (even if we can now also prove `False`); the declaration of an additional constant cannot 'break' a theory (the same is not true of systems with a more 'active' metatheory such as Twelf, [**33**]). ◇

The axioms of ZFQA are presented in Fig.15_{108} and Fig.16_{109}. Like Fig.14_{106} they are adapted from the original [**59**, Section 2.2, Fig 2.3,2.4, P.26,27].

Most of the axioms are discussed in [**59**, Part 2.2, "Syntax of set theory"], we only discuss the points of difference from the original ZF files.

We shall start with `Atm_quine`, the axiom we add to control the elements of the new constant `Atm` discussed in §15.2. We shall use the occasion in R15.2.3 to justify implementing ZFQA and not ZFA (see D15.2.1). In §15.3 and §15.4 we discuss the two instances where existing definitions of Isabelle/ZF had to be modified in the change to Isabelle/ZFQA.

15.2. Discussion of `Atm_quine`.

Definition 15.2.1 (Quine atoms)**.** *Atoms a are **Quine atoms** when it is the case that $a = \{a\}$, and **empty atoms** when a has no $\in$-related elements (i.e. is empty, though nevertheless not equal to the empty set).*

Remark 15.2.2 (Quine)**.** "Quine atoms" are discussed by Quine in [**72**, §4, p.31]. See also R15.2.5. ◇

In FM we used empty atoms (§8). In Isabelle/FM we use Quine atoms. We call "ZF ***with empty atoms***" by the name ZFA and "ZF ***with Quine atoms***" by the name ZFQA. The slogan is

> "We choose ***empty atoms*** in Chapter II and ***Quine atoms*** in Chapter III."

`Atm_Quine` is the axiom that states that atoms in Isabelle/ZFQA are Quine:

$$a \in \mathbb{A} \iff a = \{a\}\,.$$

```
Let_def           Let(s, f) == f(s)

Ball_def          Ball(A,P) == ALL x. x:A --> P(x)
Bex_def           Bex(A,P)  == EX x. x:A & P(x)

subset_def        A <= B  == ALL x:A. x:B
extension         A = B  <->  A <= B & B <= A

Union_iff         A : Union(C) <-> (EX B:C. A:B)
Pow_iff           A : Pow(B) <-> A <= B
foundation        A=0 | (EX x:A. ALL y:x. (y~:A | x:Atm))

Atm_quine         a:Atm <-> (ALL x. x:a <-> x=a)

replacement       (ALL x:A. ALL y z. P(x,y) & P(x,z) --> y=z) ==>
                  b : PrimReplace(A,P) <-> (EX x:A. P(x,b))
```

The Zermelo-Fraenkel Axioms

```
Replace_def  Replace(A,P) ==
                   PrimReplace(A, %x y. (EX!z. P(x,z)) & P(x,y))
RepFun_def   RepFun(A,f)  == {y . x:A, y=f(x)}
the_def      The(P)       == Union({y .  x:{0}, P(y)})
if_def       if(P,a,b)    == THE z. P & z=a | ~P & z=b
Collect_def  Collect(A,P) == {y . x:A, x=y & P(x)}
Upair_def    Upair(a,b)   ==
                 {y. x:Pow(Pow(0)), (x=0 & y=a) | (x=Pow(0) &  y=b)}
```

Consequences of replacement

```
Inter_def    Inter(A) == {x:Union(A) . ALL y:A. x:y}
Un_def       A Un  B  == Union(Upair(A,B))
Int_def      A Int B  == Inter(Upair(A,B))
Diff_def     A - B    == {x:A . x~:B}
```

Union, intersection, difference

Figure 15. Rules and axioms of ZFQA

```
cons_def     cons(a,A) == Upair(a,a) Un A
succ_def     succ(i) == cons(i,i)
infinity     0:Inf & (ALL y:Inf. succ(y): Inf)
```

Finite and infinite sets

```
Pair_def       <a,b>     == {{{a},{a,b}}, 0}

split_def      split(c,p) == THE y. EX a b. p=<a,b> & y=c(a,b)
fst_def        fst(A)     == split(%x y. x, p)
snd_def        snd(A)     == split(%x y. y, p)
Sigma_def      Sigma(A,B) == UN x:A. UN y:B(x). {<x,y>}
```

Ordered pairs and Cartesian products

```
converse_def   converse(r) == {z. w:r, EX x y. w=<x,y> & z=<y,x>}
domain_def     domain(r)   == {x. w:r, EX y. w=<x,y>}
range_def      range(r)    == domain(converse(r))
field_def      field(r)    == domain(r) Un range(r)
image_def      r `` A      == {y : range(r) . EX x:A. <x,y> :  r}
vimage_def     r -`` A     == converse(r)``A
```

Operations on relations

```
lam_def    Lambda(A,b) == {<x,b(x)> . x:A}
apply_def  f`a         == THE y. <a,y> : f
Pi_def     Pi(A,B) == {f: Pow(Sigma(A,B)). ALL x:A. EX! y. <x,y>:f}
restrict_def   restrict(f,A) == lam x:A. f`x
```

Functions and general product

Figure 16. Further definitions of ZFQA

This contrasts with of the ZFA set theory of Fig.2_{26} and in particular $(\text{Sets})_{26}$ which imply

$$a \in \mathbb{A} \iff a \neq \emptyset \wedge \forall x.\ x \notin a.$$

Non-wellfounded sets?! And FM is supposed to be easy? I see the point but it is obvious to me, and I hope it will become so to the reader, that Quine atoms are

a completely innocuous form of non-wellfounded behaviour (in particular I prove consistency of ZFQA wrt ZFA in R15.2.5). Worrying about them is a red herring. Furthermore, they were a vital design choice without which Isabelle/FM would not exist. R15.2.3 and R15.2.4 below explain why.

Remark 15.2.3 (Empty atoms bad)**.** Quine atoms are the better choice for Isabelle/ZFQA. Suppose instead we implemented Isabelle/ZFA. As discussed in ft.36_{104} and ft.37_{105} the ZFA-rule for `extension` in Fig.15_{108} would have to be (something logically equivalent to)

```
BAD_extension   "A = B  <->  (A:Atm & B:Atm & A=B) |
                             (A~:Atm & B~:Atm & A <= B & B <= A)"
```

Sets that are not atoms are extensionally equal, sets that are atoms are equal precisely when they are equal. The set-equality intro rule `equalityI` would change to something like

```
originalZF equalityI   "[| A<=B ; B<=A |] ==> A=B"
newZFA     equalityI   "[| A<=B ; B<=A ; A~:Atm ; B~:Atm |] ==> A=B"
```

In the language of R14.6 this would be highly non-local. Uses of `extension` permeate the implementation and each would have to be patched. Results depending directly or indirectly on extensionality would require extra conditions to keep them true, and these modified results would in turn create work each time they are used.

Worse still, any other development building on Isabelle/ZF would be incompatible with Isabelle/ZFA. It would almost certainly use extension and hence could not be easily imported. This will not be the case with Isabelle/ZFQA. As we mentioned in R15.1.2, an extra constant `Atm` will not stop scripts that do not mention it from compiling—so long as the parts of the theory that they *do* refer to are unchanged. ◇

Remark 15.2.4 (Empty atoms terrible)**.** Empty atoms cause failure in other ways. Consider the set-theoretic $\bigcup$-operator, realised in Isabelle/ZF by `Union`. We would expect that $\bigcup\{x\} = x$ but it follows from `Union_iff` in Fig.15_{108} that for `a:Atm`, `Union({a}) = 0`.

In R15.2.3 we showed that empty atoms would provoke non-local change to `extension`, forcing work patching proof scripts and results. We now see that empty atoms cast doubt on the implementation of the very basic term-former `Union`.

The problem cannot be ignored. Consider `The`, the Isabelle/ZF ι-term-former, defined in Isabelle/ZF as follows:

```
the_def       "The(P)    == Union({y . x:{0}, P(y)})"
```

Now suppose

$$P = \lambda y.(y = a) \text{ where } a \in \mathbb{A} \quad \text{and} \quad \texttt{P = \%y.y=a} \text{ where } \texttt{a:Atm}.$$

At this point ZFA and Isabelle/ZFA diverge, since

$$\iota y.P(y) = a \qquad \text{but} \qquad \texttt{The(P)} = \texttt{Union(\{a\})} = \texttt{0}.$$

We opt for Quine atoms. ◇

Remark 15.2.5 (Pedigree of Quine Atoms)**.** 'Quine atoms' have a distinguished history. They may be unfamiliar to the reader but are not exotic. Quine discusses them in [**72**, §4, p.31], where he favours them for just that feature which so suits them for Isabelle/FM, namely leaving extensionality intact. Concerning the consistency of Quine atoms, the reader need only consider in ZFA the predicate

$$x \in' y \stackrel{\text{def}}{=} x \in y \vee (x = y \wedge x \in \mathbb{A}).$$

Any formula in the logic of ZFQA true of all models of ZFQA (call this 'valid') is mapped to a valid sentence of ZFA by replacing every $\in$ by $\in'$. In particular, if ZFQA is inconsistent, $\bot$ is one of these formulae, and it must map to a valid ZFA formula $\bot$. So long as we believe ZFA is consistent this cannot happen. ◇

15.3. Discussion of `Pair_def`. In §15.2 we argued that Isabelle/ZFQA is better than Isabelle/ZFA for implementation. That is not to say that the Quine atoms of Isabelle/ZFQA do not have a price, and part of it is that the axiom `Pair_def` must now change. We consider how, why, and the change's implications. Let us consider the original and the new.

```
originalZF   Pair_def      <a,b>   == {{a,a},{a,b}}
newZFQA      Pair_def      <a,b>   == {{{a,a},{a,b}}, 0}
```

The first point is the difference between the original Isabelle/ZF definition of pairs (rewritten in traditional notation)

$$(a, b) = \{\{a, a\}, \{a, b\}\}$$

and the standard set-theoretic implementation,

$$(a, b) = \{\{a\}, \{a, b\}\}.$$

It is true that $\{a, a\}$ and $\{a\}$ are extensionally equal, but {a,a} and {a} are different terms. The 'symmetric' form is easier to work with.[41]

Unfortunately, using the original `Pair_def` in the presence of Quine atoms breaks some interface theorems. For instance for `a:Atm`,

$$\texttt{a} = \texttt{<a,a>},$$

so two results in `pair.ML`,

[41] This claim is direct from the original Isabelle/ZF file `ZF.thy`. From my own experience I can agree.

```
Pair_neq_fst   "<a,b>=a ==> P"
Pair_neq_snd   "<a,b>=b ==> P"
```

are no longer true. I have looked and cannot find them used by the rest of the theory so it might not be affected. Some strange results would become provable though, for example

$$\texttt{Sigma(Atm,\%x.(\{x\}))} = \texttt{Atm},$$

but so long as we do not actually prove them our scripts, this need not trouble us (there is some small danger of them creeping in through an automated proof tool).

However, the loss of `Pair_neq_fst` and `Pair_neq_snd` troubled me sufficiently that I decided to modify the definition to save them by modifying `Pair_def` as shown above. This provoked considerable, but entirely local, changes in `pair.ML`.

15.4. Discussion of `foundation`. Clearly in the presence of Quine atoms such that $a = \{a\}$, the traditional formulation of foundation

$$A = 0 \ \vee \ \exists x \in A.\ \forall y \in x.\ y \notin A$$

fails. Just take $A = a \in \mathbb{A}$. We use a modified version.

Remark 15.4.1 (Different `foundation`)**.** Consider the rule `foundation` (Fig.15_{108}) written in both Isabelle and set notation.

```
foundation   "A=0 | (EX x:A. ALL y:x. (y~:A | x:Atm))"
```

$$A = 0 \ \vee \ \exists x \in A.\ \forall y \in x.\ (y \notin A \vee x \in \mathbb{A}).$$

We can read this as

> "Foundation still holds, so long as we pretend that atoms are empty."

◇

There are two subtleties to the axiom's design:

1. Why is `foundation` not written as follows?

   ```
   not_foundation1   "A=0 | (EX x:A. x:Atm | (ALL y:x.y~:A))"
   ```

   Isabelle's primary proof-method, resolution, deconstructs terms' syntax from the top down (as opposed to unfolding definitions or simplification, which act from the bottom up). It is good strategy to push changes down into the syntax. Changes to proofs are thus delayed to when the proof-state is more reduced (perhaps even solved), and when as many instantiations of unknowns as possible have been settled by the original designers' code.
2. Why is `foundation` not written as follows?

   ```
   not_foundation2   "A=0 | (EX x:A. ALL y:x.(x:Atm | y~:A))"
   ```

> An Isabelle proof-state is an ordered list of subgoals and the standard style of manual proof is to attack them in order. If `x:Atm` comes before `y~:A`, new subgoals arising from `x:Atm` will tend to appear first in the proof-state, ahead of original subgoals arising from `y~:A` for which code already exists. It is better, as in Case 1 of this list, to postpone change.

We edify this with a principle!

Remark 15.4.2 (Principle of latest change)**.** When we modify an Isabelle script it is best to provoke script changes later rather than sooner. This means that:

1. We have access to a more developed theory.
2. Changes have less time to propagate exponentially, so less results have to be modified.
3. On the level of individual proofs, the proof-state is more reduced and unknowns are more likely to be instantiated.

Point 3 is especially important in the common case that a modification provokes a number of trivial typing subgoals, e.g. `x~:Atm`. My usual strategy has been to structure the proof so they are postponed till the end of a proof and eliminate them uniformly with an automated proof tool. These tools have trouble with unknowns (they are a second-order phenomenon after all), and in the worst case may *not* fail, but *succeed* by instantiating an unknown inappropriately, which completely breaks the rest of the proof. ◇

To the casual reader R15.4.2 might not seem important, but it is a powerful strategy and underlies a whole style of programming. As such it characterises a great difference between Isabelle and paper: not only is the semantic significance of a proposition important, but at a time when theorem-proving environments are only just easy enough for humans to use, the fine details of the syntax can be decisive.

Remark 15.4.3 (Change to `foundation` non-local)**.** `foundation` is fundamental but appears relatively late in the implementation. The change to it described in R15.4.1 is non-local; everything that depends on $\in$ (a.k.a. `:`) being well-founded, such as script on rank and ordinals, must be rewritten. However, nearly all of this occurs *later*. In keeping with R15.4.2 this is, if not good, at least not too bad. And yet there is a price to pay. We consider part of it now and the nastiest effects in §15.5. ◇

In fact the first problem comes in the first script file `ZF.ML`. There is just one result stating that the empty set has no $\in$-related elements:

```
not_mem_empty   "a~:0"
```

Remarkably, this is not an axiom, but follows from `foundation` (and extensionality). See `ZF.ML` for more details. My early versions of `foundation` broke the proof. Unfortunately it is by an automated proof tool called `best_tac`. These are difficult to trace and debug and even though `best_tac` is well-documented the proof stayed broken. In fact my first version of `foundation` was semantically incorrect, but even the corrected versions did not work. It was one of the early victories of R15.4.2 that the original proof-script worked for the final version of `foundation`, whose design it guided.

Remark 15.4.4 (Debugging)**.** I have slipped into the first person and the above is essentially anecdotal. Unfortunately, writing proof-script in Isabelle can be low-level.[42] Debugging is slow and errors rarely educational. Most mistakes are made in tiredness or stupidity.

The reader should be aware and beware that at least half of the development time of Isabelle/FM—my time—was spent chasing really silly bugs. With discipline and experience, as always, one learns habits of programming that minimise the risks of creating such errors, but the system does not force these habits on the programmer in the same way that, say, ML tries to. ◇

15.5. Further discussion of `foundation`. We now come to one of the most demanding parts of the implementation of Isabelle/ZFQA. As mentioned in R15.4.3 the change to `foundation` provoked by Quine atoms has non-local effects. The sections most seriously affected are `Ordinal.ML`, dealing with the theory of ordinals, and the later `Epsilon.ML`, dealing with $\in$-induction and rank. The problem in both cases is that $\in$ ceases to be well-founded in the presence of Quine atoms. We shall consider how each proof-script was mended, starting with `Ordinal.ML`, but first we consider a case where a proof-script was *not* changed.

`WF.ML`, which deals with the theory of well-founded relations and which is used by both `Ordinal.ML` and `Epsilon.ML`, compiles without modification in Isabelle/ZFQA; the theory of well-founded relations remains the same regardless of whether $\in$ is well-founded. But now $\in$ is not well-founded because of Quine atoms and we might like to change `WF.ML`'s definition of "well-founded relation". This is introduced as a predicate `wf` of type `i=>o` in `WF.thy`, where its one argument is intended to be an internal relation-set. The Isabelle/ZF definition, unchanged in Isabelle/ZFQA, reads

```
wf_def "wf(r) == ALL Z. Z=0 | (EX x:Z.ALL y. <y,x>:r --> ~y:Z)"
```

Now suppose this were changed to:

[42] ... imperative, technical, difficult to debug, difficult to modify.

```
OTHER_wf_def "wf(r) == ALL Z. Z=0 |
                          (EX x:Z.ALL y. <y,x>:r --> (~y:Z | y:Atm))"
```

The form of this definition, it is clear, is chosen so as to make :, the Isabelle/ZFQA ∈-relation, be 'OTHER-well-founded' (cf. R15.4.1). We entertain this thought because:

1. Much of `Epsilon.ML` compiles without change (I've tried).
2. One might argue that the usual notion of well-foundedness is not so appropriate in the presence of Quine atoms.

Against this speak the following:

1. We may need the theory of well-founded relations anyway. For example, recursion on syntactic datatypes is ultimately based on well-foundedness of a relation (essentially "is a subterm of")—so we should be careful before changing it, we may need the original form.
2. Quine atoms are a convenience only and have nothing to do with FM as such. We may even wish to remove them in a future rewrite. Should we twist basic definitions to accommodate them?
3. A change to the definition of 'well-foundedness' has an impressive potential for confusion.
4. Purely in practice—and never mind everything else, this reason is telling—many of the results in `WF.ML` quote `wf_def` almost verbatim but slightly changed, for example

```
 wf_induct" "[| wf(r);
   !!x.[| ALL y. <y,x>: r --> P(y) |] ==> P(x)
|]  ==>  P(a)";

OTHER_wf_induct" "[| wf(r);
   !!x.[| ALL y. <y,x>: r --> (P(y) | y:Atm) |] ==> P(x)
|]  ==>  P(a)";
```

   These explicitly quoted instances would change with `wf_def` and propagate systematically to conditions in many interface results. This is, as discussed in R14.7, *exceedingly* undesirable.

So `WF.thy` remained unchanged. `WF.ML` compiles smoothly.

Now consider `Ordinal.ML`. Five constants are introduced in `Ordinal.thy`;

1. `:` is internalised by `Memrel::i=>i`, where `Memrel(A)` is axiomatised as a relation-set between elements of `A`.
2. 'Is a transitive set' is introduced as a predicate `Transset::i=>o`
3. 'Is an ordinal' is introduced as a predicate `Ord::i=>o`.

```
Memrel       :: i=>i
Transset,Ord :: i=>o
"<"          :: [i,i] => o  (infixl 50)
Limit        :: i=>o

Memrel_def   "Memrel(A)  == z: A*A . EX x y. z=<x,y> & x:y & y~:Atm "
Transset_def "Transset(i)== (ALL x:i. (x<=i))"
Ord_def      "Ord(i)     == Transset(i) & (ALL x:i.(Transset(x) & x~:Atm))"
lt_def       "i<j        == i:j & Ord(j)"
Limit_def    "Limit(i)   == Ord(i) & 0<i & (ALL y. y<i --> succ(y)<i)"

ZF_Memrel_def   "Memrel(A)  == z: A*A . EX x y. z=<x,y> & x:y "
ZF_Transset_def "Transset(i)== (ALL x:i. (x<=i))"
ZF_Ord_def      "Ord(i)     == Transset(i) & (ALL x:i. Transset(x))"
ZF_lt_def       "i<j        == i:j & Ord(j)"
ZF_Limit_def    "Limit(i)   == Ord(i) & 0<i & (ALL y.y<i --> succ(y)<i)"
```

FIGURE 17. `Ordinal.thy` Definitions

4. The notion of 'less than' on ordinals is introduced as a binary predicate `<::[i,i]=>o`.
5. 'Is a limit ordinal' is introduced as a predicate `Limit::i=>o`.

The definitions in `Ordinal.thy` are presented in Fig.17$_{116}$ along with the original Isabelle/ZF versions (tagged `ZF_`). Note the unusual definition of `Ord`, clearly chosen to work well in Isabelle.[43]

Atoms are ordinals according to the original ZF definition `ZF_Ord_def`; $a = \{a\}$ is a transitive set all of whose elements are transitive sets. If atoms are ordinals then desirable properties of `<` break, such as antisymmetry and irreflexivity. Typing conditions like `i~:Atm` are no option because of the non-local changes they would cause. We therefore restrict the predicate `Ord` as seen in `Ord_def` in Fig.17$_{116}$ so it is true only of the 'original' ordinals. Note that the following (logically equivalent) version,

```
BAD_Ord_def "Ord(i) == i~:Atm & Transset(i) & (ALL x:i. Transset(x))"
```

is utterly unsuitable, the difference in effort between using `BAD_Ord_def` and `Ord_def` is at least an order of magnitude. See R15.4.2.

[43] It comes from Halmos [**23**].

`Memrel` causes the second problem. In ZF the relation-set internalising $\in$ on A is always well-founded. In ZFQA it is not; consider $A = \{a\}$ for $a \in \mathbb{A}$, for which `Memrel(A)` $= \{$`<a,a>`$\}$. The proof of the following result breaks:

```
wf_Memrel   "wf(Memrel(A))"
```

This is an important interface result and a change to it would have serious non-local consequences, e.g. in `Epsilon.ML` where rank and $\in$-recursion are developed.

Remark 15.5.1 (Hack)**.** The solution I chose is a hack of which I am particularly proud. We change `Memrel` as shown in Fig.17$_{116}$. `Memrel` still thinks that atoms are empty and that $\in$ is well-founded even as `:` disagrees. A critical group of interface results, including `wf_Memrel`, remains unchanged, and this prevents most modifications from propagating. I call this a 'hack' because traditional set theorists would never *dream* of doing such a thing (nor did we in Chapter II in FM)—but they are not programming in Isabelle.[44]

However, the 'hack' has some moral justification aside from its great technical utility: as already remarked Quine atoms are orthogonal to Isabelle/FM as such. We insulate the internal universe from the details of its implementation. $\diamond$

We need to add typing conditions like `i~:Atm` in some results (for example `Transset_induct`, see `Ordinal.ML`) but the changes are local to the script and limited because most are of the form "[`Ord(i)` and/or `i<j`] imply `P`". Since `Ord(i)` and `i<j` already imply $i, j \notin \mathbb{A}$, there is usually no need to add this as an extra condition. Once these results are syntactically unchanged (even though the meanings of their constant symbols have shifted) later results whose proofs use them do not break, and the changes quickly die out.

Now we consider `Epsilon.ML`. This deals with $\in$-recursion on the set universe and rank. A somewhat edited list of the constants introduced is in Fig.18$_{118}$. We briefly consider each.

1. `eclose(A)` is the transitive closure of A under the relation-class `:`.
2. `rank(a)` is the rank of `a`. Since rank is defined by $\in$-recursion, no longer well-founded, its definition must be reconsidered.
3. `transrec` takes a set `a::i` and function `H::[i,i]=>i` and returns a set `transrec(a,H)::i`, which has the unfolding property that for $a \notin \mathbb{A}$,
   `transrec(a,H) = H(a, lam x:a. transrec(x,H))`

[44] But we used empty atoms in FM. Are then not Quine atoms a 'hack'? Indeed so! It has been suggested that both are good solutions, not hacks. I'm happy to accept that, but propose we call them 'hacks' but not 'kludges'. The real reason for this preference is that using 'hack' in connection with my Isabelle work gives it a 'je ne sais quoi' of swashbuckling romance which makes up for the 'pourquoi moi' of actually doing it. Do not begrudge me my small fantasies.

```
eclose    :: i=>i
transrec  :: [i, [i,i]=>i] =>i
rank      :: i=>i

eclose(A) == UN n:nat. nat_rec(n, A, %m r. Union(r))
transrec(a,H) == wfrec(Memrel(eclose(a)), a, H)
rank(a) == transrec(a, %x f. Union( z . y:x , (x~:Atm & z=succ(f`y))))

ZF_eclose(A) == UN n:nat. nat_rec(n, A, %m r. Union(r))
ZF_transrec(a,H) == wfrec(Memrel(eclose(a)), a, H)
ZF_rank(a) == transrec(a, %x f. UN y:x. succ(f`y))
```

FIGURE 18. `Epsilon.thy` Definitions

This is of course $\in$-recursion.

We consider `eclose` first. We can think of this as a function that unfolds a set ω times and puts all the bits in a big union. Whereas we modified `Memrel` so it thinks atoms are empty, we have not changed `eclose`. For `a:Atm`, `eclose(a)` is equal to {`a`} and not `0`. Changing `eclose` would cause an inconvenient code rewrite and leaving it seems to cause no trouble (probably because the only property of $\in$-closure we use in practice is that it *contains* the transitive closure of $\in$).

`transrec` is interesting both for its definition and for the effect that the change to `Memrel` has on its behaviour. From Fig.18_{118} we see `transrec(a,H)` is defined by well-founded recursion on

> "the restriction of [the internalisation of :] to [the transitive closure of `a`] with [the recursion function `H`]".

Remark 15.5.2. Now transitive closure of `a` is taken using : which thinks atoms are Quine, while `Memrel` the internalisation of : thinks they are empty. Clearly `transrec` could do strange things on atoms. Before we check whether this mismatch has strange consequences it is interesting to consider what the quote above actually means by comparing this construction with the pencil-and-paper development of recursion in ZF. ◇

Remark 15.5.3 (Comparison of paper and Isabelle)**.** The standard development of $\in$-recursion in ZF (not ZFQA) is as follows. We construct in the language of set theory the class of all function-sets defined on a (transitively closed) portion of the universe that satisfy an appropriate recursive unfolding rule. These function-sets, usually called ***trial functions***, correspond exactly to

`transrec(a,H)`. In the language of set-theory we then use foundation to show that all such function-sets must agree on the intersection of their domains. Hence we see that there exists a unique function-class with the unfolding property (though in the language of ZF we cannot say that).

In Isabelle all this reasoning is simply left out. The Isabelle/ZF implementation does not 'justify' $\in$-recursion on the type `i`. The implementors just extract this internal computation and write it up in Isabelle code as `transrec_def`. $\diamond$

We return to R15.5.2. `Memrel` and `:` disagree on atoms. The former considers them empty, the latter considers them Quine (see R15.5.1). Now consider the definition and the unfolding rules for `transrec`, side by side.

```
transrec_def   transrec(a,H) == wfrec(Memrel(eclose(a)), a, H)
unfold rule    transrec(a,H) = H(a, lam x:a. transrec(x,H))
```

We see that the unfolding rule uses `:` whereas the definition uses `Memrel`. There is a risk that my hack of R15.5.1 will come back to haunt me in the behaviour of `transrec` on `a:Atm`. But things resolve themselves surprisingly well. The unfolding rule on `a:Atm` is precisely what it would be if atoms were empty:

```
a:Atm ==> transrec(a,H) = H(a, 0)
```

This enables us to program the behaviour of `transrec` on elements of `Atm` with a simple boolean test on `a` in `H`.

Rank is a problem. It is defined by $\in$-recursion as follows:

```
rank   :: i=>i
rank(a)  ==transrec(a,%x f. Union({z. y:x, (x~:Atm & z=succ(f`y))}))
ZFrank(a)==transrec(a,%x f. UN y:x.succ(f`y))
```

The reader will see that the Isabelle/ZFQA version of `rank_def` is completely rewritten, in violation of R15.4.2. Both definitions start with `transrec` but this similarity is illusionary because the standard proof-method with `transrec` is to unfold it, which brings the `Union`/`UN` difference right to the top of the term. Isabelle operates on syntax top-down. If the top-level term-formers change, so do the proofs; this is a simple and brutal characteristic of the system. I *did* mend the section on rank, and this was very difficult, but it is not needed for the development of datatypes of syntax up to variable binding. We have in the end been forced to completely rewrite a definition and its proofs, but have postponed this for so long that there is very little of the theory left to worry about. The rank of atoms is set (somewhat arbitrarily) to zero.

16. Isabelle/FM

In §15 I discussed how I re-engineered Isabelle/ZF to Isabelle/ZFQA. This section discusses how this becomes Isabelle/FM.

16.1. The theory of Isabelle/FM. As briefly discussed in R14.2 the theory behind Isabelle/FM is not quite FM (§9). We now reconsider how and why.

Remark 16.1.1 (More constants)**.** First of all, and this is of no consequence, pencil-and-paper developments of set theory tend to limit the number of constants to $\in$ (and perhaps $\emptyset$), introducing extra constants as macros with the excuse of 'notational convenience' and the injunction that they don't exist, really. Isabelle/FM on the other hand is full of constant symbols. This is fundamental but need not concern us further. Cf. ft.12_{27}. ◇

Remark 16.1.2 (Many types of atom)**.** The main difference is that Isabelle/FM makes provision for many types of atoms. It is quite common to have variable symbols of different sorts, for example term and type variable symbols x and σ respectively, and Isabelle/FM provides support for this.

Recall that Isabelle/ZFQA collects all atoms into a single set `Atm::i`. In Isabelle/FM this set is partitioned into disjoint infinite sets. When the И quantifier is introduced in §16.7 it takes as argument one of these sets and picks a fresh atom from it. ◇

The rules and constants of Isabelle/FM are presented in Fig.19_{121} and Fig.20_{122}. We use Isabelle syntax. The theory is a linear development from `New_BOTTOM` where the most basic constants are declared, up to `New_TOP`, which is an empty theory documenting the dependencies and sequence of development. Following our 'just-in-time' strategy of R14.5, constants are declared only in the theory file of the script that develops their theory.

Remark 16.1.3 (`The` and `New`)**.** As discussed in §11.4 the И axioms of FM are anti-choice. There is for example no choice function from $pow_{\mathrm{fin}}(\mathbb{A})$ to $\mathbb{A}$. Fortunately the designers of Isabelle/ZF made their choice function `The` an ι-operator, which introduces no inconsistency (R11.4.3). ◇

We now discuss the files in turn and comment on their design. Recall that an Isabelle theory is a collection of pairs of a `.thy` and `.ML` file. The `.thy` file adds new constants, definitions, rules, and macros to the theory. The `.ML` file (the '***proof-script***') proves results involving them.

16.2. `New_BOTTOM`**.** This theory consists only of a `.thy` file with no associated `.ML` file. The intended denotation of `AtmPart` is the set of types of `Atm` (see R16.1.2). The associated axioms state that the elements of `AtmPart` are infinite subsets of `Atm`. We do not yet insist that they be a partition, i.e. that they be pairwise disjoint and their union be equal to `Atm`, though this becomes necessary later on in `New_DISJ`.

```
Atm       :: i
Atm_quine        "a:Atm <-> (ALL x. x:a <-> x=a)"
```

From ZFQA

```
Atmpart   :: i
AtmPart_notempty "EX X. X:AtmPart"
AtmPart_inf      "X:AtmPart ==> X~:Fin(X)"
AtmPart_elt_subset_Atm
                 "X:AtmPart ==> X<=Atm"
```

New_BOTTOM

```
Newfor    :: [i,i=>o]=>i
Newfor_def       "Newfor(X, P) == {a:X.~P(a)}"
```

New_Newfor

```
Perm      :: [i,i,i]=>i       ("(3'(_ _') ./ _)" 150)  (Sugar!)
Perm_def   "Perm(a,b,x) == if(a:Atm & b:Atm, transrec(x, %y f. (
              if(y:Atm,if(y=a,b,if(y=b,a,y)),f``y)
            )),x)"
```

Note: `Perm(a,b,x)` is sugared to `(a b).x`.

New_Perm

```
New       :: [i,i=>o] => o
           "NEW x:X. P" == "New(X, %x.P)"
New_def    "New(X,f) == Newfor(X,f):Fin(Atm)"
```

New_NEW

Figure 19. Constants of FM 1

Since `X:AtmPart` is typically an infinite and not cofinite subset of `Atm`, `AtmPart`'s structure seems to contradict C9.4.4 (subsets of $\mathbb{A}$ are finite or cofinite). In Isabelle/FM we will effectively restrict permutations to respect types `X:AtmPart` (see

```
AP        :: i => i
"#"       :: [i,i] => o   (infixl 120) (*Apartness*)
Supp      :: i => i                    (*Support*)
Supp_Fin   "Supp(x):Fin(Atm)"
Disj_def   "(a#x) == (NEW b:AP(a). (((b a).x) = x)) | a~:Atm"
Supp_def   "Supp(x) == a:Atm. ~(a#x)"
AtmPart_disj "[| X:AtmPart ; Y:AtmPart ; a:X ; a:Y |] ==> X=Y"
AP_Atm_to_AtmPart
           "a:Atm ==> AP(a):AtmPart"
AP_cont    "a:Atm ==> a:AP(a)"
AP_AtmPart_from_Atm
           "X:AtmPart ==> (EX a:Atm. X=AP(a))"
AP_off_Atm "a~:Atm ==> AP(a)=0"
```

New_DISJ

```
Abs         :: [i,i] => i          ("(3 [_] _)" 100)
Con         :: [i,i] => i          ("(3 _ <_>)" 100)
Abstraction :: [i,i] => o
Abst        :: [i,i] => i              (*The set of abstractions*)
Abs_def     "([a]x) == {<b,(b a).x>. b:{n:AP(a). n=a | n#x}}"
Con_def     "x<a> == (THE y. (<a,y>:x))"
Abstraction_def
            "Abstraction(X,x') == (EX a:X. (EX x. (x'=[a]x)))"
Abst_def    "Abst(X,U) ==
                {[a]x. <a,x>:{b:X. b#U}*U}"
```

New_Abs

No constants declared in New_ABST.

```
New_TOP = New_ABST + New_Abs + New_DISJ + New_NEW + New_Perm +
   New_Newfor + New_Atm + New_Prelim + New_BOTTOM + Finite
```

FIGURE 20. Constants of FM 2

§16.6 and §18.2), so it is subsets of X that will be necessarily finite or cofinite, not Atm.

16.3. New_Prelim. This file contains some ZF theorems that were found to be useful but do not appear in the original Isabelle/ZF distribution. Examples are

```
Upair(A,B)=Upair(B,A)                (P & (P-->Q)) <-> (P & Q)
x<=U ==> x Un (U-x) = U              X~:Fin(X) ==> X~=0
u:Fin(X) ==> u-v : Fin(X)            y:Fin(A*B) ==> domain(y):Fin(A)
```

16.4. New_Atm. The theory of the set Atm is developed. Examples are

```
Atm_inf                 Atm~:Fin(Atm)
Atm_not_0               Atm~=0
Atm_not_Atm             Atm~:Atm
FinAtm_ex_new_elt       u:Fin(Atm) ==> (EX x:Atm. x~:u)
FinAtmPart_ex_new_elt   [| u:Fin(X) ; X:AtmPart |] ==> (EX x:X. x~:u)
```

Results like Atm_not_0 and Atm_inf are part of Isabelle/FM and *not* Isabelle/ZFQA. No axioms of ZFQA prohibit Atm from being empty. Only when we add AtmPart and its axioms do we know Atm is infinite and therefore nonzero. Atm_not_Atm is difficult to prove because, using Quine atoms, it amounts to Atm~={Atm}. The automated tools tend to loop on this. This problem occurs in the early proof scripts until we develop enough of the theory of Atm to abstract away from the concrete implementation.

The last two examples are perhaps the first interesting results of Isabelle/FM. In themselves they merely rephrase Atm_inf and AtmPart_inf, but they will be used in the theory of NEW, where u will be the support of a set and x will be a fresh atom. Both results are special cases of the following more general lemma:[45]

```
[| u:Fin(Atm) ; X:AtmPart |] ==> EX x:X. x~:u
```

16.5. New_Newfor. Newfor::[i,i=>o]=>i is a technical precursor to NEW. Recall from D9.4.2 that (rephrasing slightly)

$$И x.\ P(x) \iff \left\{a \in \mathbb{A} \;\middle|\; \neg P(a)\right\} \in pow_{\mathrm{fin}}(\mathbb{A})$$

Newfor(X,P) is the set of a:X such that *not* P(a), where we intend X to be a type of atoms in AtmPart and P to be a proposition on X.

Although the results of Newfor.ML are quite simple, for example

```
Newfor(X,P) <= X
Newfor(X , %x.(P(x) & Q(x))) <= Newfor(X,P) Un Newfor(X,Q)
Newfor(X,P) <= Newfor(X,%x.(P(x) & Q(x)))
```

they will later translate directly to more impressive results about И. E.g. the second one is a distributivity lemma for И over logical conjunction (cf. C9.4.5).

[45] This has Atm on the LHS. FinAtmPart_ex_new_elt does not.

16.6. `New_Perm`**.** The real work starts here. The permutation action of atoms is defined by $\in$-recursion precisely as in D8.1.8 in FM (except that the permutation is restricted to transpositions from the start, cf. D9.1.1. Note that `Perm` ignores the partitioning of `Atm` and we can permute any two atoms whether or not they are in the same `X:AtmPart`. There is an argument for restricting `Perm`, it is made in §18.2. See also R16.6.2. ***Note also that we sugar*** `Perm(a,b,x)` ***to*** `(a b).x`.

Remark 16.6.1 (Technical difficulty)**.** Directed proof-search with `Perm` is rather problematic:

1. In the very earliest results we unfold the definition of `a:Atm` as `a={a}` and automated tools loop on this. Just imagine what this axiom would do to the sanity of a substitution-based simplifier (cf. §16.4). This problem disappears as soon as we abstract away from the implementation of atoms.
2. Because $(a\ b)\cdot x = (b\ a)\cdot x$ it can sometimes occur that we have $(a\ b)\cdot x$ in the assumptions and $(b\ a)\cdot x$ in the conclusions. We can't just put their equality into the search space because it causes looping. By careful structuring of the results I was able to avoid this.
3. The real problem is that $(a\ b)\cdot x = y \iff x = (a\ b)\cdot y$. Again, we cannot add this to the search space because the tools would loop. Such situations occur repeatedly (see R16.6.3 below) and are a real problem. I discuss my solutions to this problem in §17.

Sadly our results often cry out for automation:

1. `Perm` is defined by cases on elements of `Atm` (consider its nested if-then-else structure in Fig.19_{121}) so proofs of elementary results which unfold `Perm`s definition are often by cases. If three or even four permutations occur together (for examples see the list below, especially `Perm_commute`), cases can multiply.
2. The Isabelle/FM analogues of the commutativity results of R8.1.13 are an important class of results. Unfortunately equivariance as proved in T8.1.10 for ZFA and FM *cannot* be proved in Isabelle (see R16.6.3). The next best thing is a uniform automated proof method. See §17.

$\diamond$

Results of `New_Perm` are of roughly three kinds. We consider two of them here, and the third in R16.6.3. Most of the proofs of the following displayed one or both of the difficulties discussed in R16.6.1.

```
Perm_abx_x      "[| x:Atm ; x~=a ; x~=b |] ==> (a b).x = x"
Perm_Atm_to_Atm "x:Atm ==> (a b).x :Atm"

Perm_idem       "(a b).(a b). x = x"
Perm_idem_twist "(a b).(b a). x = x"
Perm_ab_ba      "(a b).x = (b a).x"
Perm_commute    "(a b).(c d).x = (c d). ( ((c d). a) ((c d). b) ).x"
```

The standard proof technique in the first two results is by cases on elements of `Atm`. For the rest we consider `x:Atm` first[46] and then use $\in$-induction. There is nothing unexpected here. The devil is in the execution and the sheer volume of lemmas.

Remark 16.6.2 (Isabelle/ZFQA and Isabelle/FM)**.** Permutation `Perm` ignores `AtmPart`. So `New_Perm` belongs to ZFQA more than FM. However, the line dividing *Isabelle*/ZFQA and Isabelle/FM is more implementational: Isabelle/ZFQA is the modified Isabelle/ZF and Isabelle/FM carries on from there. However, see §18.2. ◇

Remark 16.6.3 (Equivariance results)**.** Now we come to the ***equivariance*** or ***commutativity results***. Together they represent a significant fraction of the developmental effort. They are precisely the commutativity results of R8.1.13. Now in ZFA this was a corollary of a general result T8.1.10, which was proved by induction on the syntax of the language of ZFA set theory starting from some basic constants which were obviously equivariant. But the datatype of the syntax of Isabelle terms is of course not an Isabelle type, so we cannot do this![47] So these results must be proved individually (this is hard). The Isabelle/ZFQA analogue of $(13)_{30}$ must be proved symbol by symbol, for example the case $\in$ is called `Perm_epsilon` (Fig.21_{126}) and proved by $\in$-induction. ◇

Examples of equivariance results are given in Fig.21_{126}. See §19.2 for a concrete instance where we need an equivariance result in a case study. Their proofs present these difficulties:

1. They are numerous, one for each constant.
2. They are often difficult to automate (see R16.6.6 and also §17).
3. Some commutativity results cannot be stated. That of `RepFun` should be
   `(a b).RepFun(A,f) == RepFun((a b).A,(a b).f)`

[46] We prove this special case using a standard tactic I developed to split it into a multitude of subcases according as `a` and `b` are [1. not atoms], [2. atoms not equal to `x`], or [3. atoms equal to `x`], and then by automation with a standard library of basic results about permutation on atoms. Without this technique `Perm_commute` would not bear thinking about.

[47] ... but what we can do is add equivariance as an axiom. See §20.

```
Perm_0       "(a b).0 = 0"
Perm_Pair    "(a b). <x,y> = <(a b).x,(a b).y>"
Perm_cross   "(a b). (A*B) = (((a b).A) * ((a b).B))"
Perm_domain  "(a b).domain(h) = domain((a b).h)"
Perm_ext     "(a b).x = {(a b).y . y:x}"
Perm_epsilon "x: (a b).y <-> (a b).x : y"
```

FIGURE 21. A few equivariance results of Isabelle/FM

but this is badly typed because `f::i=>i` is a function and `Perm(a,b)`, being of type `i=>i`, cannot be applied to it. A new proof is needed for each instantiation of `f` in the code. Similar problems occur with replacement, `The(P)`, `UN x:A. B(x)`, `Sigma(A,f)`, and similar.[48] The list also includes `lfp(a::i,h::i=>i)` and `transrec(a::i,H::[i,i]=>i)` with *disastrous* consequences for our Isabelle/FM theory of datatypes, because we cannot prove general equivariance results for datatypes and functions out of them respectively. See §18.3 and §19.1.

4. They are inexhaustible. We frequently declare new constants and must prove equivariance for each. Sometimes this is trivial by unfolding definitions. Sometimes, bearing in mind point 3 above, it is not.

Remark 16.6.4 (`Perm_Collect`)**.** We take a moment to continue Item 3 above and mention the interesting halfway case of `Perm_Collect`, which should be

```
(a b).Collect(A,P) == Collect((a b).A,(a b).P)
(a b).x:A. P(x)  == {x:(a b).A . ((a b).P)(x)}
```

(the second version uses syntactic sugar). But `Collect` types as `[i,i=>o]=>i`, which is bad (see Item 3 above). Still, we *can* prove this:

```
Perm_Collect  "(a b).{x:A . P(x)} = {y . x:A, P(x) & y=(a b).x}"
```

The RHS uses the replacement term-former `Replace::[i,[i,i]=>o]=>i`. The term means

$$(a\ b)\cdot\left\{x \in A \;\middle|\; P(x)\right\} = \left\{y \;\middle|\; \exists x \in A.\ P(x) \wedge y = (a\ b)\cdot x\right\}.$$

So we have a rule to pull permutations inside collections, but the top-level term former changes from `Collect` to `Replace`. This change will invalidate any

[48] Note however that `Perm_domain` in the list above does appear to apply permutation to a function `h`. Of course here `h::i` is an internal function-*set*.

subsequent parts of an existing proof, if we are modifying one, and complicates proofs we write ourselves.[49] ◇

Remark 16.6.5 (Two notions of equivariance)**.** There are two possible notions of equivariance for a (nullary) constant `C`:

```
[| a:Atm ; b:Atm |] ==> (a b).C=C  and
                        [| a:X ; b:X ; X:AtmPart |] ==> (a b).C=C
```

Because permutations need not respect types of atoms (we can quite happily transpose `a` and `b` in `C` for `a:X` and `b:Y` of distinct types `X,Y:AtmPart`) these are not the same. E.g. `C=Atm` fulfils both and `C=Y:AtmPart` fulfils only the second. We can be confident that constants inherited from ZF set theory will be equivariant in the unconditional sense. More sophisticated objects such as `Supp` and `Disj` (§16.8) are not. This whole issue would vanish if we made permutation respect types of atoms, see §18.2. ◇

Remark 16.6.6 (Proof method for equivariance)**.** Suppose we have expressions e_1, e_2 with `Perm` at the top and bottom respectively and we want to prove e_1=e_2. The standard proof method is to unfold definitions and push `Perm` inside e_1 using a library of results.

Unfortunately, if the constants are axiomatised so that there is nothing to unfold, or if one of them has an argument of function type so that the equivariance result could not be stated (see point 3 of the list above), we must use extensionality. At this point we encounter all the difficulties of proof search with `Perm` discussed in R16.6.1. This happens sufficiently often to be a programming burden. We discuss the matter further in §17. ◇

Remark 16.6.7 (Irrelevant behaviour vital)**.** Having discussed why this proof-script is difficult, let us consider why it is not even harder. Isabelle has no subtyping, so `Perm` is typed as `[i,i,i]=>i`: `a` and `b` can not only be different types of atom, they can be any set. Results must account for this, e.g. with typing conditions like `a:Atm`. So for example, this would occur:

```
      ZF_Perm_subset "X<=Y <-> (a b).X <= (a b).Y"
OTHER_Perm_subset "[|a:Atm ; b:Atm|] ==> X<=Y <-> (a b).X <= (a b).Y"
```

The problem is twofold. First, these conditions spawn subgoals with resolution. Second, the automated proof tools include a simplifier (called `Simp_tac`) which rewrites arbitrarily deep inside a term using a supplied library of equalities and logical equivalences. This tool is unique and important because it does not, like all other facilities in Isabelle, operate strictly top-down on terms. But it does

[49] See §19 and in particular *p.*147. See also R16.9.1 and the discussion following it.

```
NEW_iff  "(NEW a:X. P(a)) <->
                    (EX u:Fin(Atm). (ALL a:X. (a~:u --> P(a))))"
NewI     "[| u:Fin(Atm) ; (ALL a:X. (a~:u --> P(a))) |] ==>
                                                (NEW a:X. P(a))"
NewD     "[| (NEW a:X. P(a)) |] ==>
                    (EX u:Fin(Atm). (ALL a:X. (a~:u --> P(a))))"
New_conj "[| (NEW a:X. P(a)) ; (NEW a:X. Q(a)) |] ==>
                                        (NEW a:X. (P(a) & Q(a)))"
```

FIGURE 22. Major results of `New_NEW`

not handle *conditional* rewrite rules such as `OTHER_Perm_subset` quite as well as unconditional ones.

The reader without first-hand experience of Isabelle should understand that this is important. The choice of the 'irrelevant' behaviour of `Perm` literally determined the practicality of the entire enterprise. The two obvious candidates for badly types `a` or `b` are to make `%x.Perm(a,b,x)` the constant empty set function or the identity on `i`. The latter proved more convenient, virtually eliminating typing conditions. cf. §18. ◇

16.7. `New_NEW`**.** We come to the theory of the И quantifier, although we cannot get very far because the critical axiom corresponding to $(\text{Fresh})_{35}$ appears in the later script §16.8. `New_NEW` merely establishes introduction and elimination rules for `NEW`, but this is not entirely straightforward.

Consider the syntax in Fig.19_{121}. `New` is declared taking as arguments `X:i` and `P::i=>o`. We intend that `X:AtmPart` and that `P` is a proposition on `X`, so `New(X,P)` picks a fresh atom `a:X` and tests whether `P(a)`.

`New` is implemented in terms of existing constants:

```
New(X,P) == Newfor(X,P):Fin(Atm)
```

clearly corresponding to D9.4.2 in FM on *p.*40. Note that we do not use `Newfor(X,P):Fin(X)`. Since `Newfor(X,P)<=X<=Atm` this is equivalent, but the other version is much more convenient.

The four major results of the script are given in Fig.22_{128}. `NEW_iff` simply axiomatises the definition of `New`. `NewI` and `NewD` are clearly derived from it and are suitable for intro and destruct rules. `New_conj` is a basic result, part of C9.4.5.

We consider `New_conj` first. It is very useful:

Remark 16.7.1 (New_conj very useful)**.** Suppose a subgoal has just one condition resolving with New. Then we apply destruct resolution with NewD to simplify the condition to

```
EX u:Fin(Atm). (ALL a:X. (a~:u --> P(a)))
```

and proceed from there by elim-resolving with the bounded existential quantifier to produce a constant unknown u.

Now suppose the subgoal has two or more such conditions. If we apply NewD and eliminate EX in each individually we will have distinct unknowns u1, u2, etc. But using New_conj we could make these unknowns equal. This extremely useful technique is programmed into a simple ML-function New_conj_tac.[50] ◇

Now to the most powerful tool so far.

Remark 16.7.2 (NewI_tac)**.** Suppose a conclusion of a subgoal resolves with (the conclusion of) NewI. Resolution produces two subgoals with conclusions u:Fin(Atm) and (ALL a:X. (a~:u --> P(a))) for P suitably instantiated by the resolution.

The standard proof-method, very successful, is this:

1. Instantiate u to the union of all Supp(x) where Supp is support (T9.2.1 and §16.8) and the x are the free variables of type i appearing in the proposition resolving with P.
2. Solve the first subgoal by a standard automated method.
3. Simplify the second subgoal with ballI and impI (which intro-resolve against ALL and -->).

The ML-function NewI_tac automates this. The technical details are complex (and interesting) and correspond only in an abstract sense to the recipe shown above.

If P resolves with a term containing higher-order unknowns this proof-method fails because Supp::i=>i cannot be applied to them in Step 1. In this case we use one of simpler precursors to NewI_tac.

Corresponding functions exist for NewD, for example NewD_tac. ◇

We can now manipulate NEW almost as conveniently as ALL and EX.

16.8. New_DISJ**.** Isabelle/FM takes shape here. We introduce many types of atoms, #, the atom-apartness relation (#, see N9.2.4), Supp, support (**Supp**, see T9.2.1), and the finite support property Supp_fin (see R9.1.3) which is equivalent to the crucial FM-axiom $(\text{Fresh})_{35}$.

[50] Destruct-resolution with New_conj obtains a new condition from two old ones, but also leaves one of the old conditions hanging around littering the proof-state. It's just the way the rules of resolution work out. What New_conj_tac does is tidy up.

We introduce axioms asserting that `AtmPart` partitions `Atm`.[51] `AP(a)` is that unique `X:AtmPart` such that `a:Atm` is in `X`. We arbitrarily set `AP(x)=0` when `x~:Atm` (cf. R16.6.7), so for *all* `a`:

```
AP_subset_Atm "AP(a)<=Atm"
AP_iff        "a:Atm <-> a:AP(a)"
a_APc_eq      "a:AP(c) ==> AP(a)=AP(c)"
a_APc_c_Atm   "a:AP(c) ==> c:Atm"
a_APc_a_Atm   "a:AP(c) ==> a:Atm"
```

Remark 16.8.1. We now have various characterisations of atoms `a:Atm`: `a:Atm`, `a:AP(a)`, `a:AP(n)` and `n:AP(a)` for any `n`.[52] ◇

We introduce `a#x`, apartness. `#` queries `AP` for the type of `a` and therefore does not take it as an argument. If `a~:Atm` then we set `a#x` to `True`. Otherwise, we use the following definition (see Fig.19_{121})

```
NEW b:AP(a). ((b a).x = x)
```

corresponding to the following characterisation of #:

$$a\#x \iff \text{И}b \in \mathbb{A}.\ (b\ a)\cdot x = x.$$

This is an old friend last seen in L9.5.7. Results proved include

```
Disj_Atm_char  "x:AP(n) ==> a#x <-> a~=x"
Disj_subsetAtm "X:Fin(Atm) ==> a#X <-> a~:X"
Disj_Perm_fix  "[|a#x ; b#x ; a:AP(c) ; b:AP(c)|] ==> (a b).x = x"
Perm_Disj      "[|a:AP(n) ; b:AP(n)|] ==> c#x <-> ((b a).c)#(b a).x"
Disj_Perm_I    "[|(c=b & a#x) | (c=a & b#x) |
              (c~=a & c~=b & c#x) ; a:AP(n) ; b:AP(n)|] ==> c#(a b).x"
```

The first two results characterise apartness for the special cases `Atm` (recall the characterisation of `a:Atm` of R16.8.1) and `Fin(Atm)`. The standard proof method with such results is to break them into two implications, of course, and then do the following with each subgoal.

1. Unfold the definition of `#`,
2. Use `NewD_tac` and `NewI_tac` (in that order to get variable dependencies right) to simplify to a subgoal in FOL (without `New`).
3. Solve using standard methods.

Remark 16.8.2 (Overall strategy)**.** When proving a result involving `New` (which any result involving `#` implicitly does) we face a difficulty. Suppose we wish to intro-resolve against `New x.P(x,y,z)`. This *should* reduce to

```
[| x#y ; x#z |] ==> P(x,y,z)
```

[51] Elements pairwise disjoint and union equal to `Atm`.

[52] There is also the low-level `a={a}`, which we avoid. It is implementation-dependent and bad for proof-search.

but by resolution it is *impossible* to extract the free variables of P.

The standard method is therefore to (unfold # and then) eliminate New immediately using the programs developed in New_NEW. They extract the set of free variables by inspecting the syntax of a subgoal with an ML-program and we can then proceed normally. ◇

We saw Disj_Perm_fix in FM in L9.2.7. Its Isabelle/FM proof is far harder and forced me to develop much of New_NEW, as well as the strategy of R16.8.2. Because of a#x and b#x we also need New_conj_tac (see R16.7.1). The typing conditions a:AP(c) and b:AP(c) insist a and b be in the same type X:AtmPart of atoms. Otherwise the conclusion does not follow.[53]

Perm_Disj is another equivariance result (see R16.6.3).

Disj_Perm_I is an example of a family of useful intro and elim rules.

We now develop a library of results like these:

```
Disj_Atm     "a#Atm"
Disj_Un_I    "[| x#A ; x#B |] ==> x # (A Un B)"
Disj_Upair_I "[| x#A ; x#B |] ==> x # Upair(A,B)"
Disj_succ_I  "a#x ==> a#succ(x)"
```

and like these:

```
Disj_succ_D  "a#succ(x) ==> a#x"
Disj_Upair_D "x#Upair(A,B) ==> x#A & x#B"
Disj_sum_D1  "a#(X+Y) ==> a#X"
```

Note that some likely-looking results are not true:

```
NOT TRUE, X=Atm-{a} Y={a}   "a#X Un Y ==> a#X & a#Y"
NOT TRUE, X=Atm             "[| a#X ; x:X |] ==> a#x"
```

We have considered # a great deal, what about Supp? Supp(x) is simply the atoms of any type not apart from x, collected into a set (were Atm a proper class this would be impossible, see R15.1.1). We could define Supp as follows:

```
NOT_Supp_def    "NOT_Supp(X,x) == {a:X. ~(a#x)}"
    Supp_def    "Supp(x)       == {a:Atm. ~(a#x)}"
```

So NOT_Supp takes as argument an X:AtmPart and returns only atoms of that type. This is a needless complication which we consider no further. Because Supp is defined just using #, its theorems are directly derived from the theory of #.

Remark 16.8.3 (Pairs of constants)**.** Supp and # are a natural pair. In §16.9 we shall see Abs and Con (atom abstraction and concretion). On paper we tend to define such pairs separately, prove their duality, and develop their properties in parallel. In Isabelle, because of the constructive style, we tend to choose one of the pair as '***dominant***' and define the other in terms of it. The standard method with

[53] In §18.2 I discuss a better way of setting things up that would eliminate the typing conditions.

a defined constant is to unfold its definition, so the theory of the non-dominant of the pair becomes a list of corollaries. ◇

Remark 16.8.4 (Different development)**.** `New_DISJ` and the paper development in §9 differ in their sequence of development. It is roughly

$$\text{Perm} \longrightarrow \mathbf{Supp} \longrightarrow \# \longleftarrow \text{И} \longleftarrow \text{Perm}$$

on paper[54], but

$$\texttt{Perm} \longrightarrow \texttt{New} \longrightarrow \texttt{\#} \longrightarrow \texttt{Supp}.$$

in Isabelle. Notions of elegance sometimes diverge between paper and Isabelle. ◇

16.9. `New_Abs`**.** The final declarations of Isabelle/FM are made here. Atom abstraction and concretion `Abs::[i,i]=>i` and `Con::[i,i]=>i` are introduced (D9.5.1 and D9.5.14) along with sugar `[a]x` and `x<a>`, a predicate `Abstraction::i=>o` which is true precisely on sets of the form `[a]x` (N9.5.3), and the abstraction-set set former `Abst::i=>i` (D9.6.1).

Notice that `Abs_def` completely differs from the FM-definition D9.5.1. We prove the two versions equivalent in L9.5.11. We see the discussion of `Abs_twiddle` on *p.*134 below.

We seem to have developed the theory sufficiently that paper and Isabelle converge. In previous files most results were quite tedious. This file is full of interesting ones many of which are familiar from Chapter II. A selection is in Fig.23_{133}.

The first group concerns the general theory of abstractions: an intro and destruct rule, and an example of a typical result (L9.5.4 living in Isabelle/FM). Because of the typing `Abs::[i,i]=>i` we need `Abs_off_Atm` to control the case `a~:Atm`.

`Perm_Abs` is the inevitable equivariance result. As with many of the later equivariance results the proof is extremely complex.[55] Just for once, we consider it in full. See Fig.24_{134}. This works, but it is clearly unstructured nonsense. It took days to write, and with each fundamental change to the the theory (one of them is discussed in §18 below) or the `claset`, the proof broke and took days to fix.

---

[54] $a\#x$ is defined as $a \notin \mathbf{Supp}(x)$. We define И independently and connect it to # in L9.5.7.

[55] For me, this is the indication that there's something wrong with the Isabelle/FM treatment of equivariance. The proof of a basic result should get *simpler* as the theory develops. When the reverse happens, there's something wrong. See §18.2 and §20.

```
Abstraction_func_on_fst "[| Abstraction(AP(a),x') ;
                                        <a,x>:x' ; <a,y>:x' |] ==> x=y"
Abstraction_I "[| x'=[a]x ; a:AP(n) |] ==> Abstraction(AP(n),x')"
Abstraction_D "[| Abstraction(AP(n),x') |] ==> EX a:AP(n). (EX x. x'=[a]x)"
Abs_off_Atm  "a~:Atm ==> [a]x=0"
Perm_Abs   "[| b:AP(n) ; a:AP(n) |] ==> (b a).([c]x) = [(b a).c] ((b a).x)"
Pair_in_Abs_char "[| a:AP(n) ; b:AP(n) |] ==>
                         <a,x>:[b]y <-> ((a=b & x=y) | (a#y & x=(a b).y))"

Abs_twiddle  "[|<a,x>:[n]u ; <b,y>:[n]u|] ==> NEW c:AP(n).(c a).x=(c b).y"
Abs_twiddle' "[|<a,x>:[n]u ; <b,y>:[n]u ; AP(n)=AP(n')|] ==>
                                        NEW c:AP(n').(c a).x=(c b).y"

Abs_relocate_binder "[|a#x ; a:AP(n) ; b:AP(n)|] ==> [a]((a b).x) = [b]x"

Abs_eq_Pair_I  "<a,x>:[b]y ==> [a]x=[b]y"
Abs_eq_Perm_D  "[| [a]x=[b]y ; a:AP(n) ; b:AP(n) |] ==> y=(a b).x"
Abs_func_snd_D "[| [a]x = [a]y ; a:AP(n) |] ==> x=y"

Disj_Abs_a    "a#([a]x)"
Disj_Abs_char "a:AP(n) ==> b#([a] x) <-> (b#x | b=a)"
Supp_Abs_eq   "a:AP(n) ==> Supp([a]x)=Supp(x)-a"
Abs_eq_Disj_I "[| a#x ; b#x ; a:AP(n) ; b:AP(n) |] ==> [a]x=[b]x"
Abs_eq_New_D  "[| [a]x=[b]y ; a:AP(n) ; b:AP(n) |] ==>
                                        NEW c:AP(n). (c a).x = (c b).y"
Abs_eq_Int_I  "[| u:[a]x ; u:[b]y |] ==> [a]x=[b]y"

Abstraction_Disj_char_as_Pair  "Abstraction(AP(a),x) ==>
                                                a#x <-> (EX y. <a,y>:x)"
Abstraction_New_D "Abstraction(AP(n),x) ==> (NEW a:AP(n). (EX y. x=[a]y))"

Con_Abs_Perm_b  "[| b#x ; b:AP(a) |] ==> ([a]x)<b> = (b a).x"
Con_x_Pair_in_x "[| a#x ; Abstraction(AP(n),x) ; a:AP(n) |] ==> <a,x<a>>:x"
Perm_Con        "[| a:AP(n) ; b:AP(n) |] ==>
                                    (a b).(x'<r>) = ((a b).x')<(a b).r>"
Disj_Con_I      "[| n#a ; n#x'|] ==> n#(x'<a>)"
```

FIGURE 23. Results of New_Abs

The standard technique with a complex proof is to simplify the problem to simpler results. But the problem is already simple (that is, the statement of Perm_Abs is not long or compound): the *proof* is complex, and being so unstructured has no obvious modularisation.

So we try to find a simpler lemma that gives us a better handle on Perm_Abs, perhaps some characterisation of equality between abstractions [a]x. Pair_in_Abs_char was the result, but its proof reduces, slowly and horribly, to Perm_Abs.

Remark 16.9.1 (Grit our teeth)**.** Sometimes we must unfold all the definitions and take the consequences. With skill we can minimise the work but it will always be unpleasant. This pattern—a simple result which is difficult to prove and

```
Goalw [Abs_def] "[| b:AP(n) ; a:AP(n) |] ==>
                            (b a).([c]x) = [(b a).c] ((b a).x)";
by (asm_full_simp_tac (Perm_simpset()) 1);
br equalityI 1;
by (REPEAT (
        (SOMEGOAL (eresolve_tac [RepFunE,conjE,CollectE,disjE]))
 ORELSE (SOMEGOAL (resolve_tac
          [subsetI,RepFun_eqI,Pair_iff RS iffD2,conjI,CollectI]))
 ORELSE (SOMEGOAL hyp_subst_tac) ));
br (Perm_commute RS trans) 14;
basm 14;
by (REPEAT (fast_tac (claset()
   addSIs [Perm_commute_reversed RS sym RS trans RS sym]
   addIs (Perm_LEFT_SIs())@[Perm_Disj_I,Perm_idem RS sym]
   addSDs (Perm_LEFT_SDs())@[Perm_Disj_RL,Perm_idem_epsilon_LL]) 1));
qed "Perm_Abs";
```

FIGURE 24. Proof of `Perm_Abs`

apparently impossible to break up into simpler problems—is a recurring theme I observe in the initial stages of an Isabelle development. ◇

By now many sets are defined by collection or replacement. Because `Perm_collect` and `Perm_Replace` are inelegant (see R16.6.4), equivariance results are particularly prone to suffer from R16.9.1, and this is part of the reason they are such a nuisance (see R16.6.3).

Now we move on to `Abs_twiddle`. Recall from D9.5.1 that an FM abstraction set $a.x$ is an equivalence class of (a, x) under $\sim$, where $\sim$ is the relation

$$(a, x) \sim (a', x') \stackrel{\text{def}}{=} \mathsf{N}b.\ (b\ a)\cdot x = (b\ a')\cdot x'.$$

This is no good for Isabelle. The equivalence class is not a set and cannot be expressed as an element of `i`. We can implement it as `P::[i,i,i,i]=>o` where `P(a,x,a',x')` is suitably axiomatised, and then prove {`x.P(x)`} (for well-typed arguments!) is a set. In order to do *this* though we would either have to exhibit the set `U` directly and prove it extensionally equal to {`x.P(x)`}, or concoct some argument using the theory of universes `univ(Atm)` and collection.

It is clearly simpler to take `U` as the definition. The reader will see from `Abs_def` in Fig.20_{122} that we do just that, using the concrete characterisation of abstraction we saw in L9.5.11.

Remark 16.9.2. `Abs_twiddle` and `Abs_twiddle'` (Fig.23_{133}) are one half of the proof that U and the domain of truth of P are equal. Although semantically interesting they are of no practical use because proof in Isabelle is by resolution and we do not often encounter conclusions of the form `NEW c:AP(n).(c a).x = (c b).y`.

Intro and elim rules are useful. They are implementation-independent, provide standard methods for simplifying goals, and (if well-designed) give good automated proof-search. They tend to be the real interface results (see R14.6) and `Abs_eq_Pair_I` to `Abs_func_snd_D` are cases in point. ◇

Remark 16.9.3. `Abs_relocate_binder` (Fig.23_{133}) is a special case of an equivariance result. It is pivotal (used in the proofs of `Abs_eq_Pair_I` and the very important `Disj_Abs_a`). I had terrible trouble with `Disj_Abs_a` and `Abs_eq_Pair_I` and obtained horrible proofs for them. It seemed to be a case of R16.9.1. Then I realised I could deduce both using `Abs_relocate_binder`. The proof is still nasty but we only have to do it once. ◇

`Disj_Abs_a` is L9.5.6 and an important intro rule. For once, the Isabelle/FM and FM proofs are not dissimilar. We use `Abs_relocate_binder`. It is one part of `Disj_Abs_char`, which itself has a long and involved proof and is precisely C9.5.9. With these two results the apartness behaviour of atom-abstraction `[a]x` is under control. `Supp_Abs_eq` is easy to prove and semantically satisfying, but not immediately useful. The same holds of `Abs_eq_Disj_I` and `Abs_eq_New_D`.

`Abs_eq_Int_I` is rather interesting. It provides an intro rule for equality of abstractions alternative to `Abs_eq_Pair_I` and related results.

The other results are fairly straightforward. `Abstraction(X,x)` means "`x` is an abstraction of type `X`". Concretion is hardly used. Firstly, it only makes sense to write `x<c>` when `x` is an abstraction, and if we know this we can replace `x` by `[c']x'` and simplify `x<c>` to `(c' c).x'`. Secondly, we are designing these files to provide support for abstract datatypes. Most reasoning with abstract datatypes is by resolution, which amounts to pattern-matching. Patterns match only with values, which can be of the form `[a]x` but not `x<a>`.

The 'irrelevant' behaviour of abstraction and concretion (R16.6.7) is not addressed in the files at all, because it turns out not to be an issue.

16.10. `New_ABST`**.** In §16.9 we developed abstraction and concretion. In this final file we develop abstraction types (D9.6.1). The difficult files were `New_Abs`, `New_DISJ` and `New_Perm`. `New_ABST` is easy by comparison. Some of the more interesting results are:

```
Perm_Abst  "[| a:AP(n) ; b:AP(n) |] ==>
                  (a b).Abst(AP(n'),X) = Abst(AP(n'),(a b).X)"
Abst_nosupport "x#X ==> x#Abst(AP(n),X)"
Con_I "[| x:Abst(AP(n),X) ; a#X ; a#x ; a:AP(n) |] ==> x<a>:X"
Abst_bad_typing "n~:AP(n) ==> Abst(AP(n),X)=0"
Abst_mono "X<=Y ==> Abst(AP(n),X) <= Abst(AP(n),Y)"
Abst_E "[| x:Abst(AP(n),X) ;
           !!b y.[| b:AP(n) ; y:X ; x=[b]y ; b#X |] ==> Q
         |] ==> Q"
```

Perm_Abst is the equivariance result, its proof is dreadful. Note that we do not have AP((a b).n') on the RHS because AP(n') is equivariant (whether n' is an atom or not). Abst_nosupport is actually a useful intro-rule needed for the theory of datatypes with binding. Con_I is useless in practice. Abst_bad_typing controls the irrelevant behaviour. Abst_mono is L10.2.5 and needed for the datatypes package (see monos in Fig.25_{144}), presumably for the same reasons we proved the theorem in FM. Abst_E seems to be useless in practice.

16.11. The theory of finite sets. We conclude the discussion of Isabelle/FM proper. The appropriate FM notion of 'finite set' differs from the usual ZF one—namely that the set has finite cardinality. The theory of cardinalities is badly hit by the addition of atoms and FM finite support arising from $(\text{Fresh})_{35}$ because, although Atm may be countably finite from the outside, it cannot be bijected with Nat or any other ordinal inside the set-universe (we can think about the support of the bijecting function-set and create a proof similar to that of T11.4.1).

A finite set can still be bijected with a finite subset of Nat with a function-set of finite support, so we could pursue a theory of "cardinalities of finite sets". Instead I chose to characterise 'X is finite' by X:Fin(X), that is $X \in pow_{\text{fin}}(X)$, and developed a script of results such as

```
[| Y<= X ; X:Fin(X) |] ==> Y:Fin(Y)
```

In Isabelle finite sets are inductively defined, so I use this theory to prove L13.2.1 and its corollary L24.1.5. I used this to implement the beginnings of Chapter IV. However, for the reasons discussed in R19.3.1 (not to mention limits of time), I have not implemented all of Chapter IV.

17. Automation and Perm

Having described the structure of Isabelle/FM I shall indulge myself by describing one a simple idea in proof-technique that made a big difference. Recall from Fig.19_{121} and §16.6 that the permutation action is defined by $\in$-recursion on i just as in FM it is defined on $\mathcal{V}_{\text{FM}}$ in D8.1.8.

Permutation appears mostly in equivariance results, which are numerous. How do we program the automated proof tools to handle a typical result? Consider Perm_Union from New_Perm.ML:

```
Perm_Union "(a b). Union(A) = Union ((a b).A)"
```

Isabelle/FM has nothing like T8.1.10 (see R16.6.3) so as with any other goal we must dissect its syntax. In this case this means using extensionality. First we split the equality into two set inclusions, then we reduce the set inclusions to implications:

```
1. !!x. x:(a b).Union(A) ==> x:Union((a b).A)
2. !!x. x:Union((a b).A) ==> x:(a b).Union(A)
```

The problem is the same for any term-former ?Con, in this case ?Con = Union: we have a lot of results for conclusions and assumptions of the form x:?Con(?A), but almost none for x:(a b).?Con(?A). The solution is simple, we (somehow) pull all permutations to the left of : as shown:

```
1. !!x. (a b).x:Union(A) ==> x:Union((a b).A)
2. !!x. x:Union((a b).A) ==> (a b).x:Union(A)
```

Now everything is of the form ?x:Union(?A) and we can apply standard intro and elim rules to simplify the proof-state to this:

```
1. !!x B. [| (a b).x:B ; B:A |] ==> ?B7(x,B):(a b).A
2. !!x B. [| (a b).x:B ; B:A |] ==> x:?B7(x,B)
3. !!x B. [| x:B ; B:(a b).A |] ==> ?B9(x,B):A
4. !!x B. [| x:B ; B:(a b).A |] ==> (a b) . x:?B9(x,B)
```

Again we pull permutations to the left, and the rest is trivial.

So I developed a collection of libraries with which to program this algorithm (and others as required) into the automated proof tools. A standard example is

```
Perm_epsilon_LR "(a b).x:y ==> x:(a b).y"
```

If we intro-resolve this against a conclusion it "pulls Perm to the left". Note also if we destruct-resolve against an assumption it "pulls Perm to the right".[56]

The other major term-former is equality =. There the issue is slightly different because the proof-tools will substitute for x in a subgoal if they encounter an assumption of the form x=?A, but not of the form (a b).x=?A. The algorithm is therefore to pull permutation to the *right* in assumptions. In conclusions we may prefer permutation on left, because the automated simplifier works with theorems of the form [|Subgoals|] ==> Complex = Simple. So I developed libraries of (elim)

[56]Hence Perm_epsilon_LR is actually in two libraries: Perm_LEFT_SIs and Perm_RIGHT_SDs. LEFT and RIGHT denote the direction the transposition travels. S is a technical flag meaning 'safe' (see [**58**, §11, 'The Classical Reasoner'], in particular §11.4, first paragraph) I stands for 'intro' and D for 'destruct'.

rules to pull permutation to the right in assumptions and (intro) rules to pull it to the left in conclusions.

This can cause conflict if we need to resolve a conclusion (with Perm on the left) with an assumption (with Perm on the right). In a future code rewrite I should give some thought to this.

We find similar libraries useful for, say #. For example:

```
Perm_Disj_RL "[| c#(b a).x ; a:AP(n) ; b:AP(n) |] ==> (b a).c#x"
Perm_Disj_LR "[| (b a).c#x ; a:AP(n) ; b:AP(n) |] ==> c#(b a).x"
```

Used appropriately they can pull permutation to the left or right of an assumption or conclusion of the form c#(b a).x or (b a).c#x. Note that the typing conditions a:AP(n) and b:AP(n) will spawn two typing subgoals with each application. This is a real nuisance, I discuss how to eliminate it in §18.2.

We might use these to solve this subgoal 6:

```
6. !!x' aa xa.
      [| a#x; a:AP(n); b:AP(n); <aa, xa>:[a](a b).x; aa:AP(a);
         aa#(a b).x; aa=a; aa~=b |]
      ==> <aa, xa>:[b]x
```

(which arises with 15 other similar subgoals just before the end of the proof of Abs_relocate_binder, see Fig.23_{133}). We substitute b for aa, and then pull the permutation from acting on x on the right of the sixth assumption (where it does no good at all), to the left. We have

```
!!x' aa xa.
   [| a#x; a:AP(n); b:AP(n); <a, xa>:[a](a b).x; a:AP(a);
      (a b).a#x; a~=b |]
   ==> <a, xa>:[b]x
```

We can then simplify (a b).a to b, so we have a#x and b#x in the assumptions. We have a result

```
Disj_Perm_fix  "[| a#x ; b#x ; a:AP(c) ; b:AP(c) |] ==> (a b) .x=x"
```

This allows us to simplify <b,xa>:[a](a b).x to <b,xa>:[a]x. The conclusion is <b,xa>:[a]x so the subgoal is solved. In this way all sixteen cases can be handled automatically by blast_tac with Perm_Disj_RL in the destruct rules and Perm_Disj_LR in the intro rules.[57]

[57] Isabelle experts may wonder how we can simplify <b,xa>:[a](a b).x to <b,xa>:[a]x using pure resolution, with Disj_Perm_fix as shown, in blast_tac. I wrote a couple of two-line ML functions which take a list of simplification rules (like Disj_Perm_fix) and a list of contexts (like "%x.?y:[?a]x") and return a long list of intro (or elim) rules that implement all possible simplifications within the possible contexts. You just pass this list to blast_tac, et voilà, simplification with a tableau theorem prover! Has it been invented before? It was terribly useful, maybe it should be provided as a standard facility.

Perhaps the reader thinks this trivial? In a way it is, but what a difference is makes! Twelve-line proofs collapsed to one-liners overnight. Consider the proof of `Perm_Union`:

```
Goal "(a b). Union(A) = Union ((a b).A)";
br equalityI 1;
by (auto_tac (Perm_claset(),Perm_simpset()));
qed "Perm_Union";
```

Here `Perm_claset()` is a standard library of intro- and elim-rules implementing the strategies above. `Perm_simpset()` is a standard library of *simplification* rules (whose design I shall not discuss), for example `(a b).(a b).x=x`. Clearly this proof is quite robust and standardised. It works just as well for `Perm_Upair`, `Perm_Un`, `Perm_cons` and many more.

18. Alternative approaches

In §15 and §16 we described Isabelle/FM as it is. Of course this misses interesting ideas which for whatever reason were *not* included. Let us now consider some of them.

18.1. Meta-level types of atoms. We go back to the moment we have finished Isabelle/ZFQA and are about to begin Isabelle/FM. We intend to develop the theory of permutation and partition `Atm::i` into many types (R16.1.2). How? The following strategy was the one I originally adopted. It was extensively developed, but then *discarded and does not exist* in the source code. `New_BOTTOM.thy` declares a new class

```
classes
  atmc < term
```

Recall that the type of sets `i` is of arity `term`, so this declares a collection of types `atmc`, each of which is on a par with `i` in the Isabelle meta-system. We intend `'a::atmc` to be types of atoms, implemented as separate Isabelle types in their own right. We declare a polymorphic casting function `@::'a::atmc=>i` along with suitable axioms for injectivity and so forth. It is convenient also to declare an inverse to the casting function `-@`.

```
"@"   :: 'a::atmc => i
"-@"  :: i => 'a::atmc

Cast_inj      "(@a)=(@b) ==> a=b"
Cast_inv_char "@-@a=a"
Cast_to_Atm   "@a:Atm"
```

Functions in Isabelle `f::a=>b` are total. If `f` is badly-behaved off its intended domain we may in theory need typing conditions in its results, which is inconvenient (see R14.7 and R16.6.7). With meta-types of atoms `Perm` need not be typed as

```
Perm :: [i, i, i] => i
```

which leaves great possibility for inappropriate input in the first two arguments which 'should morally be atoms' but may be anything. Instead we can declare

```
Perm :: ['a::atmc,'a::atmc,i] => i
```

There is now no question of inappropriate input.

Results that might need typing conditions, such as

```
Perm_bij              "[| a:Atm ; b:Atm |] ==> x=y <-> (a b).x = (a b).y"
Perm_Diff             "[| a:Atm ; b:Atm |] ==>
                                        (a b).(X-Y) = ((a b).X) - ((a b).Y)"
Disj_Perm_interpolate "[| b#x ; c#x ; b:AP(n) ; c:AP(n) ; a:AP(n') |] ==>
                                                  (b c). (c a). x = (b a).x"
```

lose their typing conditions (although the meta-level typing conditions look messy)

```
Perm_bij            "x=y <-> (a::'a::atmc b::'a::atmc).x = (a b).y"
Perm_Diff           "(a::'a::atmc b::'a::atmc).(X-Y) = ((a b).X) - ((a b).Y)"
Disj_Perm_interpolate "[| (b::'n::atmc)#x ; (c::'n::atmc)#x |] ==>
                                        (b c). (c a). x = (b a::'m::atmc).x"
```

and less conditions is good (R14.7 and R16.6.7).

So why is this bad? Isabelle's meta-language is too weak to really support it. For example, the useful result

```
[| n1~=n2 ; a:AP(n1) ; b:AP(n1) ; a':AP(n2) ; b':AP(n2) |] ==>
                                (a b). (a' b'). x = (a' b'). (a b). x
```

cannot be rephrased as

```
'n1::atmc~='n2::atmc ==>
          (a::'n1 b::'n1). (a'::'n2 b'::'n2). x = (a' b'). (a b). x
```

because there is no equality between meta-level types `'n1::atmc`, `'n2::atmc`.

`Supp` and related functions have problems. For example, Isabelle has no subtyping so we cannot collect all `'a::atmc` into a supertype `Atoms`. `Supp` must be polymorphic over `'a::atmc` and no longer returns the global support. Furthermore, types cannot be passed directly as arguments to functions. We use indirect and inelegant means:

```
Supp :: ['a::atmc=>i,i] => i
```

Here the first argument is a dummy function (probably `@`) that instantiates `'a::atmc` in `Supp`.

Isabelle does not display type information by default. Type information can be switched on, but then all variables are completely typed and even simple results become unreadable. Debugging was very difficult indeed.

I switched once I had realised I could manipulate the 'irrelevant' behaviour of `Perm::[i,i,i]=>i`. By making `Perm(a,b)` the identity for `a` and `b` not both atoms we get the best of both worlds. In §18.2 below I discuss how I might make the 'irrelevant' behaviour even better.

18.2. Restrict permutation to types of atoms only. Recall that Isabelle/FM has many types of atoms (R16.1.2). Isabelle/ZFQA does not; that is, `Perm` as implemented does not respect them and will permute atoms of one type for another. We could move types of atoms to Isabelle/ZFQA, i.e. extend the 'irrelevant' behaviour (R16.6.7), to make `Perm(a,b)` the identity for `a,b:Atm` not of the same type `X:AtmPart`.

This probably would not introduce extra typing conditions (for example into `Perm_commute`) because the irrelevant behaviour of `Perm` is so well-behaved. The definition of `Perm` would become something like:

```
CURRENT_Perm_def
  "Perm(a,b,x) == if(a:Atm & b:Atm, transrec(x, %y f. (
      if(y:Atm,if(y=a,b,if(y=b,a,y)),f``y)
    )),x)"
TYPED_Perm_def
  "Perm(a,b,x) == if(EX X:AtmPart. (a:X & b:X), transrec(x, %y f. (
      if(y:Atm,if(y=a,b,if(y=b,a,y)),f``y)
    )),x)"
```

The bounded existential quantifier would complicate proofs in `New_Perm.ML` that unfold `Perm_def`, but the effects would be local. I have in fact tried a similar approach. I moved `AP` from `New_DISJ.thy` to `New_Perm.thy` and used it to implement `Perm` as follows:

```
ALTERNATIVE_Perm_def
  "Perm(a,b,x) == if(a:AP(b) , transrec(x, %y f. (
      if(y:AP(y) , if(y=a,b,if(y=b,a,y)) , f``y)
    )),x)"
```

I developed this quite extensively but then dropped it. `ALTERNATIVE_Perm_def` was a *total* disaster because of the '`a:AP(b)`'. Permutation is symmetric on its first two arguments and this condition caused parity difficulties in automated proof-search. That is, the proof-state reduced to subgoals of the form

```
b:AP(a) ==> a:AP(b)
```

We cannot just add `a:AP(b)==>b:AP(a)` as an intro-rule, it is instant death to proof-search through looping.[58] Of course we could always replace the offending clause with something symmetric like `EX n.a:AP(n) & b:AP(n)`, but then we might as well use `TYPED_Perm_def`.

What I did not appreciate is the reward in the later theory where we can drop typing conditions in many important results. E.g.

```
AP_Perm_I     "[|a:AP(n) ; b:AP(n) ; c:AP(n')|] ==> (a b).c:AP(n')"
Disj_Perm_fix "[|a#x ; b#x ; a:AP(c) ; b:AP(c)|] ==> (a b).x = x"
Perm_Disj     "[|a:AP(n) ; b:AP(n)|] ==> c#x <-> ((b a).c)#(b a).x"
Abs_relocate_binder
              "[|a#x ; a:AP(n) ; b:AP(n)|] ==> [a] ((a b).x) = [b]x"
Perm_Abs      "[|b:AP(n) ; a:AP(n) |] ==>
                                  (b a).([c]x) = [(b a).c] ((b a).x)"
```

simplify to

```
              "c:AP(n') ==> (a b).c:AP(n')"
              "[|a#x ; b#x|] ==> (a b).x = x"
              "c#x <-> ((b a).c)#(b a).x"
              "a#x ==> [a] ((a b).x) = [b]x"
              "(b a).([c]x) = [(b a).c] ((b a).x)"
```

respectively.

So `Perm` should be made to respect types of atoms in the next code rewrite.

18.3. Extend permutation to function types. Recall from §16.6 we commented that equivariance results for some constants of Isabelle/FM cannot be stated. Consider for example what the equivariance rule should be for `transrec::[i,[i,i]=>i]=>i` (which is discussed on *p.*118).

```
(a b).transrec(t,H) = transrec((a b).t,(a b).H)
```

This is badly typed because `%x.(a b).x::i=>i` and `H::[i,i]=>i`, so `(a b).H` is badly-typed. Recall that when we developed FM set theory on paper we built the permutation action by $\in$-induction just as we did in Isabelle/FM, but then found it useful to extend it to functions and predicates (D8.1.8 and the subsequent text). The action was

$$(a\ b)\cdot f = \lambda x.(a\ b)\cdot f((a\ b)\cdot x) \tag{79}$$

So why don't we extend `Perm` similarly? Well, there is room for discussion how we will handle the extension itself. Do we just define it for functions `f::i=>i` like so;

```
Perm' :: [i,i,i=>i]=>(i=>i)
Specific_axiom  "Perm'(a,b,f::i=>i)(x)=Perm(a,b,(f(Perm(a,b,x))))"
```

or do we try something cleverer and more general like

[58] I was thinking of this when I wrote Item 2 of R16.6.1.

```
Perm' :: [i,i,i=>'a]=>(i=>'a)
HO_axiom   "(Perm'(a,b,f::i=>'a))(x)=Perm'(a,b,f(Perm'(a,b,x)))"
LO_axiom   "Perm'(a,b,x::i)=Perm(a,b,x)"
```

But this does not actually matter because it brings us nothing. To avoid details we argue very abstractly. Suppose we generalise `Perm` to all types by lifting from `i` as above. Then instead of writing the above, we can do as we did in $(9)_{29}$ and write

```
(a b).transrec = transrec
```

(Isabelle/FOL gives us equality on higher-order types). `Perm` on higher types is defined by its action at lower types based on `i` so to prove the statement above we must apply `transrec` to arguments `(a,H)`, unfold definitions, and so on. The only thing we can do with a function is apply it to arguments, so if we get anywhere it must necessarily be to a commutativity result of the form of those in Fig.21_{126}, where permutation acts only on elements of `i`. Of course if our goal contains function variables we can't even do that.

Now let us be more concrete and illustrate this with the specific case of `transrec`. Suppose we have established enough theory to allow us to state the attractive formula

```
(a b).transrec(t,H) = transrec((a b).t,(a b).H)
```

as a goal in Isabelle. We have to prove it. The proof-method for `transrec` is unfolding, and (assuming for simplicity `t~:Atm`) we obtain

```
(a b).H(t, lam x:t. transrec(x,H)) =
   ((a b).H) ((a b).t, lam x:t. transrec(x,(a b).H))
```

We now have no option but to unfold the definition of permutation on `H` and obtain

```
(a b).H(t, lam x:t. transrec(x,H)) =
   (a b).H(t, (a b).(lam x:t. transrec(x,(a b).H)))
```

We skip the details, but this reduces to

```
H(t, lam x:t. transrec(x,H)) =
   H(t, lam x:t. transrec((a b).x,(a b).H))
```

Since `H` is a function variable symbol, we have no option but to take the last step of reducing this to

```
(a b).transrec(t,H) = transrec((a b).t,(a b).H)
```

and we are back where we began.

If of course we are trying to prove the statement above for a particular instantiation of `H` to `%x y.T` then we do not have to take this last step. However, we might as well just reformulate the initial equivariance result to avoid applying `Perm` to `H` in the first place.

```
consts
  aTerm :: i
  term  :: i

datatype <= "univ(Atm)"
  "term" =      tVar ("a: AP(aTerm)")
              | tApp ("a: term", "b: term")
              | tAbs ("a: Abst(AP(aTerm),term)")
  monos      "[Abst_mono]"
  type_intrs "[Supp_Atm_0, subset_refl,AP_subset_Atm RS subsetD]"
  type_elims "[make_elim (rotate_prems 2 (Abst_in_univ RS subsetD))]"

end
```

FIGURE 25. Declaration of `term`

In fact I more-or-less go through the loop above (for `lfp` not `transrec`) in §19.2 trying to prove in that subsection's notation that `(a b).term=term`.

I tried to extend `Perm` to the type `i=>i` as in $(79)_{142}$ and developed the theory, but I discovered it was pointless just as described above. Now if permutation were taken as primitive over all types, of course, with some appropriate axiomatisation, then this story would completely change—and I would have been spared the discomfort of the equivariance results. See §20.

19. The λ-calculus in Isabelle

After much work the stage is set for the raison d'être of Isabelle/FM: datatypes.[59]

The untyped λ-calculus has been our running example until now, so let us use the power of Isabelle/FM to actually declare a datatype Λ_α (a.k.a '**L**', last seen D10.3.4) as an Isabelle datatype `term`. All the following takes place in scripts named `Term.thy` and `Term.ML`.

19.1. `Term.ML`**.** The Isabelle code is in Fig.25_{144}. First we declare a set `aTerm::i`, intended to be an atom `aTerm:Atm` so that `AP(aTerm)` is its type of atoms

[59]The Isabelle/ZF datatypes package is under continuous development so the version my implementation inherited from Isabelle98-1/ZF (R14.4) may differ from any the reader may be using.

and can serve as a set of syntactic variable symbols.[60] We do not add an axiom `aTerm:Atm` so `AP(aTerm)` may be empty (see §16.8). Such an axiom can of course be added later if needed.

We then declare the set `term::i` of terms. The `datatype` declaration calls an ML program referred to as "***the Isabelle/ZF datatypes package***" which axiomatises `term` as an appropriate least fixedpoint and proves a few of its basic properties using a default method over which the user has some control via the lists `monos`, `type_intrs` and `type_elims`. I shall call this the ***user-interface*** for this section. For edited versions of these properties see Fig.26_{146}.

Remark 19.1.1 (Hack trace)**.** The datatypes package displays the usual problem of automation in Isabelle—no good error messages or trace—but its code is quite readable. Obviously the unfamiliar abstraction set-former `Abst(AP(aTerm),term)` in `term` (Fig.25_{144}) confused the program and I had to figure out what results to put in the user-interface to make it work again. Most of my learning curve was understanding the code sufficiently to rewrite it to produce a crude trace[61] and see what had to be done. It is a tribute to the design of the package that I *could* make it handle abstraction-types using the interface and proof-method provided, i.e. without hacking code except to get enough trace to understand the power of the tool. ◇

The results of Fig.26_{146} are enough for us to inductively reason on `term`. However, with no functions yet defined out of it (e.g. substitution), we have few interesting theorems to prove except one:

```
Perm_term  "[| a:AP(n) ; b:AP(n) |] ==> (a b).term = term"
Disj_term  "a#term"
Supp_term  "Supp(term)=0"
```

These are three results but only technically so. They are all logically equivalent (quite easily, see Fig.27_{149}) and I shall collectively call them "equivariance of `term`".[62]

Because of the dependencies of the definitions in Isabelle/FM, proofs of the second and third results must reduce to a proof of the first so we restrict attention to `Perm_term`. There are two ways of proceeding:

---

[60] I should have made `aTerm::AtmPart`. Taking `aTerm::i` was a strategic error and should be changed in the next code rewrite. See R20.2.

[61] Perhaps the designers of Isabelle would like to include this facility in future versions?

[62] The ideas of §18.2 (make `Perm` respect types of atoms) would pay off here. `Perm_term` would lose its typing conditions `[|a:AP(n) ; b:AP(n)|]` because `Perm(a,b)` would be the identity for `a`, `b` badly typed. This would simplify subsequent proofs, see R16.6.7.

```
["term == lfp(univ(Atm),
    %X. {z: univ(Atm) .
        (EX a.   z=tVar(a) & a:AP(aTerm)) |
        (EX a b. z=tApp(a, b) & a:X & b:X) |
        (EX a.   z=tAbs(a) & a:Abst(AP(aTerm), X))})"] : thm list
```

term.defs

```
["term_case(?f_tVar, ?f_tApp, ?f_tAbs) ==
  case(?f_tVar,
    case(split(%v. ?f_tApp(v)), ?f_tAbs))",
 "tVar(?a)     == Inl(?a)",
 "tApp(?a, ?b) == Inr(Inl(<?a, ?b>))",
 "tAbs(?a)     == Inr(Inr(?a))"] : thm list
```

term.con_defs

```
"term <= univ(Atm)" : thm
```

term.induct

```
"[| ?x:term;
    !!a. a:AP(aTerm) ==> ?P(tVar(a));
    !!a b. [| a:term; ?P(a); b:term; ?P(b)|] ==> ?P(tApp(a, b));
    !!a. a:Abst(AP(aTerm), Collect(term, ?P)) ==> ?P(tAbs(a)) |]
       ==> ?P(?x)" : thm
```

term.dom_subset

```
["[| tVar(?a) = ?c; ?c = Inl(?a) ==> PROP ?W |]
       ==> PROP ?W",
 "[| tApp(?a, ?b) = ?c; ?c = Inr(Inl(<?a, ?b>)) ==> PROP ?W |]
       ==> PROP ?W",
 "[| tAbs(?a) = ?c; ?c = Inr(Inr(?a)) ==> PROP ?W |] ==> PROP ?W",
 "Inl(?a) = Inr(?b) ==> ?P",
 "Inr(?b) = Inl(?a) ==> ?P",
 "[| Inl(?a) = Inl(?b); ?a = ?b ==> PROP ?W |] ==> PROP ?W",
 "[| Inr(?a) = Inr(?b); ?a = ?b ==> PROP ?W |] ==> PROP ?W",
 "[| <?a, ?b> = <?c, ?d>; [|?a = ?c; ?b = ?d |] ==> ?R |]
      ==> ?R"] : thm list
```

term.free_SEs

```
["?a : AP(aTerm) ==> tVar(?a) : term",
 "[| ?a : term; ?b : term |] ==> tApp(?a, ?b) : term",
 "?a : Abst(AP(aTerm), term) ==> tAbs(?a) : term"] : thm list
```

term.intrs

FIGURE 26. Results of Datatypes Package for `term`

1. Unfold definitions repeatedly (beginning with `term.defs`) and use a library of equivariance results (which we saw in FM in R8.1.13 and again in Isabelle/FM in R16.6.3) to reduce the problem to triviality. This approach is elegant and fails, we discuss why below.
2. Use extensionality to reduce the problem to two subproblems:
   ```
   Perm_term_elt1 "[| a:AP(n) ; b:AP(n) |] ==> x:term ==> (a b).x:term"
   Perm_term_elt2 "[| a:AP(n) ; b:AP(n) |] ==> (a b).x:term ==> x:term"
   ```
   In fact we can be clever and use idempotence of permutation to prove `Perm_term_elt2` from `Perm_term_elt1` by a single automated tactic (see Fig.27$_{149}$). We still have to prove `Perm_term_elt1`. Proof by hand is out of the question because of the number of cases involved, even for this simple datatype. Automated proof is not easy either. We shall return to the proof below.

So why does option 1 fail? We unfold `term` using `term.defs` and obtain a subgoal of the form

```
[| a:AP(aTerm) ; b:AP(aTerm) ; x:lfp(univ(Atm),h) |]
  ==> (a b).x:lfp(univ(Atm),h)
```

We now use results from `New_Perm` to pull the permutation into the least fixedpoint operator.

```
WRONG [| a:AP(aTerm) ; b:AP(aTerm) ; x:lfp(univ(Atm),h) |]
        ==> x:lfp((a b).univ(Atm),(a b).h)
```

I have developed the theory of universes in Isabelle/FM and so we can reduce `(a b).univ(Atm)` to `univ(Atm)`. The `(a b).h` however is badly-typed and nonsense: `Perm(a,b)::i=>i`[63] cannot be applied to `h::i=>i` as discussed in Item 3 on *p.*125.

Our only option is to unfold the definition of `lfp` using

```
lfp_def "lfp(D,h) == Inter({X:Pow(D). h(X)<=X})"
```

and hope that the problem goes away (`Inter` is the Isabelle/ZF version of $\bigcap$). Unfortunately it does not. We can use

```
Perm_Inter "(a b).Inter(A) = Inter((a b).A)"
```

but we then we come up against `Collect` (written here in sugared notation)

```
(a b).{X:Pow(D). h(X)<=X}
```

In R16.6.4 we remarked that this term-former is somewhat problematic. This is a case in point. When we apply `Perm_Collect`[64] we obtain

```
[|a:AP(aTerm) ; b:AP(aTerm) ; x:Inter({X:Pow(univ(Atm)). h(X)<=X})|]
  ==> x:Inter({Y. X:Pow(univ(Atm)), h(X)<=X & Y=(a b).X})
```

[63] Recall that `(a b).u` is sugar for `Perm(a,b,u)`. So `Perm(a,b)`, sugared, is `(%u.(a b).u)::i=>i`.

[64] `Perm_Collect "(a b).{x:A. P(x)} = {y.  x:A, P(x) & y=(a b).x}"`

see R16.6.4.

To stop things getting too technical we switch into ordinary set-theoretic notation. We need to show that (omitting typing conditions on a and b and the set-universe typing condition in the collection term),

$$\forall X.\ (x \in \bigcap \{X \mid h(X) \subseteq X\}) \ \wedge\ (h(X) \subseteq X \wedge x \in X) \implies x \in (a\ b)\cdot X$$

But of course $x \in (a\ b)\cdot X \iff (a\ b)\cdot x \in X$ and if we think about it the equation above is just

$$x \in \texttt{term} \implies (a\ b)\cdot x \in \texttt{term},$$

which is `Perm_term_elt1` above. We have gained nothing! We shall discuss this further in §20.

We have shown that option 1 of the list above reduces in Isabelle/FM to option 2, which amounts to proving the result

```
Perm_term_elt1 "[| a:AP(n) ; b:AP(n) |] ==> x:term ==> (a b).x:term"
```

by induction on `x:term`. We accordingly use `term.induct` and obtain the following:

```
> be term.induct 1;
Level 1
[| a:AP(n); b:AP(n); x:term |] ==> (a b).x:term
 1. !!aa. [|a:AP(n); b:AP(n); aa:AP(aTerm)|] ==> (a b).tVar(aa):term
 2. !!aa ba.
       [| a:AP(n); b:AP(n); aa:term; (a b).aa:term; ba:term;
          (a b).ba:term |]
       ==> (a b).tApp(aa, ba):term
 3. !!aa. [| a:AP(n); b:AP(n);
             aa:Abst(AP(aTerm), {u:term . (a b).u:term}) |]
         ==> (a b).tAbs(aa):term
```

The first two subgoals are not difficult; We can expand the definitions of `tVar` and `tApp` using `term.con_defs` and then use standard equivariance techniques. The third subgoal is a little harder because the set inside the `Abst` is not just `(a b).term`. Nevertheless there is a simple automated proof of it. With this result it is fairly simple to derive `Perm_term`. The full code is in Fig.27_{149}.

Having settled equivariance there is little else for `Term.ML` to do. We prove a few injectivity results of the form

```
tApp_inj "tApp(x,y) = tApp(x',y') ==> x=x' & y=y'"
```

by a standard automated method because they're useful later, and that's about it.

19.2. Discussion of `Term.ML`. `Perm_term` (Fig.27_{149}) is important in various ways, we consider just one. Suppose we have `t:term` and `a:AP(aTerm)`. We can easily prove `tLam([a]t):term`. Suppose further that `c:AP(aTerm)` and `c#[a]t`. We

```
Goal "[| a:AP(n) ; b:AP(n) |] ==> x:term ==> (a b).x:term";
be term.induct 1;
by (ALLGOALS (asm_full_simp_tac (simpset() addsimps
                                  [Perm_tVar,Perm_tApp,Perm_tAbs])));
by (blast_tac (claset() addSIs term.intrs@[AP_Perm_abc_epsilon]) 1);
by (blast_tac (claset() addSIs term.intrs@[AP_Perm_abc_epsilon]) 1);
by (auto_tac (claset() addSIs (Perm_RIGHT_SIs()) addSDs (Perm_RIGHT_SDs())
                       addSIs term.intrs,simpset() addsimps [Perm_Abst]));
br (Abst_mono RS subsetD) 1;
ba 2;
by (blast_tac (claset() addSIs (Perm_LEFT_SIs())) 1);
qed "Perm_term_elt1";

Goal "[| a:AP(n) ; b:AP(n) |] ==> (a b).x:term ==> x:term";
br (Perm_idem RS subst) 1;
by (blast_tac (claset() addSIs (Perm_LEFT_SIs()) addIs [Perm_term_elt1]) 1);
qed "Perm_term_elt2";

Goal "[| a:AP(n) ; b:AP(n) |] ==> (a b).term = term";
by (blast_tac (claset() addSIs (Perm_LEFT_SIs()) addSDs (Perm_LEFT_SDs())
                        addDs [Perm_term_elt1,Perm_term_elt2]) 1);
qed "Perm_term";

Goalw [Disj_def] "a#term";
by (DisjI_jamie_tac 1);
by (NewI_tac 1);
by (blast_tac (claset() addIs [Perm_term,AP_cont]) 1);
qed "Disj_term";

Goalw [Supp_def] "Supp(term)=0";
by (asm_full_simp_tac (simpset() addsimps [Disj_term]) 1);
qed "Supp_term";
```

A few notes:

`term.intrs` is proved by the datatypes package, see Fig.26_{146}.

`Perm_tVar` etc. are equivariance results proved by a standard automated method.

`Perm_RIGHT_SIs` etc. are standard libraries, see §16.6.

See §20 for my ideas how to reduce this proof to a few lines.

FIGURE 27. Code of an equivariance result

shall certainly encounter situations where `([a]t)<c>=(c a).t` is of interest, e.g. evaluating `tApp(tLam([a]t),c)`. But we cannot sensibly discuss `(c a).t` until we know `(c a).t:term`. We deduce this by pulling the permutation to the right of `:` and using `Perm_term`. Other examples abound.

The case study continues. I have constructed various inductively defined functions out of `term`, including a size function, a free variables function, and a substitution function illustrated in Fig.28_{150}.

```
consts
  Size   :: i
inductive
  domains "Size" <= "term * nat"
  intrs Atm_I "a:AP(aTerm) ==> < tVar(a),1 >:Size"
        App_I "[| <x,sx>:Size ; <y,sy>:Size |] ==>
                                    <tApp(x,y),sx #+ sy #+ 1>:Size"
        Abs_I "[| <x,sx>:Size ; a:AP(aTerm) |] ==>
                                        <tAbs([a]x),sx #+ 1>:Size"
```

SIZE

```
consts
  FV   :: i
inductive
  domains "FV" <= "term * Fin(AP(aTerm))"
  intrs FV_Atm_I "a:AP(aTerm) ==> <tVar(a),a>:FV"
        FV_App_I "[| <x,fx>:FV ; <y,fy>:FV |] ==>
                                          <tApp(x,y),fx Un fy>:FV"
        FV_Lam_I "[| <x,fx>:FV ; a:aTerm |] ==>
                                           <tAbs([a]x),fx-a>:FV"
```

FREE VARIABLES

```
consts
  Sub   :: i
  n     :: i
  t     :: i
inductive
  domains "Sub" <= "term * term"
  intrs AtmI1 "[| a=n ; a:AP(aTerm) ; t:term |] ==> <tVar(a),t>:Sub"
        AtmI2 "[| ~(a=n) ; n:AP(aTerm) ; t:term |] ==>
                                             <tVar(n),tVar(n)>:Sub"
        AppI  "[| <x,sx>:Sub ; <y,sy>:Sub |] ==>
                                       <tApp(x,y),tApp(sx,sy)>:Sub"
        AbsI  "[| <x,sx>:Sub ; a:AP(aTerm) ; a#t ; a#n |] ==>
                                      <tAbs([a]x),tAbs([a]sx)>:Sub"
```

SUBSTITUTION

FIGURE 28. Functions out of `term`

The datatypes package of Isabelle98-1/ZF does not include facilities for declaring functions out of abstract datatypes directly, they must be declared as inductive relations and proved functional by hand.

Remark 19.2.1 (Terrible Struggle)**.** Each of the sets in Fig.28_{150} was a *terrible* struggle. For example `Size`, the size function. Before I could even begin to prove results about it I had to go back to Isabelle/ZFQA and extend the adaptation of Isabelle/ZF to include the theory of ordinal arithmetic (and everything leading up to it). ◇

I have proved that the relations in Fig.28_{150} are functions, proved their equivariance results, and established some of their simpler properties. For example:

```
a:AP(aTerm) ==> <tVar(a),fx>:FV ==> fx=a
```

and so on. My personal favourite is this:

```
Size_Omega_is_9  "a:AP(aTerm) ==>
<tApp(tAbs([a] (tApp(tVar(a),tVar(a)))),
      tAbs([a] (tApp(tVar(a),tVar(a)))) ), ?x>:Size";
```

During the proof, the Isabelle system instantiates the unknown `?x` step by step to the number nine. The result has little to do with FM set theory, but is a lovely little demonstration of the unity of logic and computation. It also shows it is possible to implement a deterministic programming language in Isabelle along with an executable big-step reduction relation.

Back to FM. Note how the definition of substitution in Fig.28_{150} is the same as the iterative equations defining substitution on $\mathbf{L}$ in $(49)_{69}$ except that I have expanded newness explicitly in terms of `#` (the theoretical justification for this being T9.4.6).

By expanding $И a$ we forfeit the useful ability to assume a apart from extra variables in the context (C9.4.11) as well as the ability to assume different new variables are equal (see `New_conj_tac` in R16.7.1, a practical instance of which appears in a later paper proof in R23.1.13).

Why did I do this? The datatypes package takes its rules in the meta-language:

(80) `[| Subgoal`$_1$ `; ... ; Subgoal`$_n$ `|] ==> Conclusion`

instead of Isabelle/FOL;

(81) `(Subgoal`$_1$ `& ... & Subgoal`$_n$`) --> Conclusion`

so on the face of it, to accommodate $(49)_{69}$ we would have to program $И$ into the meta-language of Isabelle itself—or change the datatypes package to accept rules in FOL+New.

The first alternative is not particularly appealing. The second is in principle just a small programming job: we write a small filter to detect rules of the form of $(81)_{151}$, possibly with $\forall\!\!\forall$ quantifiers up front, and apply `NewI_tac` (§16.7) and various other intro rules to reduce it to the form of $(80)_{151}$. But this is all rather pointless because it automates the wrong thing—getting rid of $\forall\!\!\forall a$ the moment it is handed to the datatypes package. We want to *preserve* $\forall\!\!\forall a$, leave it in the inductive principle of the datatype, and unpack it at the last moment during some inductive proof so that we may assume a apart from any extra variables in the context of that particular proof.

So forget that idea, we do not need it:

Remark 19.2.2 (Exceedingly nice datatypes)**.** FM has *exceedingly nice* properties which mean that *even though* the datatypes package only understands standard ZF, so to speak, it can still be made to handle FM datatypes with a minimum of fuss. C9.4.5 tells us that $\forall\!\!\forall$ distributes over conjunction and implication. So we can ask the programmer (or program into the datatypes package the ability) to rephrase a rule of the form

$$\forall\!\!\forall a.\ \frac{\phi(a,\vec{x})}{\psi(a,\vec{x})}$$

to the *logically equivalent* form

$$\frac{\forall\!\!\forall a.\ \phi(a,\vec{x})}{\forall\!\!\forall a.\ \phi(a,\vec{x})}. \tag{82}$$

By C9.4.5 we can also distribute $\forall\!\!\forall$ down through conjunctions and disjunctions in ϕ and ψ if we wish. We the designers of the system can easily put `New_conj_tac`, `NewI_tac` and `NewD_tac` (§16.7) into the datatypes package code[65] which will automatically reconstitute the original rule $(82)_{152}$ *in the course of a proof* placing `a#x` in the assumptions for every `x` in the context of that proof. ◇

Remark 19.2.3 (No **fresh**)**.** While we are on the subject, I mention that Isabelle/FM has no theory of **fresh** (T9.6.6). I should probably write one so we can use rules like $(48)_{69}$ and functions like C9.6.9. ◇

I conclude this subsection with one more incidental comment, a forward reference to Chapter IV.

Remark 19.2.4. Sometimes when ZF forces us to choose 'fresh variables x', FM allows us not to. For example in $(75)_{97}$ we do not need to declare names for new atoms because they remain hidden in the abstraction t_{**}. In ZF dialect

[65] I know the code quite well (R19.1.1).

we would *have* to write something like $(77)_{98}$.[66] I merely wish to point out that traditionally we always unpack abstractions and then re-bind the new atoms as appropriate. FM not only gives us И so we can do this much more cleanly than before, it also gives us the option of *not* doing so gratuitously. I next take up this issue in R25.5. ◇

19.3. Conclusion. Developing the theory of Λ_α was far harder than I expected. I should not have been surprised. Being the first application ever of Isabelle/FM, it naturally threw up many of its weaknesses and forced me to rewrite the code many times. Because of the rather concrete nature of Isabelle proof, to correct even a small strategic mistake (let alone a large one) can often mean trawling through all the code making hundreds of changes.

One of the major changes was a systematic organisation and development of the standard libraries of results discussed in §17. These libraries have names like `Perm_Left_SIs` (this collection of results helps pull permutations to the left of `=`, `:` and `#` in conclusions by intro-resolution) and thus allow us to easily implement ideas like "Pull all the permutations to the left of the set-inclusion". These fairly simple measures had a dramatic simplifying effect and reduced code-length sometimes by a factor of ten, just by helping me effectively apply the automated tools.

Remark 19.3.1 (Why I stopped where I did)**.** I began more substantial case studies than the λ-calculus. For example I declared the syntax and operational behaviour of the π-calculus, and also the inductive types of syntax and operational behaviour of an FreshML-like language (see Chapter IV). I did some work, and learnt for example that coinductively defined sets seem to present no new difficulties over inductively defined sets. The real problem was the equivariance results (R16.6.3), they just get bigger and badder. *Then* I realised how to get rid of them forever, see §20 below. The necessary work would have taken more time than I had. So I stopped. ◇

20. Future work: Releasing Isabelle/FM as a tool

In the last section I said the next step was to make a real tool of Isabelle, something that real people can use to do real syntax. I also said that the real problem with Isabelle/FM as it now exists is the equivariance results. Their proofs must be simplified:

> "Yes, Isabelle/FM lets you manipulate syntax with binding without worrying about silly technical ZF lemmas to do with choosing new

[66] In strict ZF we would not even have a И-quantifier.

> variable names. But you *do* have to spend ages on technical FM lemmas to do with permuting variable names."

No, we can't have that. An obvious solution is to develop a uniform automated proof-method for equivariance. Unfortunately this is impossible as things stand, recall R16.6.6. The best we can do is an automated proof for the kinds of datatypes users tend to define. In the course of my case studies I have already gone some of the way to doing this but I have fundamental objections to basing a system on the idea:

1. It would take thought and we could get it wrong.
2. I have executed such automated proofs for moderately nontrivial datatypes (datatypes of terms and typings of an FreshML-type language, for example) and they were slow. The proofs would presumably get slower as n^2 or similar in the complexity of the type. Still, computers get faster every day.
3. What happens to the user with a *new* kind of datatype which we did not consider? I was in this position wishing to declare datatypes of syntax involving abstraction `Abst(U,X)` (D9.6.1 and §16.10), which of course the (ZF) datatypes package did not know about. I hacked code for a week (thankfully just to get a trace, I did not have to change any algorithms, see R19.1.1).
4. Not all equivariance results concern datatypes. The user is free to axiomatise any set they wish, proving equivariance could be very tricky.

In Chapter II we used T8.1.10 and T9.1.6 to give us equivariance 'for free' by the properties of the logic of FM itself. We cannot reason about the meta-language of Isabelle/FM in Isabelle/FM, but we can axiomatise the properties we need.

I suggest the following. We extend Isabelle/FM to Isabelle/FM^{++}, at the beginning of which are (something like) the declarations of Fig.29$_{155}$. From these axioms we can for example prove

```
(a b).x = (b a).x
(a b).x = y <-> x = (a b).y
((a b).f)(x) = (a b).f((a b).x)
```

In addition we add an ML function `Equivar:string -> thm` that takes a string, parses it to an Isabelle term `t`, and checks that `t` has no free variables or scheme variables (see 'scheme' in the index of [**59**]), returning the theorem `(?a ?b).t=t` if so and `False` if not.[67]

[67] We could simulate the effects of `Equivar` by declaring a rule `(a b).Const=Const` for every `Const` in the theory and giving the result names in some systematic way. The ML program would then access these results through this systematic naming system to *prove* the required result. This makes for a lot of rule declarations and we would have to make new ones for each constant

```
classes
  atmc < term
consts
  PERM :: ['a::atmc,'a,'b] => 'b  ("(3'(_ _') ./ _)" 150)
rules
  PERM_idem   "(a b).((a b).x) = x"
  PERM_aba_b  "(a b).a = b"
  PERM_abb_a  "(a b).b = a"
  PERM_abx_x  "[| ~(x=a) ; ~(x=b) |] ==> (a b).x = x"
  PERM_ext    "(ALL x::'a::atmc. (a b).x = (c d).x) ==>
                                          (a b).z = (c d).z"
  Equivar_app "(a b).(f(x))=((a b).f)((a b).x)"
```

We also declare `Equivar:string -> thm` such that

$$\texttt{Equivar(t)} = \begin{cases} \texttt{(?a ?b).t=t} & \texttt{t} \text{ has no free variables or meta-variables} \\ \texttt{False} & \text{Otherwise} \end{cases}$$

`'a::atmc` is a polymorphic type of arity `atmc`. `?a` is a so-called 'scheme variable', which is a free variable which may be instantiated in unification (see [**59**]).

FIGURE 29. Declarations of Isabelle/FM^{++}

We would introduce object-level types of atoms in `AtmPart::i` just as in FM (§9) along with casting functions to biject them with `'a::atmc` as in §18.1. We would introduce object-level permutation `Perm::[i,i,i]=>i` just as in ZFA (§8). Now the casting functions mean `Perm` and `PERM` coincide on atoms, and by equivariance for `:` the action of `PERM` distributes over $\in$-structure just like `Perm`. By $\in$-induction `Perm` and `PERM` coincide.

Thus in §19.1, Fig.27_{149} instead of the code leading up to `qed "Perm_term"` we would just have

```
> bind_thm("Perm_term",Equivar("term"));
val Perm_term = "(a b).term = term" : thm
```

introduced. Furthermore, in some sense this is all completely equivalent to the proposal in the body of the text. But we would have kept our hands off the meta-system.

Pollack strongly recommends to me the approach of this footnote on the grounds that the meta-system should be inviolate. Paulson does not seem to care, quote "so long as it's consistent"; there is a difference in attitude here to the whole enterprise of mechanised mathematics. My thanks to them both for talking to me about this.

Remark 20.2. Note incidentally that the theory of λ-terms as constructed would break because I added a constant `aTerm:Atm` to the theory and this is *not* equivariant (see Fig.25_{144}). This is my error, but it is superficial. I should have made `aTerm:AtmPart` be a *type* of atoms instead. This can be done and it's no big deal. ◇

1. *The meta-language of Isabelle is a higher-order logic (HOL). Have we just addressed the question of "how to implement HOL/FM"?*

   No. There is no axiom in the meta-language corresponding to $(\text{Fresh})_{35}$.

2. *So what have I really done?* We really do just give the meta-level T8.1.10 which, both paper and Isabelle agree, most comfortably sits there and not lower down in the object level.

   The theory of $\mathbb{N}$ remains definitely object-level inside sets, and indeed Fig.29_{155} mentions no meta-level axiom of the form of $(\text{Fresh})_{35}$ for `'a::atmc` (it could, but that would bring us very little, discussion omitted).

   Of course we could use this extended meta-level to build any FM-style theory, not just Isabelle/FM (e.g. Isabelle/HOL-FM).

Remark 20.3 (The future)**.** The plans for the future are therefore these: We clean up the code and restrict permutation to types of atoms, see §18.2). We package equivariance into the meta-level as discussed above to make a general 'FM toolkit' not specific to set theory. We add a theory of **fresh** (T9.6.6, R19.2.3) to Isabelle/FM. We also extend the datatypes package as described in and around R19.2.2.

It would also be very convenient to mechanise §10.7 in Isabelle/FM^{++} (Fig.29_{155}) along with standard automated methods to prove that the various functions defined 'for free' on $\mathcal{V}_{\text{FM}}$ or **PreStx** correspond to the usual functions a developer would normally define on a datatype. Thus for example Isabelle/FM datatypes would automatically have a **FV** function defined iteratively by the system on request along with theorems proving it equal to **Supp**.

We can then carry out a few large case-studies. Isabelle/FM would then be a proven tool ready for general use. ◇

It sounds so easy.

Chapter IV

Programming Language: FreshML

21. Introduction

In Chapter II we constructed FM. Now we examine one aspect of what we might call its 'computational content'—how we can interpret programming languages and their operational semantics in it.

In Chapter IV we construct the datatypes of terms and values of a language FreshML *in* FM, build—still in FM—an operational semantics for it, and prove one significant correctness result T21.9 about that operational semantics. This proof is a prime target for automation, see §31.

Remark 21.4 (Totally Ordinary)**.** If what the reader is about to see looks indistinguishable from normal nameful ZF proofs (R4.14) he or she should be very happy—this is the way it is supposed to be—and not take this for granted; other approaches (§33) have never quite managed this '***totally ordinary***' look. ◇

Remark 21.5 (See §12!)**.** ***Chapter IV is carried out in*** FM ***and*** **uses** ***the extra power that gives us. The reader is reminded that in*** **§*12 I laid groundwork for this, discussed inductive reasoning in*** FM, and how the 'totally ordinary' look is achieved (for example we use R12.3.2). ◇

For the future we might like to do to an established programming language what FM does to ZF, i.e. extend it with intuitive facilities for handling binding[68] *while* leaving everything looking more-or-less the same. The fact that the "informal programming language" of §4 seems quite reasonable and *non-technical* (at least to me) suggests this is feasible. Note too that other approaches to the problem of datatypes with binding (§33) have *not* given us anything like FreshML or even the informal programming language, which suggests that this thesis is on to a good thing.

Pitts and myself have begun work on such a language, see [**66**], but I do not discuss that here. The aim of Chapter IV is more modest. We return to R4.10. Recall from there that although we do not and indeed cannot give the informal language of §4 a rigorous semantics[69] we *can* think of a program loosely as a function out of an FM-set. This is a powerful idea. I used it heavily to motivate my development of FM (Chapter II). Conversely, as FM has developed it has shown us what features our language *should* have. E.g. **fresh** (C9.6.7) would probably not have occured to us without FM.

---

[68] ... abstraction, concretion, **fresh**, etc.

[69] We have no theory of FM domains so cannot handle nontermination. We would not necessarily anticipate difficulties developing such a theory, but proper denotational semantics are complicated in themselves.

Notation 21.6. *We call the language we shall construct* FreshML*.*

Remark 21.7 (Lax denotational semantics)**.** FreshML has an (informal) denotational semantics as sets and functions in FM. For closed values (of lower-order-types), for which termination and typing contexts are no issue, we ***write the semantics of a value*** V ***as*** $[\![V]\!] \in \mathcal{V}_{\mathrm{FM}}$. For example, FreshML has an 'atom-binding' term-former $a.V$ and

$$[\![a.V]\!] = [\![a]\!].[\![V]\!]$$

($a.x$ introduced in D9.5.1). We may abuse notation where convenient and write $[\![t]\!]$ for t not necessarily a closed value. ◇

The point R4.10 makes is that if contextual equivalence does not appropriately match informal denotation—if there are V and V' such that $[\![V]\!] = [\![V']\!]$ but $V \not\equiv_{\mathbf{ctx}} V'$, vice versa, or both—then something is wrong. And indeed it might be that perfectly reasonable assumptions about type-constructors, type-destructors, operational semantics and the like unexpectedly interact to demolish even our informal ideas of what the programs denote.

Remark 21.8 (FreshML design aims)**.** In FreshML we take a collection of these reasonable assumptions and mathematically prove that we "get reasonable behaviour out", which is formalised in T21.9 below. The general idea of the result is that denotation and contextual equivalence are precisely equal in an appropriately restricted sense. The issue is *not* useability nor expressivity, except in so far as FreshML is not a trivial language. ◇

FreshML is based on a fragment of ML, enhanced with new features (inspired by FM) to support a type of λ-terms up to α-equivalence which we shall write Λ_α. We shall show that the following result holds (restated more rigorously in T29.25):

Theorem 21.9 (Sanity Clause)**.**[70] *Given*

- *an appropriate language* FreshML *whose types include a datatype* Λ_α *with informal denotation* $\boldsymbol{L}$ *(D10.3.4), untyped* λ*-terms up to* α*-equivalence,*
- *an appropriate operational semantics, and*
- *an appropriate inductive definition of contextual equivalence* $\equiv_{\boldsymbol{ctx}}$ *for that operational semantics,*

then for all closed values-in-context of type Λ_α

$$\emptyset \vdash V, V' : \Lambda_\alpha,$$

[70] Who says there's no such thing?

it is the case that contextual equivalence and α-equivalence[71] *coincide:*

$$\vdash V \equiv_{\boldsymbol{ctx}} V' : \Lambda_\alpha \iff V =_\alpha V'.$$

In other words:

> "Terms of the untyped λ-calculus up to α-equivalence are *fully* and *faithfully* represented by closed values of the appropriate FreshML datatype Λ_α modulo contextual equivalence."

Remark 21.10 (Provenance of Proof)**.** The guts of the proof-method used for T21.9 above are an instance of "Howe's Method". I used [**64**, Appendix A] by Pitts as my model, cf. R26.6.1. ◇

Remark 21.11 (... $\neq$ semantics of terms)**.** We conduct this work in FM; datatypes of terms and values, the operational semantics, contextual equivalence, are all inductively defined FM sets and we exploit in an integral way the full power of FM logic and the N quantifier.[72]

This is a shift of emphasis. In Chapter II our primitive notion of syntax was abstract syntax in some external universe (see Item 1 in R4.4) and terms were written $\mathtt{t}$. The FM-semantics we constructed were a derived notion intended to model this syntax in a controlled environment, so to speak. We called the FM semantics of syntax 'semantic terms' (N8.2.4), they were written $t = [\![\mathtt{t}]\!] \in \mathcal{V}_{\mathrm{FM}}$.

Now that we have FM, binding signatures (D10.1.3), T10.5.8, and so forth ***we throw away*** $\mathtt{t}$ ***and take as primitive the*** FM ***sets.*** ◇

Remark 21.12 (Potential for confusion)**.** I see a particular potential for confusion here because in R21.7 we introduce the concept of the denotation $[\![V]\!]$ of V a closed value of FreshML. Firstly there is a certain clash between this notation and the notation $[\![\mathtt{v}]\!]$, which is now primitive and written V. Worse, not only are syntax and denotation in the same universe, the denotation *itself* may be a set of syntax. The particular case which should most concern us is closed values of type Λ_α. They biject with **M** on *p.*57, the λ-terms *not* up to α-equivalence. Meanwhile

[71] To avoid confusing people with the truth I place this comment in a footnote: here "α-equivalence" does not refer to terms of FreshML, which are in the 'meta-level'. It refers to α-equivalence of 'object-level' closed values of type Λ_α *representing* (possibly open, possibly unevaluated) terms of the λ-calculus. See R22.1.6. This relation on closed values of type Λ_α will be written $\equiv^{se}$, rigorously constructed in D26.3.1, and used in the rigorous restatement of this result, T29.25. $\equiv^{se}$ enters into the proof via Fig.38_{196}.

[72] As in FML$_{tiny}$ (R12.1.2 Item 1) terms and values are defined in Fig.31_{164} by *mutual* induction. We did not consider mutually inductive definitions in §10. However as discussed there they can be encoded in ordinary inductive definitions so in theory the mathematics is powerful enough to include our syntax.

(83)	$\tau ::=$	`Atm`	Atoms
		$\mid$ `Bool`	Bool
		$\mid$ `Nat`	Nat
		$\mid \tau \to \tau$	Function types
		$\mid \tau \times \tau$	Product types
		$\mid (\tau)$`List`	List types
		$\mid$ [`Atm`]τ	Abstraction types
		$\mid \Lambda_\alpha$	Lambda terms up to α-equivalence

FIGURE 30. D22.1.1 - Types of FreshML

the type's values' informal denotation bijects with **L** (D10.3.4), the λ-terms *up to* α-equivalence. Cf. R22.1.6 and ft.71_{160}. ◇

Remark 21.13 (Many types of atom)**.** Our FM here has many types of atoms (R16.1.2). We need this to hypothesise two types of atoms **Var** and **AtmC** to implement syntactic sets of variable symbols and constant symbols respectively.

Now the language FreshML will have a type of atoms `Atm` (see Fig.30_{161}). The constant symbols in **AtmC** will all be of type `Atm` and we shall need an FM type of atoms $\mathbb{A}$ for the *denotation* of the FreshML type `Atm`.[73] In principle this is a third type of atoms. *However*, FM does not admit a bijective function-set between distinct types of atoms—such a bijection would have nonfinite support, proof omitted but similar to that of T11.4.1, cf. also §16.11. Thus $\mathbb{A}$ and **AtmC** must actually be identical and the function mapping a to $[\![a]\!]$ must be the identity. We shall ignore this and preserve distinct symbols for **AtmC** the set of syntax and $\mathbb{A}$ the denotation. ◇

22. Syntax

22.1. Definition.

Definition 22.1.1 (Types of FreshML)**.** *Types τ of* FreshML *are inductively defined in Fig.30_{161}.*

[73] Recall that syntax and denotation are living in the same universe, cf. R21.12.

The type system is minimal and unsurprising except for the constants `Atm` and Λ_α and the type-former $[\texttt{Atm}]-$.

Remark 22.1.2 (Informal Denotation)**.**

1. Λ_α is a ***type of λ-terms***. Its intended denotation is the FM-set $\mathbf{L}$ of $(42)_{67}$, namely untyped λ-terms up to α-equivalence. Looking ahead for a moment to R22.1.6, we shall see that terms of type Λ_α are actually isomorphic to untyped λ-terms *not* up to α-equivalence.
2. `Atm` is a ***type of atoms***. Its intended denotation is $\mathbb{A}$ (§8 also R21.13). The only closed values of type `Atm` will be a collection of constants $a, b, c \in$ **AtmC**.
3. We call $[\texttt{Atm}]\tau$ ***abstraction types***. The intended denotation corresponds to the abstraction-set former $[\mathbb{A}]X$ of D9.6.1.

$\diamond$

As stated in R21.7, a rigorous denotational semantics for FreshML demands domain theory and we ignore this.

We shall need some constructor symbol constants to build a set of terms.

Definition 22.1.3 (Term Constructors of FreshML)**.** *We declare a set of* ***term constructors****. Abusing notation we equate the constructor symbol with its sugared (e.g. possibly infix term-constructor functional) notation:*

$$\begin{aligned}(84)\qquad \boldsymbol{TermCon} = \{&\texttt{True}, \texttt{False}, 0, \texttt{Succ}(-), (-,-),\\ &\texttt{Nil}_\tau, -::-, \texttt{Var}(-), \texttt{App}(-), \texttt{Lam}(-)\}.\end{aligned}$$

As promised in R21.13 we declare two types of atoms in FM*, one for variable symbols $x, y, z \in$ **Var** and and the other for constant symbols $a, b, c \in$ **AtmC**. The type system will in due course (§24) assign constants a, b, c type* `Atm`*. We introduce a notation for a useful syntactic class of variables-or-constants $xa, xb, \ldots \in$ **Var**$\cup$ **AtmC**:*

$$\begin{aligned}x, y, z &\in \boldsymbol{Var}\\ a, b, c &\in \boldsymbol{AtmC}\\ xa, xb, xc, ya, yb, \ldots &\in \boldsymbol{Var} \cup \boldsymbol{AtmC}\end{aligned} \tag{85}$$

Finally we define terms t and values V as datatypes with binding in FM:[74]

[74] In §10 we did not consider binding signatures for *mutually* inductive definitions such as that of D22.1.4. We can encode them using ordinary inductive definitions (Item 1 of R12.1.2), so §10 is powerful enough to include the syntax we are defining now. We gloss over the issue.

Definition 22.1.4 (Terms of FreshML)**.** *Terms* $t \in$ ***Terms*** *and values* $V \in$ ***Val*** *of* FreshML *are constructed by mutual induction in Fig.31*$_{164}$ *as datatypes with binding in* FM*.*

In those figures, binding occurrences of variable symbols are written underlined, as in $(\texttt{fix}\,\underline{f}(\underline{x}:\tau)\,\texttt{in}\;t)$.[75] The ***free variables*** function on terms (and values) is written $\mathbf{FV}(t)$, we can obtain it 'for free' by D10.7.2. Atom-abstraction $xc.t$ is *not* binding on xc—we shall call it ***synthetically binding*** instead (see §23 and R23.1.1). We write the set of ***closed terms*** and ***closed values***

$$\mathbf{CTerms} \stackrel{\text{def}}{=} \left\{ t \in \mathbf{Terms} \;\middle|\; \exists\tau.\, \emptyset \vdash t:\tau \right\} \quad \text{and}$$

$$\mathbf{CVal} \stackrel{\text{def}}{=} \left\{ V \in \mathbf{Val} \;\middle|\; \exists\tau.\, \emptyset \vdash V:\tau \right\}$$

respectively.[76]

The following ground has already been well-covered, but I should not assume the reader is completely familiar with the rest of the document and I go over this again:

Remark 22.1.5 (Permutation for free)**.** FM sets have a transposition action $(a\ b)\cdot x$ (a, b assumed to be atoms of the same type, i.e. **Var**, **AtmC**, `Atm` or $\mathbb{A}$) introduced in D8.1.8 which literally goes through the $\in$-structure of x swapping a for b and b for a (R8.2.1). In particular our inductive types in FM inherit this action 'for free' (cf. §10.7), there is no need to inductively define it over the structure of individual types. So suppose $x = t$ is a term. We use this transposition action on t much as the reader may have used name-for-name substitution $[b/a]$, and indeed the two are equal if b is fresh and does not occur in t (cf. §11.1 for why we do *not* build FM and in particular **Terms** using $[b/a]$). ◇

Remark 22.1.6 (Abstraction *not* binding)**.** $xc.V$ does not bind xc in V; $a.\texttt{Var}(a)$ is not the same term as $b.\texttt{Var}(b)$.[77] So the terms of Λ_α correspond

[75] Binding in FM datatypes has a precise meaning, that is what this thesis is all about. See R12.1.3. Note that this underlining notation is ambiguous because it does not specify the scope of the binding (cf. ft.31$_{91}$). The scoping is as I hope the reader would expect, the only slightly complicated binding is that $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$ binds x in t_1 but not t_2. I.e. `Let` is a function on $[\mathbf{Var}]\mathbf{Terms} \times \mathbf{Terms}$ (cf. §12).

[76] Here I do not get closed terms and values 'for free' from D10.7.8 as I did in D12.6.1 for FML$_{tiny}$. If we automated this proof (§31) we might prefer to do so because it sidesteps the typing relation. However, if I did that here the reader might ask to see a proof that the version using empty typing contexts above is identical to the free version, and that would be a tangent.

[77] This means quite precisely "$a.\texttt{Var}(a)$ and $b.\texttt{Var}(b)$ not identical FM sets". See R21.11. The informal denotations are equal: $[\![a.\texttt{Var}(a)]\!] = [\![b.\texttt{Var}(b)]\!]$.

$t ::=$	V	Values
	$\texttt{let}\ \underline{x} = t\ \texttt{in}\ t$	Let
	$V\ V$	Application
	$V @ V$	Atom concretion
	$\texttt{fresh}\ \underline{x}\ \textbf{in}\ t$	Atom binding
	$\texttt{if}\ V = V\ \texttt{then}\ t\ \texttt{else}\ t$	Atom destructor
	$\texttt{if}\ V\ \texttt{then}\ t\ \texttt{else}\ t$	`Bool` Case
	$\texttt{Fst}(V)$	First projection
	$\texttt{Snd}(V)$	Snd projection
	$\texttt{Case}\ V\ \texttt{of}\ \{\texttt{Var}(\underline{x}) \Rightarrow t, \texttt{App}(\underline{x}) \Rightarrow t, \texttt{Lam}(\underline{x}) \Rightarrow t\}$	Λ_α Case
	$\texttt{Case}\ V\ \texttt{of}\ \{\texttt{Nil}_\tau \Rightarrow t, \underline{x}::\underline{x} \Rightarrow t\}$	List Case
	$\texttt{Case}\ V\ \texttt{of}\ \{0 \Rightarrow t, \texttt{Succ}(\underline{x}) \Rightarrow t\}$	`Nat` Case

$V ::=$	$\texttt{fix}\,\underline{f}(\underline{x} : \tau)\ \texttt{in}\ t$	Function fixedpoints
	xc	Variables and Constants
	$\texttt{Con}(V_1, \ldots, V_n)$	Constructor arity n
	$V.V$	New Atom Binding

Bound variable symbols are underlined, cf. R12.1.3.

FIGURE 31. D22.1.4 - Terms and values of FreshML

to elements of the naïve datatype (D10.5.2) of λ-terms not up to α-equivalence. Recall that the intended denotation of Λ_α is untyped λ-terms *up to* α-equivalence.

So, the Sanity Clause T21.9 states that contextual equivalence on closed values coincides with equality in our denotation. The language cannot access the name of an atom synthetically bound by an abstraction term-former $a.(-)$, so that they really are in this (operational) sense 'bound'. ◇

We drive the point home. $x \in \mathbf{FV}(x.V)$ even for $V \in \mathbf{CVal}$ a closed value. $x.V$ is *always* an open term.

Remark 22.1.7 (Reduced Syntax)**.** We use "***reduced syntax***"; term-formers take values instead of general terms wherever possible. So for example (V_1, V_2) is a well-formed term (and value) and (t_1, t_2) is *not*. We can use $\texttt{let}\ x = t\ \texttt{in}\ t'$ to recover expressivity. E.g. (t_1, t_2) can be encoded as

$$\texttt{let}\ x_1 = t_1\ \texttt{in}\ \texttt{let}\ x_2 = t_2\ \texttt{in}\ (x_1, x_2).$$

I shall usually sugar nested lets to (expressions along the lines of) $\texttt{let}\ x_1 = t_1, x_2 = t_2\ \texttt{in}\ (x_1, x_2)$ ***without comment.*** See e.g. $(\texttt{Concat})_{166}$.

Reduced syntax is useless for programming (we drown in `Lets`) though FreshML could be an intermediate language for some more user-friendly syntax. It simplifies the theory; sequential evaluation behaviour is concentrated in one term-former ($\texttt{let}\ x = -\ \texttt{in}\ -$). So for example in evaluation relation in $\text{Fig.}36_{188}$ equation $(149)_{188}$ is not

$$\frac{t\ \Downarrow\ (V_1, V_2)}{\texttt{Fst}(t)\ \Downarrow\ V_1}, \qquad \text{but the rather simpler} \qquad \texttt{Fst}(V_1, V_2)\ \Downarrow\ V_1.$$

C.f. R16.6.7 where we see the same principle of 'avoiding conditions where possible' appear in the context of implementing proofs in a theorem-proving environment. The difference is magnified every time we do induction over the evaluation relation, notably in the rather complicated and very important T27.14. ◇

Theorem 22.1.8 (Substitution Property)**.** *If t is a term, $x \in$ **Var** a variable symbol and V a value then $t[V/x]$ (where $[-/-]$ denotes the usual capture-avoiding substitution) is a well-formed term. Furthermore, if U is a value then $U[V/x]$ is also a value.*

Proof. By mutual induction on the reduced syntax of t and U, or alternatively 'for free' (D10.7.13). □

22.2. Discussion. Let us consider some likely-looking programs we can write in FreshML, some which we would like to be typeable and some not.

1• *We can write the functions we used to biject $[\mathbb{A}](X \times Y)$ with $[\mathbb{A}]X \times [\mathbb{A}]Y$ in C9.6.9:*

(Bij1) $\qquad \texttt{fix}\, f(x : [\texttt{Atm}](\tau \times \tau'))\ \texttt{in}\ \texttt{fresh}\, u\, \textbf{in}\ (u.\texttt{Fst}(x@u), u.\texttt{Snd}(x@u))$

(Bij2) $\qquad \texttt{fix}\, f(x : ([\texttt{Atm}]\tau \times [\texttt{Atm}]\tau'))\ \texttt{in}\ \texttt{fresh}\, u\, \textbf{in}\ u.(\texttt{Fst}(x@u), \texttt{Snd}(x@u))$

2• *We can write substitution* $Subst(t,y)(-)$ *on* Λ_α*:*

$(\texttt{Subst}(t,y))$ $\texttt{fix}\, f(x:\Lambda_\alpha)\,\texttt{in Case}\, x\,\texttt{of}$

$$\left\{\begin{array}{l}\texttt{Var}(u)\Rightarrow \texttt{if}\ u=y\ \texttt{then}\ t\ \texttt{else Var}(u),\\ \texttt{App}(s)\Rightarrow \texttt{App}(f(\texttt{Fst}(s)),f(\texttt{Snd}(s))),\\ \texttt{Lam}(s_*)\Rightarrow \texttt{fresh}\, u\,\textbf{in}\,\texttt{Lam}(u.f(s_*@u))\end{array}\right\}$$

3• *We can write "list concat" and using it a "list of free variables" function:*

$(\texttt{Concat})$ $\texttt{fix}\, f(x:(\texttt{Atm})\texttt{List}\times(\texttt{Atm})\texttt{List})\,\texttt{in let}\ x_1=\texttt{Fst}(x),x_2=\texttt{Snd}(x)\ \texttt{in}$

$$\texttt{Case}\, x_1\,\texttt{of}\left\{\begin{array}{l}\texttt{Nil}_{(\texttt{Atm})\texttt{List}}\Rightarrow x_2,\\ hd{::}tl\Rightarrow hd{::}f(tl,x_2)\end{array}\right\}$$

$(\texttt{FVlist})$ $\texttt{fix}\, f(x:\Lambda_\alpha)\,\texttt{in Case}\, x\,\texttt{of}$

$$\left\{\begin{array}{l}\texttt{Var}(u)\Rightarrow u{::}\texttt{Nil}_{(\texttt{Atm})\texttt{List}},\\ \texttt{App}(s)\Rightarrow \texttt{Concat}(f(\texttt{Fst}(s)),f(\texttt{Snd}(s))),\\ \texttt{Lam}(s_*)\Rightarrow \texttt{fresh}\, u\,\textbf{in}\,\texttt{Lam}(u.f(s_*@u))\end{array}\right\}$$

4• *We can write a "body of an abstraction" function:*

(FALSE BV) $\texttt{fix}\, f(x_*:[\texttt{Atm}]\tau)\,\texttt{in}\,\texttt{fresh}\, u\,\textbf{in}\, x_*@u$

which should not type! u is chosen fresh so we need $u\#x_*@u$ for the function to have an informal denotation (T9.6.6), and this is not generally the case. The moral is

> "We may invent abitrary fresh variable names but they must not escape the scope of their declaration."

Similarly we would expect $(\texttt{fresh}\, u\,\textbf{in}\, u)$ *not to type.*

5• *We can write a plausible (but we hope untypeable) "list of bound variables" function:*

(FALSE BVlist) $\texttt{fix}\, f(x:\Lambda_\alpha)\,\texttt{in Case}\, x\,\texttt{of}$

$$\left\{\begin{array}{l}\texttt{Var}(u)\Rightarrow \texttt{Nil}_{(\texttt{Atm})\texttt{List}}\\ \texttt{App}(s)\Rightarrow \texttt{Concat}(f(\texttt{Fst}(s)),f(\texttt{Snd}(s)))\\ \texttt{Lam}(s_*)\Rightarrow \texttt{fresh}\, u\,\textbf{in}\, u{::}f(s@u)\end{array}\right\}$$

We have very simple pattern-matching in FreshML which does not include matching of abstraction-types, but the reader can see from the examples so far that

we have been implementing it anyway by systematically using `fresh` u **in** $(-)$ to invent a fresh variable and concretion $(-)@a$ to destroy the abstraction.

It does no harm to consider the same program written in the informal programming language of §4 using pattern-matching at abstraction types:

```
fun BVlist Var(u)       = Nil(Atm-list)
|   BVlist App(s1,s2)   = Concat(BVlist(s1),BVlist(s2))
|   BVlist Lam([u]s)    = u::BFlist(s);
```

We would expect the typing system to reject this.

In §23.1 we inductively construct an apartness judgement in which is part of the the typing system (§24) and rejects the undesirables from the list above.

23. Apartness judgements

23.1. Definition.

Remark 23.1.1 (Synthetic binding)**.** Recall from R9.2.10 that # is an FM version of "is not in the free variables of", and $a\#x$ can be read as "a is synthetically bound/is not synthetically free in x". Using the informal denotation we extend this terminology to FreshML and say that a ***is synthetically bound in the term*** $a.V$. ◇

The denotation of $V@a$ is $[\![V]\!]@[\![a]\!]$. In fact $[\![a]\!] = a$ by R21.13 and ***I shall in future write*** a ***for both.*** By D9.5.14, $x@a$ is defined only for $a\#x$. This should be reflected in the language or there will be terms (such as $b.a@a$ for $b \neq a$) that can have no denotation. Similarly, the denotation of `fresh` a **in** t is **fresh** $a.[\![t]\!]$, and this is only well-defined when $\mathcal{N}a.\ a\#[\![t]\!]$. Before a typing system we must develop some syntactic approximation to FM-apartness # (N9.2.4). It can only *be* an approximation because apartness at higher types in undecidable.

Remark 23.1.2 (Overview of the construction)**.** In the rest of this section we shall inductively define a syntactic approximation to apartness judgements and synthetic binding in FM as described in R23.1.1 and R9.2.10. We shall also call these 'apartness judgements', we write them

$$\Gamma_{\#} \vdash t\#xc.$$

Their intended interpretation will be "In the context $\Gamma_{\#}$ the possibly open term t does not contain xc synthetically free inside it". The apartness context $\Gamma_{\#}$ encodes information about the atoms synthetically bound in possible values for variable symbols, which may occur free in the term t. xc may be a variable symbol x (which we should imagine to be of type `Atm`) or an atom constant symbol a. ◇

There are some subtleties to this. Consider the term

$$t = (\texttt{let}\ x = a\ \texttt{in}\ a.x).$$

It should be the case that $[\![t]\!]\#[\![a]\!]$, but we can see that a occurs outside the syntactic scope of $a.-$ in t. Hence, we cannot say "xc is synthetically bound if it occurs under the scope of a synthetic binder", corresponding to "x is syntactically bound if it occurs under the scope of a syntactic binder". Our decision to use reduced syntax (R22.1.7) forces $\texttt{let}\ x = -\ \texttt{in}\ -$ to be a primitive term-former, and this leaves us no choice but to have a cut-like rule $(102)_{170}$ in the definition of the apartness judgements (Fig.32_{169} and Fig.33_{170}).

Definition 23.1.3 (Apartness Context)**.** *An **apartness context** is a finite subset of* $(\boldsymbol{Var} \cup \boldsymbol{AtmC})^2$ *written* $\Gamma_\#$ *such that for all* $xc \in \boldsymbol{Var} \cup \boldsymbol{AtmC}$, $(xc, xc) \notin \Gamma_\#$*, see R23.1.5 below. We write the set of apartness contexts* $\boldsymbol{Ctx}_\#$*.*

Remark 23.1.4. The intended interpretation of $\Gamma_\#$, abuse notation and write it $[\![\Gamma_\#]\!]$, is

$$\bigwedge_{(xc,yc)\in\Gamma_\#} [\![xc]\!]\#[\![yc]\!].$$

◇

Remark 23.1.5. Note that the definition of a typing context prevents $(xc, xc) \in \Gamma_\#$. Since the informal meaning of $(xc, yc) \in \Gamma_\#$ is $[\![xc]\!]\#[\![yc]\!]$ allowing this would allow $\bot$ into our context. This is undesirable, cf. Item 2 of §23.2. ◇

Definition 23.1.6 (Apartness Judgement)**.** *A **apartness judgement** is a triple*

$$(\Gamma_\#, t, xc)$$

where $\Gamma_\# \in \boldsymbol{Ctx}_\#$ *is an apartness context (D23.1.3),* $t \in \boldsymbol{Terms}$ *is a possibly open term and* $xc \in \boldsymbol{Var} \cup \boldsymbol{AtmC}$ *is a variable symbol or atom constant symbol. We write an apartness judgement*

$$\Gamma_\# \vdash t\#xc,$$

*and read it "t **is apart from** xc in the context $\Gamma_\#$".*

The valid apartness judgements are inductively defined in Fig.32_{169} and Fig.33_{170} using the notation in N23.1.8. The intended meaning of a apartness judgement is

$$\text{"}\forall \textit{values for variables. } [\![\Gamma_\#]\!] \implies [\![t]\!]\#[\![xc]\!]''$$

We shall write $\emptyset \vdash t\#xc$ *as* $t\#xc$ *where this is convenient.*

(86) $\Gamma_{\#} \vdash c\#a \qquad c \neq a$ **Atom Constants**

(87) $\Gamma_{\#} \vdash xc\#yc \qquad (xc, yc) \in \Gamma_{\#}$ **Variables (or Constants)**

(88) $\Gamma_{\#} \vdash \texttt{True}\#xc$ `Bool`

(89) $\Gamma_{\#} \vdash \texttt{False}\#xc$

(90) $$\frac{\Gamma_{\#} \vdash t_1\#xc \quad \Gamma_{\#} \vdash t_2\#xc}{\Gamma_{\#} \vdash \texttt{if}\ V\ \texttt{then}\ t_1\ \texttt{else}\ t_2\#xc}$$

(91) $\Gamma_{\#} \vdash 0\#xc$ `Nat`

(92) $\Gamma_{\#} \vdash \texttt{Succ}(V)\#xc$

(93) $$\frac{\Gamma_{\#} \vdash t_0\#xc \quad \Gamma_{\#}, x\#\overline{zc} \vdash t_f\#xc}{\Gamma_{\#} \vdash \texttt{Case}\ V\ \texttt{of}\ \{0 \Rightarrow t_0, \texttt{Succ}(x) \Rightarrow t_f\}\#xc}$$

(94) $\Gamma_{\#} \vdash \texttt{fix}\, f(x : \tau')\, \texttt{in}\ t\#xc$ **Function Types**

$xc \notin \mathbf{FV}(t) \wedge xc \notin \{c \in \mathbf{AtmC} \mid c \text{ in text of } t\}$

(95) $$\frac{\Gamma_{\#} \vdash V_2\#xc \quad \Gamma_{\#} \vdash V_1\#xc}{\Gamma_{\#} \vdash V_1\ V_2\#xc}$$

(96) $$\frac{\Gamma_{\#} \vdash V\#yc}{\Gamma_{\#} \vdash xc.V\#yc} \quad yc \neq xc$$ **Abstraction Types**

(97) $\Gamma_{\#} \vdash xc.V\#xc$

(98) $$\frac{\Gamma_{\#} \vdash V\#xc \quad \Gamma_{\#} \vdash yc\#xc}{\Gamma_{\#} \vdash V@yc\#xc}$$

(99) $$\frac{\Gamma_{\#}\#x \vdash t\#xc}{\Gamma_{\#} \vdash \texttt{fresh}\, x\, \textbf{in}\ t\#xc}$$ **Atom-Binding**

(100) $$\frac{\Gamma_{\#} \vdash t_1\#zc \quad \Gamma_{\#} \cup xc\#yc \vdash t_2\#zc}{\Gamma_{\#} \vdash \texttt{if}\ xc = yc\, \texttt{then}\ t_1\ \texttt{else}\ t_2\#zc} \quad xc \neq yc$$ **Atom-Destructor**

(101) $$\frac{\Gamma_{\#} \vdash t_1\#zc}{\Gamma_{\#} \vdash \texttt{if}\ xc = xc\, \texttt{then}\ t_1\ \texttt{else}\ t_2\#zc}$$

FIGURE 32. D23.1.6 - Apartness Judgements 1

(102) $$\frac{\Gamma_\# \vdash t_1 \# \overline{xc} \quad \Gamma_\#, x \# \overline{xc} \vdash t_2 \# yc}{\Gamma_\# \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \# yc}$$ **Sequential Computation**

(103) $$\frac{\Gamma_\# \vdash V \# xc \quad \Gamma_\# \vdash V' \# xc}{\Gamma_\# \vdash (V, V') \# xc}$$ **Products**

(104) $$\frac{\Gamma_\# \vdash V \# xc}{\Gamma_\# \vdash \texttt{Fst}(V) \# xc}$$

(105) $$\frac{\Gamma_\# \vdash V \# xc}{\Gamma_\# \vdash \texttt{Snd}(V) \# xc}$$

(106) $$\Gamma_\# \vdash \texttt{Nil}_\tau \# xc$$ **Lists**

(107) $$\frac{\Gamma_\# \vdash V_1 \# xc \quad \Gamma_\# \vdash V_2 \# xc}{\Gamma_\# \vdash V_1 :: V_2 \# xc}$$

(108) $$\frac{\begin{array}{c}\Gamma_\# \vdash V \# \overline{xa} \quad \Gamma_\# \vdash t_{nil} \# xc \\ \Gamma_\#, x \# \overline{xa}, y \# \overline{xa} \vdash t_{cons} \# xc\end{array}}{\Gamma_\# \vdash \texttt{Case}\ V\ \texttt{of}\ \{\texttt{Nil}_\tau \Rightarrow t_{nil}, x :: y \Rightarrow t_{cons}\} \# xc}$$

(109) $$\frac{\Gamma_\# \vdash V \# xc}{\Gamma_\# \vdash \texttt{Var}(V) \# xc}$$ **λ-terms**

(110) $$\frac{\Gamma_\# \vdash V \# xc}{\Gamma_\# \vdash \texttt{App}(V) \# xc}$$

(111) $$\frac{\Gamma_\# \vdash V \# xc}{\Gamma_\# \vdash \texttt{Lam}(V) \# xc}$$

(112) $$\frac{\begin{array}{c}\Gamma_\# \vdash V \# \overline{xa} \quad \Gamma_\#, x \# \overline{xa} \vdash t_V \# xc \\ \Gamma_\#, x \# \overline{xa} \vdash t_A \# xc \\ \Gamma_\#, x \# \overline{xa} \vdash t_L \# xc\end{array}}{\Gamma_\# \vdash \texttt{Case}\ V\ \texttt{of}\ \{\texttt{Var}(x) \Rightarrow t_V, \texttt{App}(x) \Rightarrow t_A, \texttt{Lam}(x) \Rightarrow t_L\} \# xc}$$

FIGURE 33. D23.1.6 - Apartness Judgements 2

Remark 23.1.7 (Notational Clash)**.** There is now a ***clash of notation*** between $\Gamma_\# \vdash t \# xc$, read as above as "t is apart from xc in the context $\Gamma_\#$", and $x \# y$ (N9.2.4) read as "x is apart from y". I rely on the context (... of the discourse, not $\Gamma_\#$) to disambiguate. ◇

Notation 23.1.8 (Shorthand)**.** *We use the following shorthand notations in Fig.32*$_{169}$ *and Fig.33*$_{170}$ *and the development that follows:*

1. *We write* $\overline{xc}$ *for an arbitrary finite set of* xc*.*
2. *We write*

$$\Gamma_{\#} \vdash t\#\overline{xc} \quad \textit{for} \quad \bigwedge_{xc \in \overline{xc}} (\Gamma_{\#} \vdash t\#xc).$$

3. *We write*

$$\overline{xa}\#xc \quad \textit{for} \quad \left\{(xa, xc) \;\middle|\; xa \in \overline{xa}\right\},$$

and

$$xc\#\overline{xa} \quad \textit{for} \quad \left\{(xc, xa) \;\middle|\; xa \in \overline{xa}\right\}.$$

4. *We write*

$$\Gamma_{\#} \cup xa\#\overline{xc} \quad \textit{for} \quad \Gamma_{\#} \cup \left\{(xa, xc) \;\middle|\; xc \in \overline{xc}\right\}$$
$$\textit{and} \quad \Gamma_{\#} \cup xa\#xc \quad \textit{for} \quad \Gamma_{\#} \cup \{(xa, xc)\}.$$

5. *We write*

$$\Gamma_{\#}, xa\#xc \quad \textit{for} \quad \Gamma_{\#} \cup xa\#xc \qquad \textit{and} \qquad \Gamma_{\#}, xa\#\overline{xc} \quad \textit{for} \quad \Gamma\# \cup xa\#\overline{xc},$$

   but in both cases insist additionally that there is no za *such that* $(xa, za) \in \Gamma_{\#}$*.*
6. *We write*

$$\overline{xc}[t/x] \quad \textit{for} \quad \left\{xc[t/x] \;\middle|\; xc \in \overline{xc}\right\},$$

   where $[t/x]$ *is a substitution of* x *for* t*.*
7. *t[t'/z] normally denotes substitution of* t' *for* z *in* t*. In the case of* (xa, xc) *in an apartness context* **only***,* $(xa, xc)[V/z]$ *shall mean* $(xa[V/z], xc[V/z])$ **only if** $V = zc \in \mathbf{Var} \cup \mathbf{AtmC}$ *and* **otherwise** (xa, xc) *(because it makes no sense to substitute a value not equal to an atom in an apartness context.*
8. *. . . similarly for the expression* $xc[V/z]$ *in an apartness judgement* $\Gamma_{\#} \vdash t\#xc[V/z]$*.*
9. *If* $xc\#\Gamma_{\#}$ *we write*

$$\Gamma_{\#}\#xc \quad \textit{for} \quad \Gamma_{\#} \cup \left\{(ya, xc) \;\middle|\; \exists yc.\ (ya, yc) \in \Gamma_{\#}\right\}.$$

   L23.1.9 below is rather more than enough to guarantee that $(xc, xc) \notin \Gamma_{\#}\#xc$ *and so this is well-formed.*

The following result generalises L12.4.2 although the proof is completely different.

Lemma 23.1.9. *If $R \subseteq X \times Y$ is a* finite *relation then $a\#R$ iff $a\#x, y$ for all $(x, y) \in R$.*

PROOF. A corollary of C13.2.1 and L9.3.5 for the injective pair-set function class $\lambda x, y.(x, y)$. □

Corollary 23.1.10. *As* FM *sets $zc\#\Gamma_\#$ iff $zc \neq xc, yc$ for all $xc, yc \in \Gamma_\#$.*

PROOF. Just L23.1.9 combined with L9.3.6 to turn $zc\#xc, yc$ into $zc \neq xc, yc$. □

L12.4.2 is a similar result for FML*tiny*. It often happens that we know $zc\#\Gamma_\#$ because we pick bound variables fresh in expressions such as $\texttt{fresh}\, x\, \textbf{in}\ t$ and $(\texttt{let}\ x = t_1\ \texttt{in}\ t_2)$, cf. $(72)_{94}$. E.g. in T23.1.12 below. We continue developing this in L24.1.5.

Now we prove apartness judgement weakening. We never use this result (although it is nice to know it is true). I use the proof to restate some of the points made in §12 about inductive reasoning in FM.

Remark 23.1.11 (I really mean it)**. Please note!** Here in T23.1.12 and often elsewhere I shall prove properties of an inductively defined X a subset of some D by induction "with inductive hypothesis $\phi(x)$" (where $x \in D$). Often, $\phi(x)$ will be of the form $x \in X \implies \phi'(x)$; e.g. $(113)_{172}$.

I really mean it. The property I prove of X is $\phi(x)$, not $\phi'(x)$.[78] However, in the proof itself I shall tend to elide complexities, e.g. compare $(113)_{172}$ and $(114)_{172}$. ◇

Theorem 23.1.12 (Weakening)**.**

$$(\Gamma_\# \vdash t\#xc \ \wedge\ \Gamma_\# \subseteq \Gamma'_\#) \implies \Gamma'_\# \vdash t\#xc.$$

PROOF. By induction on $\Gamma_\# \vdash t\#xc$ using induction hypothesis

$$\Gamma_\# \vdash t\#xc \implies \forall \Gamma_\# \subseteq \Gamma'_\#.\ \Gamma'_\# \vdash t\#xc. \tag{113}$$

We consider only the case of $(99)_{169}$ in any detail. Suppose we know

$$\Gamma_\# \vdash t\#xc, \quad \text{and} \quad \Gamma_\# \subseteq \Gamma'_\#. \tag{114}$$

Now suppose $t = \texttt{fresh}\, x\, \textbf{in}\ t'$.

Here is the point: Because we are using FM syntax, t is actually $\texttt{New}(t'_*)$ for $t'_* \in [\textbf{Var}]\textbf{Terms}$—see the discussion of these issues in §12, in particular R12.1.3

[78] Having things this way does no harm and sometimes the extra power is useful.

and R12.3.2. $(99)_{169}$ is shorthand for

$$\text{(Real (99))} \qquad \text{И}x.\ \forall t.\ \frac{\Gamma_{\#}\#x \vdash t\#xc}{\Gamma_{\#} \vdash \texttt{fresh}\, x\, \textbf{in}\ t\#xc}$$

and so x is chosen new and apart from everything free in the current context, including $x\#t'_*, \Gamma, \Gamma'$ (but not apart from t, which is not in the context, cf. R9.4.12). Item 5 of N23.1.8 states that for $\Gamma_{\#}\#x$ and $\Gamma'_{\#}\#x$ to be well-formed, we need $x\#\Gamma, \Gamma'$. We know this. So

$$\Gamma_{\#}\#x \vdash t\#xc.$$

We know $\Gamma'_{\#}\#x$ is well-formed too and from its definition (Item 5 of N23.1.8) we have $\Gamma_{\#}\#x \subseteq \Gamma'_{\#}\#x$ and this gives us the result by the inductive hypothesis.

In a normal ZF presentation without И some confusion might arise over whether x can occur in $\Gamma'_{\#}$, since the only explicit condition on it is that $\Gamma_{\#} \subseteq \Gamma'_{\#}$. Similarly for $(102)_{170}$, where well-formedness of $\Gamma, x\#\overline{xc}$ follows from $x\#\Gamma$, which is automatic because `let` $x = t_1$ `in` t_2 is actually $\texttt{Let}(t_1, t_2^*)$. In a ZF presentation we would need a side-condition. □

I think the point has been made. Should the reader become confused in the more complex proofs to follow about newness, he or she is referred to the above as a prototypical example of 'the genre'.

Remark 23.1.13 (Exciting)**.** Having said this, case `New` in T23.1.14 below is significantly more sophisticated than anything we have seen until now. There we need to choose x fresh in $\texttt{fresh}\, x\, \textbf{in}\ t'$ *twice*. We use $(23)_{41}$ the first time to choose *any* fresh x, and use $(21)_{41}$ the second time to choose the *same* x, observing that it is fresh for the context in which it is declared new the second time. See below.[79] ◇

[79] This is rather exciting, which is quite remarkable considering how boring syntax usually is. FM gives us something that even nonrigorous ZF does not: even if we are comfortable choosing 'new' variables in ZF and never mind about the rigour, there is no mechanism for deciding whether another variable we choose 'new' can safely be assumed equal to the old new variable. What allows us to do this in FM is simply that И is a quantifier with a well-defined syntactic scope, which we can easily observe if we write our propositions carefully. Furthermore, this scope distributes over conjunction (C9.4.5) so we can turn two new variables into one new variable where appropriate.

In fact we have seen this all before in Isabelle/FM. See §16.7 and in particular R16.7.1. The ability to assume two new variables equal was used time and time again in Isabelle/FM in situations just like T23.1.14.

We need the following result for T25.6 (soundness of apartness wrt evaluation), L24.1.8 (typing substitution property) and also C24.1.9 (overall substitution property of typing/apartness judgements).

Theorem 23.1.14 (Apartness Substitution Property)**.** *For V a possibly open value and t a possibly open term, if*

$$\Gamma'_{\#}, z\#\overline{zc} \vdash t\#yc \quad \textit{and} \quad \Gamma'_{\#} \vdash V\#\overline{zc}$$

then $\Gamma'_{\#}[V/z] \vdash t[V/z]\#yc[V/z]$.

Proof. By induction on apartness judgements $\Gamma_{\#} \vdash t\#yc$ using the hypothesis

$$(115)\quad \forall \Gamma'_{\#}, yc, \overline{zc}.\ \Big(\Gamma_{\#} \vdash t\#yc \ \wedge\ \Gamma_{\#} = (\Gamma'_{\#}, z\#\overline{zc}) \ \wedge\ \Gamma'_{\#} \vdash V\#\overline{zc}\Big) \implies \Gamma'_{\#}[V/z] \vdash t[V/z]\#yc[V/z].$$

1• *Consider the case* $(96)_{169}$. Suppose $t = xc.W$ and suppose we have $\Gamma_{\#}, \Gamma'_{\#}, yc, \overline{zc}$ such that

$$\Gamma_{\#} \vdash t\#yc \ \wedge\ \Gamma_{\#} = (\Gamma'_{\#}, z\#\overline{zc}) \ \wedge\ \Gamma'_{\#} \vdash V\#\overline{zc}.$$

Because $\Gamma_{\#} \vdash t\#yc$ holds it must be the case that

$$\Gamma_{\#} \vdash W\#yc \quad \text{and} \quad yc \neq xc.$$

By induction hypothesis we have

$$\Gamma'_{\#}[V/z] \vdash W[V/z]\#yc[V/z].$$

If it happens that $yc = z$ and $V = xc$ then we can deduce

$$\Gamma'_{\#}[V/z] \vdash xc[V/z].W[V/z]\#yc[V/z]$$

by $(97)_{169}$. Otherwise we can use $(96)_{169}$.

2• *Consider the case* $(99)_{169}$. Suppose we have $\Gamma_{\#}, \Gamma'_{\#}, yc, \overline{zc}$ such that

$$\Gamma_{\#} \vdash t\#yc \ \wedge\ \Gamma_{\#} = (\Gamma'_{\#}, z\#\overline{zc}) \ \wedge\ \Gamma'_{\#} \vdash V\#\overline{zc}.$$

We want to show

$$\Gamma'_{\#}[V/z] \vdash t[V/z]\#yc[V/z].$$

Choose x fresh such that $t = \texttt{fresh}\, x\, \textbf{in}\ t'$. This means, by the existential form of И $(23)_{41}$ that we may assume x is apart from every variable in the current context, which includes $\Gamma_{\#}, \Gamma'_{\#}, z, yc$ and $\overline{zc}$, and we do so. Because x is fresh, $(\texttt{fresh}\, x\, \textbf{in}\ t')[V/z] = \texttt{fresh}\, x\, \textbf{in}\ t'[V/z]$, so this equation really means

$$\Gamma'_{\#}[V/z] \vdash \texttt{fresh}\, x\, \textbf{in}\ t'[V/z]\#yc[V/z].$$

To prove this it suffices to show

$$(\Gamma'_{\#}[V/z])\#x \vdash t'[V/z]\#yc[V/z] \quad \text{and} \quad x \neq yc[V/z].$$

The second part is easy because x is fresh. We now prove the first part. Since $\Gamma_{\#} \vdash t\#yc$ we can resolve with $(\text{Real } (99))_{173}$ using the universal form of И $(21)_{41}$. This tells us that for some $x'\#\Gamma_{\#}, t, yc$,

$$\Gamma_{\#}\#x' \vdash t'\#yc.$$

We choose x' equal to the previous x.[80] $\Gamma_{\#}\#x$ (N23.1.8) is well-formed because $x\#\Gamma_{\#}$, and furthermore $x\#\Gamma'_{\#}, \overline{zc}$. So x occurs in them nowhere (L23.1.9, C13.2.1 and L9.3.6) and

$$\Gamma_{\#}\#x \;=\; (\Gamma'_{\#}, z\#\overline{zc})\#x \;=\; (\Gamma'_{\#}\#x), z\#(\overline{zc} \cup \{x\}).$$

We quantified over the $\overline{zc}$ in the induction hypothesis $(115)_{174}$ so we immediately have that

$$(\Gamma'_{\#}\#x)[V/z] \vdash t'[V/z]\#yc[V/z].$$

Now since x is apart from V and z,

$$\Gamma'_{\#}[V/z]\#x = (\Gamma'_{\#}\#x)[V/z] \subseteq (\Gamma'_{\#}\#x, z\#x)[V/z]$$

and we have

$$\Gamma'_{\#}[V/z] \vdash t[V/z]\#yc[V/z]$$

as required.

3• *Consider the case* $(102)_{170}$. Suppose we have $\Gamma_{\#}, \Gamma'_{\#}, yc, \overline{zc}$ such that

$$\Gamma_{\#} \vdash t\#yc \;\wedge\; \Gamma_{\#} = (\Gamma'_{\#}, z\#\overline{zc}) \;\wedge\; \Gamma'_{\#} \vdash V\#\overline{zc}.$$

and suppose $t = (\texttt{let}\ x = t_1\ \texttt{in}\ t_2)$ for x fresh (so apart from $\Gamma'_{\#}, \overline{zc}, z, t_1$ etc, but not t_2 because t is really $\texttt{Let}(t_1, t_{2*})$. We want to show

$$\Gamma'_{\#}[V/z] \vdash t[V/z]\#yc[V/z].$$

First of all, because x is chosen fresh and $x\#t1, V$,

$$(\texttt{let}\ x = t_1\ \texttt{in}\ t_2)[V/z] = (\texttt{let}\ x = t_1[V/z]\ \texttt{in}\ t_2[V/z])$$

so this equation really means

$$\Gamma'_{\#}[V/z] \vdash \texttt{let}\ x = t_1[V/z]\ \texttt{in}\ t_2[V/z]\#yc[V/z].$$

To show this it would suffice to prove for some $\overline{xc}$ that

$$\Gamma'_{\#}[V/z] \vdash t_1[V/z]\#\overline{xc}[V/z] \quad \text{and} \quad (\Gamma'_{\#}, x\#\overline{xc})[V/z] \vdash t_2[V/z]\#yc[V/z].$$

[80] See R23.1.13.

Since $\Gamma_\# \vdash t\#yc$ it must be the case that for some $\overline{xc}$,

$$\Gamma_\# \vdash t_1\#\overline{xc} \quad \text{and} \quad \Gamma_\#, x\#\overline{xc} \vdash t_2\#yc.$$

This first fact is a conjunction of judgements over all $xc \in \overline{xc}$ (N23.1.8) and because we quantified over the yc in our inductive hypothesis $(115)_{174}$, we have $(115)_{174}$ for each. We can deduce

$$\Gamma'_\#[V/z] \vdash t_1[V/z]\#\overline{xc}[V/z].$$

Recall that know $\Gamma_\#, x\#\overline{xc} \vdash t_2\#yc$—the comma is legal because $x\#\Gamma_\#$ (see N23.1.8 and C23.1.10). We can rewrite this:

$$\Gamma_\#, x\#\overline{xc} = (\Gamma'_\#, x\#\overline{xc}), z\#\overline{zc}.$$

We therefore have the inductive hypothesis for this too, so we can deduce

$$\Gamma'_\#[V/z], x\#\overline{xc}[V/z] \vdash t_2[V/z]\#yc[V/z].$$

As we observed above, this is enough to deduce

$$\Gamma'_\#[V/z] \vdash t[V/z]\#yc[V/z],$$

as required.

The other cases are longer or shorter, but they do not involve any new ideas. □

23.2. Discussion. Perhaps we should briefly run through these rules. We have not yet developed typing contexts (§24) so we tend to consider closed terms.

1• *Atom Constants.* Obvious. Denotational justification L9.3.6.

But incidentally, why did we allow constants $c \in \mathbf{AtmC}$ in the language, seeing as they were so deliberately excluded from the language of FM? It was convenient to do so and did not compromise the design aims of FreshML (see §30.1 and R21.8).

2• *Variables.* Note that $x\#y \vdash y\#x$ does *not* hold. Continuing R23.1.5 note that we can never deduce

$$\Gamma_\# \vdash x\#x.$$

The only rule that resolves against this is $(87)_{169}$ and this has the side condition $(x, x) \in \Gamma_\#$—but apartness contexts satisfy $(x, x) \notin \Gamma_\#$ (see D23.1.3). $x\#x$ is nonsense denotationally[81], so as in any logical system if we allow $\bot$ into our

[81] Indeed, for all atoms $a \in \mathbb{A}$ $\mathbf{Supp}(a) = \{a\}$ (L9.3.6).

context we can anticipate trouble.[82] We see this manifest itself in the technical details of typing (FALSE $\texttt{BV})_{166}$, which are dissected in a discussion culminating in $(145)_{185}$.

3• *Booleans.* $a\#\texttt{True}$ for all a is equivariance (N9.2.8).

4• *Natural Numbers.* Similar to `Bool`. The case-destructor deserves some comment. Note from Item 5 of N23.1.8 that $\Gamma_\#, x\#\overline{zc}$ makes sense when $(x, zc) \notin \Gamma_\#$ for any zc. Yet the rule itself has no side-condition to guarantee it. But this is automatic; being a bound variable symbol in a conclusion of an inductive rule, by R12.3.2 it must be chosen fresh for $\Gamma_\#, V, t_0, x.t_f, xc$ and everything else in the context at the moment of choice. In other words, it is a 'new' variable and only because of FM may we be quite confident it does not matter which 'new' variable we choose.

The remaining issue is whether $(\Gamma_\#)\#x$ in FM implies the necessary side-condition. It does, we use C23.1.10.

5• *Function Types.* This is less obvious. $f\#a$ for f a function is not decidable, so we do not even try to capture the behaviour of FM apartness. The side-condition of $(94)_{169}$ merely asserts "$\Gamma_\# \vdash f\#xc$ when we can be completely and utterly certain that this is the case".

Thinking in sets for a moment, note that $a \in \mathbf{Supp}(f)$ for $f = \lambda x.a.x$ even though for all x, $a\#f(x)$. Apartness rules based on 'extensional application reducing to lower types' would be false. In particular **anything similar to this version of** $(94)_{169}$ **is wrong**:

$$\text{(FALSE)} \qquad \frac{\Gamma_\#, f : \tau \to \tau' \# xc, x : \tau' \# xc \vdash t\#xc}{\Gamma_\# \vdash \texttt{fix}\, f(x : \tau')\, \texttt{in}\ t\#xc}.$$

We pay a certain price for our uncouthness. For example, we cannot prove

$$\vdash \texttt{fix}\, f(x : \mathbb{U})\, \texttt{in}\ \texttt{Fst}((0, a))\#a \tag{116}$$

even though the atom a is clearly thrown away. More significantly, because the rule for application is equally crude, we cannot deduce

$$\vdash (\texttt{fix}\, f(x : \Lambda_\alpha)\, \texttt{in}\ 0)\ a\#a. \tag{117}$$

This means for example that for some 'pretty printing' function f taking closed terms in Λ_α to de Bruijn terms expressed as elements of $(\texttt{Nat})\texttt{List}$, we shall

[82] In particular, results about apartness judgements which state that we *can't* prove something will need conditions to exclude nonsense contexts.

never be able to deduce

$$\vdash f(t)\#a$$

even though $f(t)$ is a list of natural numbers and therefore has a necessarily equivariant denotation.[83] We could extend the language with a rule stating that all terms of type `Nat` are apart from all xc of type `Atm`, but this brings problems of its own—we have no typing system yet, and if we can extend the language once, we can extend it again with exceptions which may allow calculations of type `Nat` to have nontrivial support (see [**66**, §8]). These issues are all further research (§35).

6• *Abstraction Types.* Direct from L9.3.4 for $(96)_{169}$ and L9.5.6 for $(97)_{169}$.

7• *Atom-Binding.* $\Gamma_\#\#x$ means "if x is new for the context $\Gamma_\#$ ... ". $t\#x$ means " ... and synthetically bound in the term t ... ". The conclusion means " ... then we can abstract over all new x".

8• *The other rules.* I hope fairly intuitive.

24. Typing Judgements

24.1. Definition. Typing judgements occur in a ***typing context*** $\Gamma = (\Gamma_{\mathbf{typ}}, \Gamma_\#)$. As usual $\Gamma_{\mathbf{typ}}$ is a finite partial function from term variables $x \in \mathbf{Var}$ to types τ. We call this the ***pure typing part*** of the typing context. $\Gamma_\#$ is an apartness context (D23.1.3), which we shall call the ***apartness part*** of the typing context.

$\Gamma_\#$ must satisfy the condition that for all $(xc, yc) \in \Gamma_\#$,

$$(xc, yc) \in (\mathbf{Dom}(\Gamma_{\mathbf{typ}}) \cup \mathbf{AtmC})^2.$$

Recall from R23.1.4 that the intended meaning of $(xc, yc) \in \Gamma_\#$ is $[\![xc]\!]\#[\![yc]\!]$ (N9.2.4).

Typing contexts will be written in list form

$$(x : \tau, \ldots, xc\#yc, \ldots, x : \tau)$$

where $x : \tau$ indicates that $(x, \tau) \in \Gamma_{\mathbf{typ}}$ and $xc\#yc$ indicates that $(xc, yc) \in \Gamma_\#$.

Definition 24.1.1. *Typing judgements are 3-tuples* (Γ, t, τ) *where* Γ *is a typing context,* t *is a term, and* τ *is a type. We shall write them*

$$\Gamma \vdash t : \tau.$$

[83] Thanks to Simon Peyton-Jones, ([**61**]) for raising this 'pretty-printing' function question during a conversation.

They are inductively defined according to the rules in Fig.34$_{180}$ *and Fig.35*$_{181}$ *using the notation of N24.1.2.*

Notation 24.1.2 (Typing judgement shorthand)**.** *We use the following shorthand notations Fig.34*$_{180}$ *and Fig.35*$_{181}$ *and the subsequent development:*

1. *We write*

$$\Gamma \vdash t : \tau \# xc \quad \textit{for} \quad \Gamma \vdash t : \tau \wedge \Gamma_{\#} \vdash t \# xc$$
$$\textit{and} \quad \Gamma \vdash t : \tau \# \overline{xc} \quad \textit{for} \quad \Gamma \vdash t : \tau \wedge \Gamma_{\#} \vdash t \# \overline{xc}.$$

   (Cf. Point 2 of N23.1.8.)
2. *We write*

$$\Gamma \cup x \# xc \quad \textit{for} \quad ((\Gamma_{\boldsymbol{typ}}\ ,\ (\Gamma_{\#} \cup x \# xc)).$$

   (Cf. Point 4 of N23.1.8.)
3. *We write*

$$\Gamma, x : \tau \# \overline{xc} \quad \textit{for} \quad ((\Gamma_{\boldsymbol{typ}} \cup \{x \mapsto \tau\})\ ,\ (\Gamma_{\#}, x \# \overline{xc}))$$

   on the condition that $x \notin \boldsymbol{Dom}(\Gamma_{\boldsymbol{typ}})$*. (Cf. Point 5 of N23.1.8.)*
4. *We write*

$$\Gamma \# x : \mathtt{Atm} \quad \textit{for} \quad ((\Gamma_{\boldsymbol{typ}} \cup \{x \mapsto \mathtt{Atm}\})\ ,\ (\Gamma_{\#} \# x)).$$

   (Cf. Point 9 of N23.1.8).

Notation 24.1.3 (Typeability)**.** *We say a term (or value)* t *is* ***typeable*** *in an environment* Γ *if there is a* τ *such that*

$$\Gamma \vdash t : \tau.$$

We write $\boldsymbol{CTerms}_{\tau}$ *for the set of closed terms typeable as* τ*. Similarly for* $\boldsymbol{CVal}_{\tau}$*.*

Here is a thoroughly routine lemma (but it is useful):

Lemma 24.1.4 (Typing of Values)**.** *For* U *a value and* Γ *a context then*

1. $\Gamma \vdash U : \mathtt{Atm}$ *iff* $U = c \in \boldsymbol{AtmC}$*.*
2. $\Gamma \vdash U : \mathtt{Bool}$ *iff* $U = \mathtt{True}$ *or* $U = \mathtt{False}$*.*
3. $\Gamma \vdash U : \mathtt{Nat}$ *iff* $U = 0$ *or* $U = \mathtt{Succ}(\ldots(0))$*.*
4. $\Gamma \vdash U : \tau \to \tau'$ *iff* $U = \mathtt{fix}\, f(x : \tau)\, \mathtt{in}\ t$ *for* $\Gamma, f : \tau \to \tau', x : \tau \vdash t : \tau'$*.*
5. $\Gamma \vdash U : \tau \times \tau'$ *iff* $U = (U_1, U_2)$ *such that* $\Gamma \vdash U_1 : \tau_1$ *and* $\Gamma \vdash U_2 : \tau_2$*.*
6. $\Gamma \vdash U : (\tau)\mathtt{List}$ *iff* $U = \mathtt{Nil}_{\tau}$ *or* $U = U_{hd} :: U_{tl}$ *such that* $\Gamma \vdash U_{hd} : \tau$ *and* $\Gamma \vdash U_{tl} : (\tau)\mathtt{List}$*.*
7. $\Gamma \vdash U : [\mathtt{Atm}]\tau$ *iff* $U = a.U'$ *for* $\Gamma \vdash U' : \tau$*.*
8. $\Gamma \vdash U : \Lambda_{\alpha}$ *iff* $\ldots$ `Var,App` $\ldots$ *or* $U = \mathtt{Lam}(U')$ *for* $\Gamma \vdash U' : [\mathtt{Atm}]\Lambda_{\alpha}$*.*

(118) $\Gamma \vdash c : \texttt{Atm} \quad (c \in \mathbf{AtmC})$ **Atom Constants**

(119) $\Gamma \vdash x : \tau \quad (\Gamma_{\mathbf{typ}}(x) = \tau)$ **Variables**

(120) $\Gamma \vdash \texttt{True} : \texttt{Bool}$ Bool

(121) $\Gamma \vdash \texttt{False} : \texttt{Bool}$

(122) $$\frac{\Gamma \vdash V : \texttt{Bool} \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash \texttt{if}\ V\ \texttt{then}\ t_1\ \texttt{else}\ t_2 : \tau}$$

(123) $\Gamma \vdash 0 : \texttt{Nat}$ Nat

(124) $$\frac{\Gamma \vdash V : \texttt{Nat}}{\Gamma \vdash \texttt{Succ}(V) : \texttt{Nat}}$$

(125) $$\frac{\Gamma \vdash V : \texttt{Nat} \quad \Gamma \vdash t_0 : \tau \quad \Gamma, x : \texttt{Nat} \# \overline{xc} \vdash t_f : \tau}{\Gamma \vdash \texttt{Case}\ V\ \texttt{of}\ \{0 \Rightarrow t_0, \texttt{Succ}(x) \Rightarrow t_f\} : \tau}$$

(126) $$\frac{\Gamma, f : \tau' \to \tau, x : \tau' \vdash t : \tau}{\Gamma \vdash \texttt{fix}\, f(x : \tau')\ \texttt{in}\ t : \tau' \to \tau}$$ **Function Types**

(127) $$\frac{\Gamma \vdash V_2 : \tau' \quad \Gamma \vdash V_1 : \tau' \to \tau}{\Gamma \vdash V_1\ V_2 : \tau}$$

(128) $$\frac{\Gamma \vdash xc : \texttt{Atm} \quad \Gamma \vdash V : \tau}{\Gamma \vdash xc.V : [\texttt{Atm}]\tau}$$ **Abstraction Types**

(129) $$\frac{\Gamma \vdash V : [\texttt{Atm}]\tau \# xc \quad \Gamma \vdash xc : \texttt{Atm}}{\Gamma \vdash V @ xc : \tau}$$

(130) $$\frac{\Gamma \# x : \texttt{Atm} \vdash t : \tau \# x}{\Gamma \vdash \texttt{fresh}\, x\ \textbf{in}\ t : \tau}$$ **Atom-Binding**

(131) $$\frac{\begin{array}{c}\Gamma \vdash xc : \texttt{Atm} \quad \Gamma \vdash yc : \texttt{Atm} \\ \Gamma \vdash V_3 : \tau \quad \Gamma \cup xc \# yc \vdash V_4 : \tau\end{array}}{\Gamma \vdash \texttt{if}\, xc = yc\, \texttt{then}\ V_3\ \texttt{else}\ V_4 : \tau} \quad xc \neq yc$$ **Atom-Destructor**

(132) $$\frac{\Gamma \vdash xc : \texttt{Atm} \quad \Gamma \vdash V_3 : \tau \quad \Gamma \vdash V_4 : \tau}{\Gamma \vdash \texttt{if}\, xc = xc\, \texttt{then}\ V_3\ \texttt{else}\ V_4 : \tau}$$

FIGURE 34. D24.1.1 - Typing 1

(133) $$\frac{\Gamma \vdash t_1 : \tau \# \overline{xc} \quad \Gamma, x : \tau \# \overline{xc} \vdash t_2 : \sigma}{\Gamma \vdash \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 : \sigma}$$ **Sequential Computation**

(134) $$\frac{\Gamma \vdash V : \tau \quad \Gamma \vdash V' : \tau'}{\Gamma \vdash (V, V') : \tau \times \tau'}$$ **Product Types**

(135) $$\frac{\Gamma \vdash V : \tau \times \tau'}{\Gamma \vdash \mathtt{Fst}(V) : \tau}$$

(136) $$\frac{\Gamma \vdash V : \tau \times \tau'}{\Gamma \vdash \mathtt{Snd}(V) : \tau'}$$

(137) $$\Gamma \vdash \mathtt{Nil}_\tau : \tau$$ **Lists**

(138) $$\frac{\Gamma \vdash V_1 : \tau \quad \Gamma \vdash V_2 : (\tau)\mathtt{List}}{\Gamma \vdash V_1 :: V_2 : (\tau)\mathtt{List}}$$

(139) $$\frac{\begin{array}{c}\Gamma \vdash V : (\tau)\mathtt{List} \# \overline{xa} \quad \Gamma \vdash t_{nil} : \sigma \\ \Gamma, x : \tau \# \overline{xa}, y : (\tau)\mathtt{List} \# \overline{xa} \vdash t_{cons} : \sigma\end{array}}{\Gamma \vdash \mathtt{Case}\ V\ \mathtt{of}\ \{\mathtt{Nil}_\tau \Rightarrow t_{nil}, x :: y \Rightarrow t_{cons}\} : (\tau)\mathtt{List}}$$

(140) $$\frac{\Gamma \vdash V : \mathtt{Atm}}{\Gamma \vdash \mathtt{Var}(V) : \Lambda_\alpha}$$ **λ-terms**

(141) $$\frac{\Gamma \vdash V : \Lambda_\alpha \times \Lambda_\alpha}{\Gamma \vdash \mathtt{App}(V) : \Lambda_\alpha}$$

(142) $$\frac{\Gamma \vdash V : [\mathtt{Atm}]\Lambda_\alpha}{\Gamma \vdash \mathtt{Lam}(V) : \Lambda_\alpha}$$

(143) $$\frac{\begin{array}{c}\Gamma \vdash V : \Lambda_\alpha \# \overline{xc} \quad \Gamma, x : \mathtt{Atm} \# \overline{xc} \vdash t_V : \tau \\ \Gamma, x : \Lambda_\alpha \times \Lambda_\alpha \# \overline{xc} \vdash t_A : \tau \quad \Gamma, x : [\mathtt{Atm}]\Lambda_\alpha \# \overline{xc} \vdash V_L : \tau\end{array}}{\Gamma \vdash \mathtt{Case}\ V\ \mathtt{of}\ \{\mathtt{Var}(x) \Rightarrow t_V, \mathtt{App}(x) \Rightarrow t_A, \mathtt{Lam}(x) \Rightarrow t_L\} : \tau}$$

FIGURE 35. D24.1.1 - Typing 2

PROOF. The right to left implication is in every case trivial by the typing rules.

The left to right implication follows by induction on the typing relation with induction hypotheses precisely equal to the implication. We prove the cases one at a time in sequence, since we need the case of $[\mathtt{Atm}]\tau$ and $\tau \times \tau'$ to handle Λ_α. □

The following lemma does for typing what C23.1.10 did for $\Gamma_\#$ and L12.4.2 before it did for Γ in FML*tiny*.

Lemma 24.1.5. *If* $\Gamma = (\Gamma_{\boldsymbol{typ}}, \Gamma_\#)$ *is an ordered pair of finite relations then* $a\#\Gamma$ *if and only if* $a\#x, y$ *for all* $(x, y) \in \Gamma_{\boldsymbol{typ}}$ *and all* $(x, y) \in \Gamma_\#$.

Proof. From L23.1.9 using L9.3.5 for pair-set to reduce $a\#(\Gamma_{\mathbf{typ}}, \Gamma_\#)$ to $a\#\Gamma_{\mathbf{typ}}$. and $a\#\Gamma_\#$. □

We need L24.1.5 in the development to follow, but we saw this kind of thing twice before in FML*tiny* and §23 and I shall no longer necessarily make a meal of it.

Lemma 24.1.6 (Apartness context-consistency)**.** *For all* Γ, t, τ *and* xc, *if*

$$\Gamma \vdash t : \tau \quad \textit{and} \quad \Gamma_\# \vdash t\#xc$$

then $xc \in \boldsymbol{Dom}(\Gamma_{\boldsymbol{typ}}) \cup \boldsymbol{AtmC}$.

Proof. By induction on $\Gamma_\# \vdash t\#xc$ using the hypothesis

$$\forall \Gamma', \tau.\ \left(\Gamma_\# = \Gamma'_\# \wedge \Gamma' \vdash t : \tau\#xc\right) \implies xc \in \mathbf{Dom}(\Gamma_{\mathbf{typ}}) \cup \mathbf{AtmC}.$$

□

Theorem 24.1.7 (Type Uniqueness)**.** *For* Γ, t *and* τ *such that* $\Gamma \vdash t : \tau$,

$$\forall \tau'.\ \Gamma \vdash t : \tau' \implies \tau' = \tau.$$

Proof. By induction on $\Gamma \vdash t : \tau$ using the hypothesis

$$\Gamma \vdash t : \tau \implies \forall \tau'.\ \Gamma \vdash t : \tau' \implies \tau' = \tau.$$

□

The design choices that ensure this theorem holds are that $\texttt{Nil}_\tau$ is labelled with a type, as are values of function-type $\texttt{fix}\, f(x : \tau)\, \texttt{in}\ t$. It would be more friendly for the user *not* to have to supply these type annotations, but harder on whoever is reading this proof, because non-uniqueness of typing makes it slightly more complex. For a simple example see R26.4.2.

Lemma 24.1.8 (Substitution in typing judgements)**.** *For* t *a term and* V *a value such that*

$$\Gamma, z : \tau\#\overline{zc} \vdash t : \sigma \quad \textit{and} \quad \Gamma \vdash V : \tau\#\overline{zc}$$

it is the case that

$$\Gamma[V/z] \vdash t[V/z] : \sigma.$$

PROOF. By induction on typing judgements $\Gamma \vdash t : \sigma$ using the hypothesis

$$(144)\quad \forall \Gamma', \tau.\ \Big(\Gamma \vdash t : \sigma\ \wedge\ \Gamma = (\Gamma', z\#\overline{zc})\ \wedge\ \Gamma' \vdash V : \tau\#\overline{zc}\Big)\implies \Gamma'[V/z] \vdash t[V/z] : \sigma.$$

Consider the case $(128)_{180}$**.** Suppose $t = xc.\,W$ and suppose we have Γ, Γ' such that

$$\Gamma \vdash t : \sigma\ \wedge\ \Gamma = (\Gamma', z\#\overline{zc})\ \wedge\ \Gamma' \vdash V : \tau\#\overline{zc}.$$

Because $\Gamma \vdash t : \sigma$ holds it must be the case that for some type σ',

$$\sigma = [\mathbb{A}]\sigma', \quad \Gamma \vdash W : \sigma' \quad \text{and} \quad \Gamma \vdash xc : \mathtt{Atm}.$$

By induction hypothesis we have

$$\Gamma'[V/z] \vdash W[V/z] : \sigma' \quad \text{and} \quad \Gamma'[V/z] \vdash xc[V/z] : \mathtt{Atm}.$$

We can then deduce

$$\Gamma'[V/z] \vdash xc[V/z].\,W[V/z] : [\mathtt{Atm}]\sigma',$$

by $(128)_{180}$ as required.

Consider the case $(129)_{180}$**.** Suppose $t = W@xc$ and suppose we have Γ, Γ' such that

$$\Gamma \vdash t : \sigma\ \wedge\ \Gamma = (\Gamma', z\#\overline{zc})\ \wedge\ \Gamma' \vdash V : \tau\#\overline{zc}.$$

Because $\Gamma \vdash t : \sigma$ holds it must be the case that

$$\Gamma \vdash W : [\mathtt{Atm}]\sigma \quad \text{and} \quad \Gamma_{\#} \vdash W\#xc.$$

By inductive hypothesis and T23.1.14 we know

$$\Gamma'[V/z] \vdash W[V/z] : [\mathtt{Atm}]\sigma \quad \text{and} \quad \Gamma'_{\#}[V/z] \vdash W[V/z]\#xc[V/z].$$

It follows that

$$\Gamma'[V/z] \vdash t[V/z] : \sigma,$$

as required.

Consider the case $(133)_{181}$**.** Suppose we have Γ, Γ' such that

$$\Gamma \vdash t : \sigma\ \wedge\ \Gamma = (\Gamma', z\#\overline{zc})\ \wedge\ \Gamma' \vdash V\#\tau\overline{zc}.$$

We want to show

$$\Gamma'[V/z] \vdash t[V/z] : \sigma.$$

Now choose fresh x apart from $\Gamma', \overline{zc}, z, \ldots$, with $t = \texttt{let}\ x = t_1\ \texttt{in}\ t_2$. Then $(\texttt{let}\ x = t_1\ \texttt{in}\ t_2)[V/z] = (\texttt{let}\ x = t_1[V/z]\ \texttt{in}\ t_2[V/z])$ so this equation really means

$$\Gamma'[V/z] \vdash \texttt{let}\ x = t_1[V/z]\ \texttt{in}\ t_2[V/z] : \sigma.$$

To show this it would suffice to prove for some $\overline{xc}$ and τ' that

$$\Gamma'[V/z] \vdash t_1[V/z] : \tau' \# \overline{xc}[V/z] \quad \text{and} \quad (\Gamma', x : \tau' \# \overline{xc})[V/z] \vdash t_2[V/z] : \sigma.$$

Since $\Gamma \vdash t : \sigma$ it must be the case that for some $\tau', \overline{xc}$,

$$\Gamma \vdash t_1 : \tau' \# \overline{xc} \quad \text{and} \quad \Gamma, x : \tau' \# \overline{xc} \vdash t_2 : \sigma.$$

This first fact is a conjunction of judgements over all $xc \in \overline{xc}$ (N23.1.8) and we have the inductive hypothesis $(144)_{183}$ for each. We can deduce

$$\Gamma'[V/z] \vdash t_1[V/z] : \tau' \# \overline{xc}[V/z].$$

We also know $\Gamma, x : \tau' \# \overline{xc} \vdash t_2 : \sigma'$. We have the inductive hypothesis for this too,[84] so we can deduce

$$\Gamma'[V/z], x : \tau' \# \overline{xc}[V/z] \vdash t_2[V/z] : \sigma.$$

As we observed above, this is enough to deduce

$$\Gamma'[V/z] \vdash t[V/z] : \sigma,$$

as required. □

Corollary 24.1.9. *For t a term and V a value such that*

$$\Gamma, z : \tau \# \overline{zc} \vdash t : \sigma \# \overline{yc} \quad \textit{and} \quad \Gamma \vdash V : \tau \# \overline{zc}$$

it is the case that

$$\Gamma[V/z] \vdash t[V/z] : \sigma \# \overline{yc}[V/z].$$

Proof. We combine T23.1.14 (substitution for apartness judgements) and L24.1.8 (substitution for pure typing judgements). □

[84] Using the fact that

$$(\Gamma', x : \tau' \# \overline{xc}), z \# \overline{zc} = \Gamma, x \# \overline{xc}.$$

The issue is as in T23.1.14 the commas separating the parts of these contexts, which have a particular meaning (N23.1.8). x was chosen fresh for all variables in the context at the time of its declaration, which included $\Gamma'_{\#}$, z, and $\overline{zc}$, so by C23.1.10 and C13.2.1 this equality works.

24.2. Discussion. Now that we have typing judgements we can type the example programs of §22.2. Going through them on paper would take far too much space. Let us briefly consider $(\texttt{Bij1})_{165}$:

$$t = \texttt{fix}\, f(x : [\texttt{Atm}](\tau \times \tau'))\, \texttt{in}\, \texttt{fresh}\, u\, \textbf{in}\, (u.\texttt{Fst}(x@u), u.\texttt{Snd}(x@u))$$

We wish to type this as a closed term:

$$\vdash t : [\texttt{Atm}](\tau \times \tau') \to [\texttt{Atm}]\tau \times [\texttt{Atm}]\tau'$$

so we resolve against $(126)_{180}$ to strip the function-abstraction and obtain

$$f : [\texttt{Atm}](\tau \times \tau') \to [\texttt{Atm}]\tau \times [\texttt{Atm}]\tau',\ x : [\texttt{Atm}](\tau \times \tau') \vdash \\ \texttt{fresh}\, u\, \textbf{in}\, (u.\texttt{Fst}(x@u), u.\texttt{Snd}(x@u)) : [\texttt{Atm}]\tau \times [\texttt{Atm}]\tau'$$

We resolve against $(130)_{180}$ and obtain the following proof-obligation:

$$f : [\texttt{Atm}](\tau \times \tau') \to [\texttt{Atm}]\tau \times [\texttt{Atm}]\tau' \# u,\ x : [\texttt{Atm}](\tau \times \tau') \# u,\ u : \texttt{Atm} \vdash \\ (u.\texttt{Fst}(x@u), u.\texttt{Snd}(x@u)) : [\texttt{Atm}]\tau \times [\texttt{Atm}]\tau' \# u.$$

This is shorthand for two obligations (see N24.1.2), a pure type judgement and a pure apartness judgement. I leave it to the reader to work through the rest of the details, they are not difficult.

More interesting are the FreshML terms which *do not* type, since it is they who cause all the trouble with "how do we know it does not matter which new name we pick" in naïve ZF-like theories of binding. Consider $(\text{FALSE}\ \ \texttt{BV})_{166}$:

$$\texttt{fix}\, f(x_* : [\texttt{Atm}]\tau)\, \texttt{in}\, \texttt{fresh}\, u\, \textbf{in}\, x_*@u$$

We try to type it just as above and eventually reach the following obligation:

$$\Gamma \ = \ \Big(f : [\texttt{Atm}]\tau \to \tau \# u, x_* : [\texttt{Atm}]\tau \# u,\ u : \texttt{Atm}\Big) \vdash x_*@u : \tau \# u.$$

The problem is the apartness judgement, which we now separate out:

$$\Gamma \vdash x_*@u \# u.$$

The only rule we can resolve with this is $(98)_{169}$, which gives us the obligations

$$\Gamma \vdash x_* \# u \quad \text{and} \quad \Gamma \vdash u \# u. \tag{145}$$

We can discharge the first with $(87)_{169}$ since $(x_*, u) \in \Gamma_\#$. We can *never* discharge the second, see Item 2 on *p.*176.

The other 'false' programs fail to type for the same reasons, and this is good. However, sometimes the logic of apartness judgements is less powerful than we might like. For example $(116)_{177}$ fails to type because $(104)_{170}$ is too crude and does not break V into (V_1, V_2) even though it could. But we were asking for

trouble writing such a silly program. $(117)_{177}$ is more serious. It fails to type because we have no way of expression in the type system that "f is such that no matter what the support of the argument x, $f(x)$ has no support". For similar reasons, considering $(\mathtt{Subst}(t,y))_{166}$, we cannot deduce

(FALSE) $$t:\Lambda_\alpha\#u, s:\Lambda_\alpha, u:\mathbb{A}\vdash \mathtt{Subst}(t,u)(s)\#u.$$

In conclusion, apartness judgements are not too bad on datatypes but hopeless on functions. Cf. §30.3.

Since the typing system has this weakness at higher orders, should we worry that the Sanity Clause T21.9 is not significant for future more sophisticated languages where better behaviour at higher-orders strengthens contextual equivalence? Actually, not necessarily. In contextual equivalence we quantify over all possible contexts $C[-]$. So if $C[-]$ contains function applications such that $C[t]$ refuses to type, a Kleene-equivalent form $C'[-]$ without the applications may well still do so. Of course FreshML is no good for programming, but we knew that already.

On the other hand, perhaps things are not that bad for programming either. Perhaps in FreshML a slightly different method of programming is more appropriate. Perhaps, instead of passing the atom name to `Subst` 'naked', we should bind it in the argument instead. What does this mean? Consider the following:

$(\mathtt{Subst}'(t))$ $$\mathtt{fix}\, f(x:[\mathtt{Atm}]\Lambda_\alpha)\,\mathtt{in}\,\mathtt{fresh}\, v\,\mathbf{in}\,\mathtt{Case}\, x\,\mathtt{of}\left\{\begin{array}{l}\mathtt{Var}(u)\Rightarrow \mathtt{if}\ u=y\ \mathtt{then}\ t\ \mathtt{else}\ v.\mathtt{Var}(u),\\ \mathtt{App}(s)\Rightarrow v.\mathtt{App}(f(\mathtt{Fst}(s)),f(\mathtt{Snd}(s))),\\ \mathtt{Lam}(s_*)\Rightarrow \mathtt{fresh}\, u\,\mathbf{in}\ v.\mathtt{Lam}(u.f(s_*@u))\end{array}\right\}$$

Even in our weak typing system, we can deduce

(VALID) $$t:\Lambda_\alpha\#u, s:\Lambda_\alpha, u:\mathtt{Atm}\vdash \mathtt{Subst}(t)(u.s)\#u.$$

But this is an unfinished story which I cannot tell here.

From now on we will show fewer examples and concentrate on developing the operational theory of the language, culminating in the Sanity Clause in §29.

25. Evaluation

Definition 25.1. *An **evaluation judgement** is a pair*

$$(t,V)\in \boldsymbol{CTerms}\times \boldsymbol{CVal}$$

of a typeable closed term and a typeable closed value. We write it

$$t\ \Downarrow\ V,$$

and define a big-step style reduction relation inductively as shown in Fig.36$_{188}$.

Notation 25.2. *We use the standard abbreviation*

$$(\texttt{let}\ x = s\ \texttt{in}\ (\texttt{let}\ x' = s'\ \texttt{in}\ t)) = (\texttt{let}\ x = s, x' = s'\ \texttt{in}\ t)$$

for nested let-expressions, for example in (147)$_{188}$.

Remark 25.3 (Evaluation nondeterminism)**.** The rule (163)$_{188}$ introduces a nondeterminism into the evaluation relation since any c not occurring textually in t may be chosen. This nondeterminism is not severe but is also not trivial to characterise. We shall try to because as it turns out there is no need. ◇

However, the following at least is true:

Lemma 25.4 (Evaluation on values)**.** *For all values V and V',*

$$V \Downarrow V \quad \textit{and} \quad (V \Downarrow V' \implies V = V').$$

Proof. Evident from Fig.36$_{188}$. □

Let us just explore these rules a little.

Remark 25.5 (Use of Let)**.** We use $\texttt{let}\ x = V\ \texttt{in}\ t$ to carry out substitutions 'within the syntax of the language' rather than the 'meta-level' alternative $t[V/x]$. This is nonstandard. For example, the standard presentation of the rule for applications would be

$$\frac{t[\texttt{fix}\, f(x:\tau)\,\texttt{in}\ t/f, U/x] \Downarrow V}{(\texttt{fix}\, f(x:\tau)\,\texttt{in}\ t)\ U \Downarrow V}.$$

Now this is written in nameful style (R4.14) which as discussed in R12.6.5 really means

$$\forall f, x.\ \forall t.\ \frac{t[\texttt{fix}\, f(x:\tau)\,\texttt{in}\ t/f, U/x] \Downarrow V}{(\texttt{fix}\, f(x:\tau)\,\texttt{in}\ t)\ U \Downarrow V} \quad \text{or equivalently (@@, D12.6.2)}$$

$$\frac{(t_{**}@@\texttt{Fix}(\tau, t_{**}))@@U \Downarrow V}{\texttt{Fix}(\tau, t_{**}) \Downarrow V}.$$

However, I have chosen to use the equivalent rule

$$\frac{\texttt{let}\ x = U, f = \texttt{fix}\, f(x:\tau)\,\texttt{in}\ t\ \texttt{in}\ t\ \Downarrow\ V}{(\texttt{fix}\, f(x:\tau)\,\texttt{in}\ t)\ U\ \Downarrow\ V},$$

(146) $V \Downarrow V \qquad V \in \mathbf{Val}$ **Values**

(147) $$\frac{\mathtt{let}\ x = U, f = \mathtt{fix} f(x:\tau)\,\mathtt{in}\ t\ \mathtt{in}\ t \Downarrow V}{(\mathtt{fix} f(x:\tau)\,\mathtt{in}\ t)\ U \Downarrow V}$$ **Functions**

(148) $$\frac{t_1 \Downarrow V_1 \quad t_2[V_1/x] \Downarrow V}{\mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 \Downarrow V}$$ **Sequential Calculation**

(149) $\mathtt{Fst}(V_1, V_2) \Downarrow V_1$ **Products**

(150) $\mathtt{Snd}(V_1, V_2) \Downarrow V_2$

(151) $$\frac{t \Downarrow V}{\mathtt{Case}\,\mathtt{Nil}_\tau\,\mathtt{of}\,\{\mathtt{Nil}_\tau \Rightarrow t, x::y \Rightarrow t'\} \Downarrow V}$$ **Lists**

(152) $$\frac{\mathtt{let}\ x = U, x' = U'\ \mathtt{in}\ t' \Downarrow V}{\mathtt{Case}\ U::U'\,\mathtt{of}\,\{\mathtt{Nil}_\tau \Rightarrow t, x::x' \Rightarrow t'\} \Downarrow V}$$

(153) $$\frac{t \Downarrow V}{\mathtt{Case}\,0\,\mathtt{of}\,\{0 \Rightarrow t, \mathtt{Succ}(x) \Rightarrow t'\} \Downarrow V}$$ `Nat`

(154) $$\frac{\mathtt{let}\ x = U\ \mathtt{in}\ t' \Downarrow V}{\mathtt{Case}\,\mathtt{Succ}(U)\,\mathtt{of}\,\{0 \Rightarrow t, \mathtt{Succ}(x) \Rightarrow t'\} \Downarrow V}$$

(155) $$\frac{\mathtt{let}\ x = U\ \mathtt{in}\ t \Downarrow V}{\mathtt{Case}\,\mathtt{Var}(U)\,\mathtt{of}\,\{\mathtt{Var}(x) \Rightarrow t, \mathtt{App}(x) \Rightarrow t', \mathtt{Lam}(x) \Rightarrow t''\} \Downarrow V}$$ Λ_α

(156) $$\frac{\mathtt{let}\ x = U\ \mathtt{in}\ t' \Downarrow V}{\mathtt{Case}\,\mathtt{App}(U)\,\mathtt{of}\,\{\mathtt{Var}(x) \Rightarrow t, \mathtt{App}(x) \Rightarrow t', \mathtt{Lam}(x) \Rightarrow t''\} \Downarrow V}$$

(157) $$\frac{\mathtt{let}\ x = U\ \mathtt{in}\ t'' \Downarrow V}{\mathtt{Case}\,\mathtt{Lam}(U)\,\mathtt{of}\,\{\mathtt{Var}(x) \Rightarrow t, \mathtt{App}(x) \Rightarrow t', \mathtt{Lam}(x) \Rightarrow t''\} \Downarrow V}$$

(158) $$\frac{t_1 \Downarrow V}{\mathtt{if}\ a = a\,\mathtt{then}\ t_1\ \mathtt{else}\ t_2 \Downarrow V}$$ **Atom-Destructor**

(159) $$\frac{t_2 \Downarrow V}{\mathtt{if}\ a = b\,\mathtt{then}\ t_1\ \mathtt{else}\ t_2 \Downarrow V} \quad (a \neq b)$$

(160) $$\frac{t_1 \Downarrow V}{\mathtt{if}\,\mathtt{True}\,\mathtt{then}\ t_1\ \mathtt{else}\ t_2 \Downarrow V}$$ `Bool`

(161) $$\frac{t_2 \Downarrow V}{\mathtt{if}\,\mathtt{False}\,\mathtt{then}\ t_1\ \mathtt{else}\ t_2 \Downarrow V}$$

(162) $$\frac{V' = (b\ a)\cdot V}{(a.V)@b \Downarrow V'} \quad (a.V \# b)$$ **Abstraction Types**

(163) $$\mathcal{N}c \in \mathbf{AtmC}.\ \frac{t[c/x] \Downarrow V}{\mathtt{fresh}\,x\,\mathbf{in}\ t \Downarrow V} \quad (x : \mathtt{Atm} \vdash t \# x)$$ **Atom-Binding**

$(b\ a)\cdot t$ the transposition action swapping b and a in t (R22.1.5).

FIGURE 36. D25.1 - Evaluation

which is really

$$\mathsf{И} f, x.\ \forall t.\ \frac{\texttt{let}\ x = U, f = \texttt{fix}\, f(x:\tau)\ \texttt{in}\ t\ \texttt{in}\ t\ \Downarrow\ V}{(\texttt{fix}\, f(x:\tau)\ \texttt{in}\ t)\ U\ \Downarrow\ V} \qquad \text{or}$$

$$\frac{(\texttt{Let}(U, \texttt{Let}(\texttt{Fix}(\tau, t_{**}), t_{**})) \Downarrow V}{\texttt{Fix}(\tau, t_{**}) \Downarrow V}.$$

This leaves the question of why I use the nonstandard formulation using `Let` instead of @@. My instinct is that if this proof were implemented in a theorem-proving environment (§31), the form I select would be easier to work with. Cf. also R19.2.4.

The reader is also invited to compare the phrasings of $(148)_{188}$ and $(75)_{97}$ in the light of R12.6.5 (the two rules correspond in the sense that they are where I have concentrated β-reduction). ◇

Theorem 25.6 (Apartness Soundness wrt Evaluation)**.** *For t a closed term and V a closed value, if*

$$t \Downarrow V \quad \textit{and} \quad \Gamma_{\#} \vdash t \# yc$$

then

$$\Gamma_{\#} \vdash V \# yc.$$

Proof. By induction on the evaluation relation with inductive hypothesis

$$t \Downarrow V \implies \forall \Gamma_{\#}, yc.\ \Gamma_{\#} \vdash t \# yc \implies \Gamma_{\#} \vdash V \# yc.$$

We consider only one case, $(148)_{188}$.

Suppose $t \Downarrow V$. Suppose we have $\Gamma_{\#}, yc$ such that

$$\Gamma_{\#} \vdash t \# yc.$$

Now we choose a fresh x such that $t = \texttt{let}\ x = t_1\ \texttt{in}\ t_2$. 'Fresh' means x is apart from all variables in the current context which includes $\Gamma_{\#}, t_1, V$ and yc.

There must exist V_1 such that

$$t_1 \Downarrow V_1 \quad \text{and} \quad t_2[V_1/x] \Downarrow V.$$

Then it must be the case that there is some $\overline{xc}$ such that

$$\Gamma_{\#} \vdash t_1 \# \overline{xc} \quad \text{and} \quad \Gamma_{\#}, x \# \overline{xc} \vdash t_2 \# yc$$

We have the inductive hypothesis for each judgement $\Gamma_{\#} \vdash t_1 \# xc$ for $xc \in \overline{xc}$ so from this first fact follows

$$\Gamma_{\#} \vdash V_1 \# \overline{xc}.$$

We can also apply T23.1.14 to the second fact and deduce

$$\Gamma_{\#}[V/x] \vdash t_2[V/x]\#yc[V/x].$$

However, x was chosen apart from $\Gamma_{\#}$ and yc so it does not appear in $\Gamma_{\#}$ (L23.1.9) and is not equal to yc (L9.3.6) so this is just

$$\Gamma_{\#} \vdash t_2[V/x]\#yc.$$

We can now apply the inductive hypothesis again to deduce

$$\Gamma_{\#} \vdash V\#yc,$$

as required. □

Theorem 25.7 (Type Soundness wrt Evaluation)**.** *For $t \in \boldsymbol{CTerms}$ a closed term and $V \in \boldsymbol{CVal}$ a closed value, if*

$$t \Downarrow V \quad \textit{and} \quad \Gamma \vdash t : \sigma$$

then

$$\Gamma \vdash V : \sigma.$$

Proof. By induction on $t \Downarrow V$ using inductive hypothesis

$$t \Downarrow V \implies \forall \Gamma, \tau.\ \Gamma \vdash t : \tau \implies \Gamma \vdash V : \tau.$$

We need C24.1.9 for cases like $t = \texttt{let}\ x = t_1\ \texttt{in}\ t_2$ where variables are added to the context by the corresponding typing rules ($(148)_{188}$, $(163)_{188}$, and similar). □

26. Bisimulation and Contextual Equivalence

26.1. Basic Definitions.

Definition 26.1.1 (Γ-Closures)**.** *Consider a context Γ as a list:*

$$\Gamma = (\Gamma_{\boldsymbol{typ}}, \Gamma_{\#}) \quad \textit{for} \quad \Gamma_{\boldsymbol{typ}} = (x_1, \sigma_1), \ldots, (x_n, \sigma_n).$$

We write $\sigma(xc)$ for σ_i if $xc = x_i \in \boldsymbol{Dom}(\Gamma_{\boldsymbol{typ}})$ and $\texttt{Atm}$ if $xc = c \in \boldsymbol{AtmC}$. We equate functions $\boldsymbol{Dom}(\Gamma_{\boldsymbol{typ}}) \to \boldsymbol{CVal}$ with lists of closed variables of length n.

*Then we say $\mathcal{V} = (V_1, \ldots, V_m)$ is a Γ-**closure** when $m = n$, and for $1 \leq i \leq n$,*

$$\emptyset \vdash V_i : \sigma_i,$$

and for each $(x, yc) \in \Gamma_{\#}$

$$\emptyset \vdash x\mathcal{V}\#yc\mathcal{V} \quad \textit{and} \quad \emptyset \vdash x\mathcal{V} : \sigma(xc).$$

Write the set of Γ-closures as

$$\boldsymbol{Closures}(\Gamma).$$

If $\Gamma \vdash t : \tau$ *and* $\mathcal{V} \in \boldsymbol{Closures}(\Gamma)$ *we write* $t\mathcal{V}$ *for* $t[V_i/x_i]$ *and call this a* ***closure*** *of* t*.*

The intended meaning of $\mathcal{V} \in \mathbf{Closures}(\Gamma)$ is

> "The V_i are possible values of the x_i in Γ. $t\mathcal{V}$ is therefore the closed term obtained by instantiating the x_i accordingly."

Definition 26.1.2 (Type-respecting relation)**.** *Let* $\mathcal{R}$ *denote the set of* ***type-respecting relations***

$$\left\{(s,t) \;\middle|\; s,t \in \boldsymbol{Terms} \wedge \exists \Gamma, \tau \in \boldsymbol{Typ}.\ \left(\Gamma \vdash s : \tau \wedge \Gamma \vdash t : \tau\right)\right\}.$$

Given $R \in \mathcal{R}$ *we write*

$$R(\Gamma, \tau) \stackrel{\text{def}}{=} \left\{(t,t') \in \boldsymbol{Terms} \times \boldsymbol{Terms} \;\middle|\; (t,t') \in R \ \wedge\ \Gamma \vdash t, t' : \tau\right\}.$$

When $(t,t') \in R(\Gamma,\tau)$ *we write*

$$\Gamma \vdash t\ R\ t' : \tau.$$

When we wish to include the condition that $\Gamma_{\#} \vdash t, t' \# xc$ *we write*

$$\Gamma \vdash t\ R\ t' : \tau \# xc.$$

We write $\mathcal{R}_{cl}$ *for the subset of* $\mathcal{R}$ *of such relations on closed terms.*

Remark 26.1.3 (Relations type-respecting)**.** We introduce $\mathcal{R}$ so to conveniently talk about relations between terms-in-context of the same type. Since only these relations interest us, **all relations on terms-in-context are henceforth type-respecting.** A corollary of this is that if $\Gamma \vdash t\ R\ t' : \tau$ holds, it *must* be the case that $\Gamma \vdash t, t' : \tau$. ◇

Notation 26.1.4. *Now T24.1.7 tells us that the typing of a term in a given context is unique if it exists, so actually the* τ *in* $\Gamma \vdash t\ R\ t' : \tau$ *is redundant. We leave it in anyway, except in the case when* $\Gamma = \emptyset$*, i.e. the terms* t, t' *are known to be closed. In that case we may abbreviate*

$$\Gamma \vdash t\ R\ t' : \tau \quad \textit{to} \quad t\ R\ t'.$$

Lemma 26.1.5. *Evaluation as a relation is in* $\mathcal{R}_{cl}$ *(D26.1.2).*

Proof. Since both t and V are by assumption typeable (D25.1), this is direct from T25.7. □

So when we write $t \Downarrow V$ we may assume they have the same type.

Definition 26.1.6 (Adequate relation)**.** *We call $R \in \mathcal{R}$ **adequate**[85] when for all $\Gamma, t, t', \mathcal{V}$ and $\tau \in \{\mathtt{Nat}, \mathtt{Bool}\}$, if $\Gamma \vdash t\ R\ t' : \tau$ and $\mathcal{V}$ is a Γ-closure (D26.1.1), then for all values V*

$$t\mathcal{V} \Downarrow V \implies t'\mathcal{V} \Downarrow V.$$

Write $\mathcal{R}_{ad}$ for this subset.

26.2. The Relation $\lhd_{\mathbf{ctx}}$.

Notation 26.2.1 (Congruence)**.** *A **congruence** is a relation on terms satisfying the rules of Fig.37$_{193}$ (which are derived in a consistent way from the typing rules of Fig.34$_{180}$ and Fig.35$_{181}$).*

Definition 26.2.2 (Contextual Preorder)**.** *The **contextual preorder**, written $\lhd_{\boldsymbol{ctx}}$, is defined coinductively as the **largest adequate congruence** (N26.2.1); the largest element of $\mathcal{R}_{ad}$ closed under the rules of Fig.37$_{193}$. We write $t \equiv_{\boldsymbol{ctx}} t'$ when $t \lhd_{\boldsymbol{ctx}} t'$ and $t' \lhd_{\boldsymbol{ctx}} t$.*[86]

Remark 26.2.3 (Unusual Construction)**.** Contextual equivalence is traditionally defined using 'contexts' with 'typed holes', $\mathcal{C}[-]$. Call this the "***context-and-hole approach***", as opposed to the "***adequate-congruence approach***"

[85] This is standard terminology in the field though what I call 'adequate' is more usually called '***preadequate***'. See for example [**42**, p41 §4.1] and [**65**, p8 Def2.2(iii)].

[86] In D26.2.2 it is not obvious that a largest $R \in \mathcal{R}_{ad}$ closed under the rules of Fig.37$_{193}$ exists. Suppose $S, T \in \mathcal{R}_{ad}$ are such. Is it the case that $S \cup T$ is closed under the rules? No. Consider $(170)_{193}$. It may be that

$$\Gamma \vdash V_2\ S\ V_2' : \tau' \quad \text{and} \quad \Gamma \vdash V_1\ T\ V_1' : \tau' \to \tau$$

but not

$$\Gamma \vdash V_2\ T\ V_2' : \tau' \quad \text{or} \quad \Gamma \vdash V_1\ S\ V_1' : \tau' \to \tau.$$

Thus $S \cup T$ is not necessarily closed under $(170)_{193}$.

There is a 'patch' for this. Call the rules in Fig.37$_{193}$ by the name **Rules**. We consider a modified version **Rules'**. Where there are two troublesome conditions like in $(170)_{193}$ in **Rules**, **Rules'** splits the rule in two:

$$\frac{\Gamma \vdash V_2 \lhd_{\mathbf{ctx}} V_2' : \tau'}{\Gamma \vdash V_1\ V_2 \lhd_{\mathbf{ctx}} V_1\ V_2' : \tau} \quad \text{and} \quad \frac{\Gamma \vdash V_1 \lhd_{\mathbf{ctx}} V_1' : \tau' \to \tau}{\Gamma \vdash V_1\ V_2 \lhd_{\mathbf{ctx}} V_1'\ V_2 : \tau}.$$

There is clearly a largest set closed under **Rules'**, call it $\lhd_{\mathbf{ctx\text{-}cl}}$. A set closed under **Rules** is closed under **Rules'** so if we can show $\lhd_{\mathbf{ctx\text{-}cl}}$ is closed under **Rules** as well, it must be the largest such. It suffices to show that $\lhd_{\mathbf{ctx\text{-}cl}}$ is transitive. This is easily done by showing that the relation

$$\left\{(x, y) \mid \exists z.\ x \lhd_{\mathbf{ctx\text{-}cl}} z \wedge z \lhd_{\mathbf{ctx\text{-}cl}} y\right\}$$

is closed under **Rules'**. Lassen has this problem in [**42**] and addresses it in some generality in [**42**, p.29, §3.7].

(164) $\Gamma \vdash c \lhd_{\mathbf{ctx}} c : \mathtt{Atm}$ **Atom Constants**

(165) $$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash x \lhd_{\mathbf{ctx}} x : \tau}$$ **Variables**

(166) $\Gamma \vdash \mathtt{True} \lhd_{\mathbf{ctx}} \mathtt{True} : \mathtt{Bool}$ `Bool`

(167) $\Gamma \vdash \mathtt{False} \lhd_{\mathbf{ctx}} \mathtt{False} : \mathtt{Bool}$

(168) $$\frac{\Gamma \vdash V \lhd_{\mathbf{ctx}} V' : \mathtt{Bool} \quad \Gamma \vdash t_1 \lhd_{\mathbf{ctx}} t_1' : \tau \quad \Gamma \vdash t_2 \lhd_{\mathbf{ctx}} t_2' : \tau}{\Gamma \vdash \mathtt{if}\ V\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 \lhd_{\mathbf{ctx}} \mathtt{if}\ V\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 : \tau}$$

... **more rules according to the same pattern, omitted** ...

(169) $$\frac{\Gamma, f : \tau' \to \tau, x : \tau' \vdash t \lhd_{\mathbf{ctx}} t' : \tau}{\Gamma \vdash \mathtt{fix} f(x : \tau')\,\mathtt{in}\ t \lhd_{\mathbf{ctx}} \mathtt{fix} f(x : \tau')\,\mathtt{in}\ t' : \tau' \to \tau}$$ **Function Types**

(170) $$\frac{\Gamma \vdash V_2 \lhd_{\mathbf{ctx}} V_2' : \tau' \quad \Gamma \vdash V_1 \lhd_{\mathbf{ctx}} V_1' : \tau' \to \tau}{\Gamma \vdash V_1\ V_2 \lhd_{\mathbf{ctx}} V_1'\ V_2' : \tau}$$

(171) $$\frac{\Gamma \vdash V \lhd_{\mathbf{ctx}} V' : \tau \quad \Gamma \vdash xc : \mathtt{Atm}}{\Gamma \vdash xc.V \lhd_{\mathbf{ctx}} xc.V' : [\mathtt{Atm}]\tau}$$ **Abstraction Types**

(172) $$\frac{\Gamma \vdash V \lhd_{\mathbf{ctx}} V' : [\mathtt{Atm}]\tau \quad \Gamma \vdash V, V' \# xc}{\Gamma \vdash V @ xc \lhd_{\mathbf{ctx}} V' @ xc : \tau}$$

(173) $$\frac{\Gamma \vdash V_1 \lhd_{\mathbf{ctx}} V_1' : \tau_1 \quad \Gamma \vdash V_2 \lhd_{\mathbf{ctx}} V_2' : \tau_2}{\Gamma \vdash (V_1, V_2) \lhd_{\mathbf{ctx}} (V_1', V_2') : \tau_1 \times \tau_2}$$ **Product Types**

(174) $$\frac{\Gamma \vdash V \lhd_{\mathbf{ctx}} V' : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{Fst}(V) \lhd_{\mathbf{ctx}} \mathtt{Fst}(V') : \tau_1}$$

(175) $$\frac{\Gamma \vdash V \lhd_{\mathbf{ctx}} V' : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{Snd}(V) \lhd_{\mathbf{ctx}} \mathtt{Snd}(V') : \tau_2}$$

(176) $$\frac{\Gamma \# x : \mathtt{Atm} \vdash t \lhd_{\mathbf{ctx}} t' : \tau \quad \Gamma \# x : \mathtt{Atm} \vdash t, t' \# x}{\Gamma \vdash \mathtt{fresh}\, x\, \mathbf{in}\ t \lhd_{\mathbf{ctx}} \mathtt{fresh}\, x\, \mathbf{in}\ t' : \tau}$$ **Atom-Binding**

... **more rules according to the same pattern, omitted** ...

(177) $$\frac{\Gamma \vdash V \lhd_{\mathbf{ctx}} V' : [\mathtt{Atm}]\Lambda_\alpha}{\Gamma \vdash \mathtt{Lam}(V) \lhd_{\mathbf{ctx}} \mathtt{Lam}(V') : \Lambda_\alpha}$$ **λ-terms**

(178)
$$\frac{\begin{array}{c}\Gamma \vdash V \lhd_{\mathbf{ctx}} V' : \Lambda_\alpha \quad \Gamma \vdash V, V' \# \overline{xc} \\ \Gamma, a : \mathtt{Atm} \# \overline{xc} \vdash t_V \lhd_{\mathbf{ctx}} t_V' : \tau \\ \Gamma, x : \Lambda_\alpha \times \Lambda_\alpha \# \overline{xc} \vdash t_A \lhd_{\mathbf{ctx}} t_A' : \tau \\ \Gamma, x : [\mathtt{Atm}]\Lambda_\alpha \# \overline{xc} \vdash t_L \lhd_{\mathbf{ctx}} t_L' : \tau\end{array}}{\Gamma \vdash \begin{array}{c}\mathtt{Case}\ V\ \mathtt{of}\ \{\mathtt{Var}(x) \Rightarrow t_V, \mathtt{App}(x) \Rightarrow t_A, \mathtt{Lam}(x) \Rightarrow t_L\} \lhd_{\mathbf{ctx}} \\ \mathtt{Case}\ V'\ \mathtt{of}\ \{\mathtt{Var}(x) \Rightarrow t_V', \mathtt{App}(x) \Rightarrow t_A', \mathtt{Lam}(x) \Rightarrow t_L'\}\end{array} : \tau}$$

FIGURE 37. D26.2.2 - Contextual Preorder $\lhd_{\mathbf{ctx}}$

used in D26.2.2. The problem with contexts-and-holes is that they are usually only informally defined; see [**64**, p.250-252, 'PCFL Contexts'] (the paper on which my overall method of proof is based) or [**63**, p.4 Def 2.1]. What do I mean? For example, hardly anyone gives a complete grammar for forming contexts. If I followed contexts-and-holes I would have to choose to either render it impossible from the start to prove T21.9 with complete rigour by using a hand-wavy definition of contexts, or to give a proper grammar for them. The latter choice amounts to the adequate-congruence approach I have followed, with one important philosophical difference: adequate-congruences bring contextual equivalence in line with the relations it is eventually proved equal to (C28.24)—a (co)inductively defined relation on syntactic datatype.[87] ◇

Lemma 26.2.4 ($\lhd_{\mathbf{ctx}}$ transitive and reflexive)**.** *$\lhd_{ctx}$ (D26.2.2) is transitive and reflexive.*

PROOF. The method is standard. For reflexivity it suffices to show that the identity relation is an adequate congruence. It is certainly a congruence (Fig.37_{193}), so we consult D26.1.6 and see it is adequate as well.

For transitivity we must apply the same argument to the set

$$\left\{(s,u) \in \mathbf{Terms} \times \mathbf{Terms} \;\middle|\; \exists t \in \mathbf{Terms}.\ s \lhd_{\mathbf{ctx}} t \wedge t \lhd_{\mathbf{ctx}} u\right\}.$$

Again, this is an adequate congruence. □

We shall not pursue the contextual preorder immediately. Instead we shall define a notion of bisimulation on terms-in-context, first between closed terms and then extending to open terms by syntactic instantiation of free variable symbols with closed values. Ultimately, we shall seek to prove that the bisimulation defined below ($\lhd$, D26.4.1) and the contextual preorder defined above ($\lhd_{\mathbf{ctx}}$, D26.2.2) coincide.

26.3. The Equivalence $\equiv^{se}$ and the Relation $\leq_{\mathbf{kl}}$. We devote a short subsection to $\equiv^{se}$ and $\leq_{\mathbf{kl}}$. $\equiv^{se}$ is a technical construction which will soon be useful, specifically in T26.4.7 and T27.11. $\leq_{\mathbf{kl}}$ is a standard relation which we take the opportunity of this clearing in the mathematics to introduce.

[87] We see this trivially in L26.2.4. More seriously, the reader is referred to the proofs of L28.19 and C28.24.

Definition 26.3.1. *Let $\equiv^{se}$ be the least equivalence relation on closed values of type Λ_α that is a congruence and satisfies the following rule:*[88]

$$(179) \qquad \text{И}c.\ \Big(\frac{(c\ a)\cdot t \equiv^{se} (c\ a')\cdot t'}{\mathtt{Lam}(a.t) \equiv^{se} \mathtt{Lam}(a'.t')}\Big) \qquad a \neq a'$$

*If two values V, V' are such that $V \equiv^{se} V'$ we say V and V' are **synthetically equivalent**.*

Lemma 26.3.2 ($\equiv^{se}$ equivalence relation)**.** *The relation $\equiv^{se}$ of D26.3.1 above is an equivalence relation.*

Proof. In C9.4.5 we proved that $(\text{И}x.\ \Phi) \wedge (\text{И}x.\ \Psi) \iff \text{И}x.\ (\Phi \wedge \Psi)$, and this suffices to establish the result. □

Remark 26.3.3 ($\equiv^{se}$ morally $=_\alpha$)**.** Actually, we proved back in T8.2.5 (before any of this theory) that $\equiv^{se}$ coincides with α-equivalence on untyped λ-terms, which itself coincides with equality in the denotation $\mathbf{L}$ (λ-terms up to α-equivalence, D10.3.4) by the general nonsense of T10.5.8.

Why do we not simply write $\equiv^{se}$ as $=_\alpha$? Because α-equivalence on terms of FreshML is a completely different creature to do with FreshML variables, and nothing to do with atoms in FM represented in the language by $a \in \mathbf{AtmC}$ (and variables $x : \mathtt{Atm}$). So we needed a new symbol for the concept. ◇

Definition 26.3.4 (Kleene Ordering)**.** *We also define the **Kleene ordering** on closed terms-in-context, written $\leq_{\mathbf{kl}}$ as follows:*

$$\vdash s \leq_{\mathbf{kl}} t : \tau \iff \vdash s, t : \tau \wedge \forall V \in \mathbf{Val}.\ \big(\vdash V : \tau \wedge s \Downarrow V\big) \implies t \Downarrow V.$$

We write $\vdash s =_{\mathbf{kl}} t : \tau$ when $\vdash s \leq_{\mathbf{kl}} t : \tau$ and $\vdash t \leq_{\mathbf{kl}} s : \tau$.

Theorem 26.3.5. *$\leq_{\mathbf{kl}}$ is transitive and reflexive.*

Proof. Easily from the definition D26.3.4. □

26.4. The Relation $\lhd$.

Definition 26.4.1 (Bisimulation)**.** *We coinductively define a relation $\lhd$ on closed terms-in-context as the greatest (post-)fixed point of the set-operator given in Fig.38*$_{196}$.

Remark 26.4.2. Recall that $\lhd \in \mathcal{R}_{cl}$ is a type-respecting relation on closed terms-in-context (see D26.1.2), which means that judgements should be written

$$\vdash s \lhd t : \tau$$

[88] When doing induction on $\equiv^{se}$ we would like backwards proof search to be linear. The side condition $a \neq a'$ keeps the rule disjoint from simple congruence.

$$
\begin{array}{rcll}
\Phi\colon \mathcal{R}_{cl} \longrightarrow \mathcal{R}_{cl} & & & \\
\Phi(R)(s,t) & \iff & & \\
\quad \forall c \in \mathbf{AtmC}.\ s \Downarrow c & \Longrightarrow & t \Downarrow c & \wedge \\
\quad s \Downarrow \mathtt{True} & \Longrightarrow & t \Downarrow \mathtt{True} & \wedge \\
\quad s \Downarrow \mathtt{False} & \Longrightarrow & t \Downarrow \mathtt{False} & \wedge \\
\quad s \Downarrow 0 & \Longrightarrow & t \Downarrow 0 & \wedge \\
\quad s \Downarrow \mathtt{Nil}_\tau & \Longrightarrow & t \Downarrow \mathtt{Nil}_\tau & \wedge \\
\quad \forall a, s'.\ s \Downarrow a.s' & \Longrightarrow & \exists b, t'.\ t \Downarrow b.t' \wedge & \wedge \\
\quad & & \mathsf{И} c.\ (c\ a)\cdot s'\ R\ (c\ b)\cdot t' & \wedge \\
\forall x, s', \tau.\ s \Downarrow \mathtt{fix} f(x:\tau)\ \mathtt{in}\ s' & \Longrightarrow & \exists t'.\ t \Downarrow \mathtt{fix} f(x:\tau)\ \mathtt{in}\ t' \wedge & \\
\quad & & \forall U \in \mathbf{CVal}_\tau.\ (\mathtt{fix} f(x:\tau)\ \mathtt{in}\ s')\, U R & \wedge \\
\quad & & (\mathtt{fix} f(x:\tau)\ \mathtt{in}\ t')\, U & \wedge \\
\forall U_1, U_2.\ s \Downarrow (U_1, U_2) & \Longrightarrow & \exists V_1, V_2.\ U_1\ R\ V_1 \wedge U_2\ R\ V_2 \wedge t \Downarrow (V_1, V_2) & \wedge \\
\forall U.\ s \Downarrow \mathtt{Succ}(U) & \Longrightarrow & \exists V.\ U\ R\ V \wedge t \Downarrow \mathtt{Succ}(V) & \wedge \\
\forall U.\ s \Downarrow U_h :: U_t & \Longrightarrow & \exists V_h, V_t.\ U_h\ R\ V_h \wedge U_t\ R\ V_t \wedge t \Downarrow V_h :: V_t & \wedge \\
\forall U.\ s \Downarrow \mathtt{Var}(U) & \Longrightarrow & \exists V.\ U\ R\ V \wedge t \Downarrow \mathtt{Var}(V) & \wedge \\
\forall U.\ s \Downarrow \mathtt{App}(U) & \Longrightarrow & \exists V.\ U\ R\ V \wedge t \Downarrow \mathtt{App}(V) & \wedge \\
\forall U.\ s \Downarrow \mathtt{Lam}(U) & \Longrightarrow & \exists V.\ U\ R\ V \wedge t \Downarrow \mathtt{Lam}(V) & \wedge
\end{array}
$$

All values and terms closed.

FIGURE 38. D26.4.1 - Bisimulation $\lhd$

for appropriate τ. However, because of T24.1.7 (type uniqueness) we can deduce τ from s or t, so we just write

$$s \lhd t.$$

Bear in mind that we can deduce from the judgement $s \lhd t$ that s and t are closed typeable terms of the same (unique) type. $\diamond$

Remark 26.4.3 (Significance of $\lhd$)**.** $\lhd$ implements a notion of equivalence on terms based on 'reducing' a term's type (by evaluating it and attacking the value's structure in a manner appropriate to the type) until we can do so no longer, and then observing the result. Then the coinductive definition expresses the slogan

"$s \lhd t$ when any behaviour s can display, t can display."

$\diamond$

Lemma 26.4.4 ($\lhd$ equivariant)**.** *$\lhd$ is equivariant:*

$$s \lhd t \iff (b\ a)\cdot s \lhd (b\ a)\cdot t.$$

Proof. To prove this we jump all the way back to Chapter II and L8.1.12. In the terminology of that lemma $\lhd$ is an unparameterised 0-ary function-constant and by that lemma $(a\ b)\cdot \lhd = \lhd$. $\in$ is equivariant so

$$(x, y) \in \lhd \iff (a\ b)\cdot(x, y) \in (a\ b)\cdot \lhd = \lhd\ .$$

Permutation commutes with pairset by R8.1.13 so we are done. □

Theorem 26.4.5 ($\lhd$ trans. and refl.)**.** *$\lhd$ is transitive and reflexive.*

Proof. We observe that the set

$$\{(t, t) \in \mathcal{R}_{cl}\}$$

is a post-fixed point of Φ, as is the set

$$\left\{(s, t) \;\middle|\; \exists t'.\ s \lhd t' \wedge t' \lhd t\right\}.$$

□

The following technical lemma is one half of T26.4.7:

Lemma 26.4.6 ($\Downarrow \subseteq \lhd$)**.** *For t a closed term and U a closed value respectively,*

$$t \Downarrow U \implies U \lhd t.$$

Proof. By L25.4 that $U \Downarrow U$ so $U \leq_{\mathbf{kl}} t$. The rest is L26.4.8. □

The main use of the following theorem is in T27.11, a corresponding result for $\lhd^*$.

Theorem 26.4.7 ($\lhd$ on closed values)**.** *For closed values U and closed terms t, the lemmas of Fig.39$_{198}$ hold.*

Proof. The results to be proved are all of the form $A \implies (B \iff C)$, so split into two implications. Those of the form $A \implies (C \implies B)$ follow from L26.4.6 above. So now we may assume this and proceed to prove the $A \implies (B \implies C)$ part.

Each case is by induction on the typing $\Gamma \vdash U : \tau$ using the predicate $A \implies (B \implies C)$ as induction hypothesis. We use L24.1.4 to relate the type to the syntactic form of U, L25.4, and the fact that $\Phi(\lhd) = \lhd$.

$$
\begin{aligned}
\vdash U, t:\mathtt{Atm} &\implies U \lhd t \iff t \Downarrow U \\
\vdash U, t:\mathtt{Bool} &\implies U \lhd t \iff t \Downarrow U \\
\vdash U, t:\mathtt{Nat} &\implies U \lhd t \iff t \Downarrow U \\
\vdash (U_1, U_2), t:\tau \times \tau' &\implies (U_1, U_2) \lhd t \iff \exists V_1, V_2. \\
&\qquad\qquad U_1 \lhd V_1 \wedge U_2 \lhd V_2 \wedge t \Downarrow (V_1, V_2) \\
\vdash U, t:\Lambda_\alpha &\implies U \lhd t \iff \exists V \equiv^{se} U.\ t \Downarrow V \\
\vdash U, t:(\tau)\mathtt{List} &\implies U \lhd t \iff \exists V.\ t \Downarrow V \wedge \big((U = V = \mathtt{Nil}_\tau) \vee \\
&\quad \exists U_h, U_t, V_h, V_t.\ (U = U_h{::}U_t \ \wedge\ V = V_h{::}V_t \ \wedge \\
&\qquad\qquad U_h \lhd V_h \ \wedge\ U_t \lhd V_t)\big) \\
\vdash U, t:[\mathtt{Atm}]\tau &\implies U \lhd t \iff \exists a, W, a', W'. \\
&\quad U = a.W \wedge t \Downarrow a'.W' \wedge \text{И}c.(c\ a)\cdot W \lhd (c\ a')\cdot W' \\
\vdash U, t:\tau' \to \tau &\implies U \lhd t \iff \exists V.\ t \Downarrow V \wedge \forall W.\ U\ W \lhd V\ W.
\end{aligned}
$$

$U, W, V \in \mathbf{CVal}$ and $t \in \mathbf{CTerms}$.

FIGURE 39. T26.4.7 - Lemmas of $\lhd$

1• *Consider the case of* `Nat`. By induction on the typing $\Gamma \vdash U : \tau$ using induction hypothesis

$$\vdash U:\mathtt{Nat} \implies U \lhd t \iff t \Downarrow U.$$

So suppose $\vdash U:\mathtt{Nat}$. Then by the typing rules it must be the case that

$$U = 0 \quad \text{or} \quad U = \mathtt{Succ}(U'),$$

for $\vdash U':\mathtt{Nat}$. We treat only the second case. Since $\Phi(\lhd) = \lhd$ we know

$$\exists V'.\ U' \lhd V' \wedge t \Downarrow \mathtt{Succ}(V).$$

Let V' be one such. Since $U' \lhd V'$ we know by L26.4.6 that $V' \Downarrow U'$ and hence by L25.4 that $V' = U'$. So $V = \mathtt{Succ}(U') = U$ and $t \Downarrow U$ as required. We consider only one other case, the only one of any novelty.

2• *Consider the case* Λ_α. L24.1.4 has three possibilities for $\vdash U : \Lambda_\alpha$; $U = \mathtt{Var}(a)$, $U = \mathtt{App}(U_1, U_2)$, and $U = \mathtt{Lam}(a.U')$. We consider only this last one.

So suppose

$$U = \mathtt{Lam}(a.U'), \quad \vdash U, t:\Lambda_\alpha \quad \text{and} \quad U \lhd t.$$

By this third fact we know there is some value V (which is of abstraction type and so by L24.1.4 is) of the form $b.V'$, such that

$$t \Downarrow \mathtt{Lam}(b.V') \quad \text{and} \quad a.U' \lhd b.V'.$$

We follow back $\lhd$ and deduce that

$$И c.\ (c\ a)\cdot U' \lhd (c\ b)\cdot V'.$$

By L25.4 and the induction hypothesis we can deduce

$$И c.\ (c\ a)\cdot U' \equiv^{se} (c\ b)\cdot V'.$$

Recall from the definition of $\equiv^{se}$ (D26.3.1) that this is precisely the condition for

$$U = a.U' \equiv^{se} a.V' = V.$$

This gives us the result.

□

Lemma 26.4.8 ($\leq_{\mathbf{kl}} \subseteq \lhd$)**.** *The relation $\leq_{kl}$ is a subrelation of $\lhd$. That is, $\leq_{kl} \subseteq \lhd$.*

Proof. By construction of $\lhd$ we need only show that

$$\leq_{\mathbf{kl}} \subseteq\ \Phi(\leq_{\mathbf{kl}}).$$

This we see immediately by inspection of Φ as defined in Fig.38_{196}. □

Corollary 26.4.9. *If $s =_{kl} s'$ then $s \lhd t \iff s' \lhd t$, and similarly if $t =_{kl} t'$.*

Proof. By L26.4.8 above and T26.4.5 ($\lhd$ transitive and reflexive). □

Remark 26.4.10. In the light of C26.4.9 we see we could have chosen any Kleene-equivalent terms in the definition of Φ in Fig.38_{196}. For example, the clauses for values of abstraction and function types could have been

$$\forall a, s'.\ s \Downarrow a.s' \implies \exists b, t'.\ t \Downarrow b.t' \wedge И c.\ (a.s')@c\ R\ (b.t')@c$$

$$\forall x, s', \tau.\ s \Downarrow \mathtt{fix} f(x:\tau)\ \mathtt{in}\ s' \implies \exists t'.\ t \Downarrow \mathtt{fix} f(x:\tau)\ \mathtt{in}\ t' \wedge$$
$$\forall U \in \mathbf{CVal}_\tau.\ (\mathtt{let}\ x = U, f = (\mathtt{fix} f(x:\tau)\ \mathtt{in}\ s')\ \mathtt{in}\ s'\ R$$
$$(\mathtt{let}\ x = U, f = (\mathtt{fix} f(x:\tau)\ \mathtt{in}\ t')\ \mathtt{in}\ t'.$$

In general I chose whichever possibility took less space on the page. ◇

We shall need the following result to prove the very similar, but much more complex, T27.14.

Corollary 26.4.11 ($\lhd$ Evaluation Box)**.** *For all closed terms t, t' and values V if*

$$t \Downarrow V \quad \textit{and} \quad t \lhd t'$$

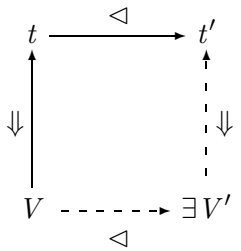
then there exists some V' such that

$$t' \Downarrow V' \quad \textit{and} \quad V \lhd V'$$

Proof. By L26.4.8 ($\leq_{\mathbf{kl}} \subseteq \lhd$) and L26.4.6 ($\Downarrow \subseteq \lhd$). □

We draw a diagram of this result because it is important.

(180)



26.5. The Relation $\lhd^\circ$. Recall we defined Γ-closures in D26.1.1 and the set of type-respecting relations $\mathcal{R}$ in D26.1.2.

Definition 26.5.1 (Open Extension)**.** *Consider $R \in \mathcal{R}_{cl}$ a relation on closed terms-in-context. We can extend it to an **open extension** $R^\circ \in \mathcal{R}$ on possibly open terms-in-context as follows:*

$$\Gamma \vdash s \; R^\circ \; t : \sigma$$

precisely when for all Γ-closures $\mathcal{V}$,

$$\vdash s\mathcal{V} \; R \; t\mathcal{V} : \sigma.$$

Lemma 26.5.2. 1. *R° inherits transitivity and reflexivity from R if R possesses these properties.*

2. *R and R° coincide on closed terms.*

3. *If $R \subseteq S$ then $R^o \subseteq S^o$.*

Proof. By unpacking D26.5.1 above. □

Definition 26.5.3. *Let $\lhd^\circ$ be the open extension of $\lhd$ (D26.5.1).*

Remark 26.5.4. By L26.5.2 $\lhd^\circ$ inherits reflexivity and transitivity from $\lhd$. ◇

Lemma 26.5.5 ($\lhd^\circ$ implies typing)**.** *If $\Gamma \vdash t \lhd^\circ t' : \tau$ then $\Gamma \vdash t, t' : \tau$.*

Proof. D26.5.3 is a type-respecting relation (D26.1.2) so this condition is part of the judgement. □

Lemma 26.5.6 ($\lhd$=$\lhd^\circ$ on closed terms)**.** *For $t, t' \in \boldsymbol{CTerms}$ and τ a type, if $\vdash t, t' : \tau$ then*

$$\vdash t \lhd^\circ t' : \tau \iff t \lhd t'.$$

PROOF. Direct from L26.5.2. □

We shall use this later.

Lemma 26.5.7 ($\lhd^\circ$ adequate)**.** *$\lhd^\circ$ is adequate (D26.1.6).*

PROOF. We continue the notation of D26.1.6. From the construction of $\lhd^\circ$, if $\Gamma \vdash t \lhd^\circ t' : \tau$ then $t\mathcal{V} \lhd t\mathcal{V}'$. By the clauses of T26.4.7 dealing with `Nat` and `Bool` we know $t\mathcal{V} \Downarrow V$ implies $t'\mathcal{V} \Downarrow V$. □

Definition 26.5.8 (Extend $\leq_{\mathbf{kl}}$ to open terms)**.** *Let $\leq^\circ_{\boldsymbol{kl}}$ be the open extension of $\leq_{\boldsymbol{kl}}$. It too inherits transitivity and reflexivity from $\leq_{\boldsymbol{kl}}$ and coincides with it on closed terms.*

We can usefully lift more properties to the open extensions.

Lemma 26.5.9 ($\leq^\circ_{\mathbf{kl}} \subseteq \lhd^\circ$)**.** *$\leq^\circ_{\boldsymbol{kl}}$ is a subrelation of $\lhd^\circ$. That is, $\leq^\circ_{\boldsymbol{kl}} \subseteq \lhd^\circ$.*

PROOF. Inherited from L26.4.8 ($\leq_{\mathbf{kl}} \subseteq \lhd^\circ$). □

This result underlies L27.13, where we use it to switch between $\leq^\circ_{\mathbf{kl}}$-equivalent terms on the left of $\lhd^\circ$. Cf. R26.4.10.

Corollary 26.5.10. *If $\Gamma \vdash s =^\circ_{\boldsymbol{kl}} s' : \tau$ then*

$$\Gamma \vdash s \lhd^\circ t : \tau \iff \Gamma \vdash s' \lhd^\circ t : \tau,$$

and similarly if $t =_{\boldsymbol{kl}} t'$.

PROOF. L26.5.9 ($\leq^\circ_{\mathbf{kl}} \subseteq \lhd^\circ$), D26.5.8 (def. $\leq^\circ_{\mathbf{kl}}$) and L26.5.4 ($\lhd^\circ$ trans. and refl.). □

Lemma 26.5.11 ($\lhd^\circ$ Substitution Properties)**.** *For U and U'* values*, if*

$$\Gamma \vdash U \lhd^\circ U' : \tau, \Gamma \vdash U, U' \# \overline{xc} \quad \textit{and} \quad \Gamma, u : \tau \# \overline{xc} \vdash t \lhd^\circ t' : \sigma$$

then

$$\Gamma \vdash t[U/u] \lhd^\circ t'[U'/u] : \sigma.$$

(U and U' must be values to guarantee that $t[U/u]$ and $t'[U'/u]$ are terms in our reduced syntax, see T22.1.8)

PROOF. From D26.5.3 (def. $\lhd^\circ$). □

26.6. Pause for Breath. Between us here and §28, that is §26.7 and §27, is a considerable mass of inductive proofs. Perhaps we should consider where we are and where we are going.

We know very little about $\lhd_{\mathbf{ctx}}$ beyond its definition, that it is the largest adequate congruence (D26.2.2). We certainly have little idea about its properties on closed values of type Λ_α, which is what the Sanity Clause (T21.9) is all about. Using our current terminology, the theorem states that on closed values of type Λ_α, $\equiv_{\mathbf{ctx}}$ and $\equiv^{se}$-equivalence coincide (R26.3.3).

Now from Fig.39_{198} and T26.4.7 we know a lot about $\lhd$ on Λ_α. The relevant line of the figure is

$$\vdash U, t : \Lambda_\alpha \implies U \lhd t \iff \exists V \equiv^{se} U.\ t \Downarrow V.$$

In the case that t is a closed value V it evaluates only to itself by L25.4, so this equation becomes precisely the Sanity Clause:

$$\vdash U, V : \Lambda_\alpha \implies U \lhd V \iff V \equiv^{se} U.$$

Seeing as we want to prove the Sanity Clause, wouldn't it be convenient if $\lhd{=}\lhd_{\mathbf{ctx}}$ on closed values of type Λ_α? Well, we could prove this if we knew $\lhd_{\mathbf{ctx}}{=}\lhd^\circ$ (D26.5.3) since by L26.5.6, $\lhd$ and $\lhd^\circ$ coincide on closed terms.

$\lhd_{\mathbf{ctx}}$ (D26.2.2) is coinductively defined so we show $\lhd^\circ$ (D26.5.3) is closed under its rules and we have $\lhd^\circ{\subseteq}\lhd_{\mathbf{ctx}}$. $\lhd_{\mathbf{ctx}}$ is a congruence and adequate. We already proved $\lhd^\circ$ is adequate in L26.5.7, but we do not know it is a congruence. This is rather hard (it all comes together in the end of the proof of L28.16).

For the converse $\lhd_{\mathbf{ctx}}{\subseteq}\lhd^\circ$, we shall show in L28.19 that $\lhd_{\mathbf{ctx}}$ is its own open extension. Meanwhile $\lhd^\circ$ is the open extension of the coinductively defined $\lhd$ (D26.4.1). Set inclusion is preserved by taking open extensions (L26.5.2) so it suffices to show that $\lhd_{\mathbf{ctx}}$ restricted to closed terms is contained in $\lhd$ (L28.20), which means proving it is closed under the monotone operator Φ (Fig.38_{196}) used to coinductively define $\lhd$. We do this after a few preparatory lemmas in C28.24.

With reference to R26.2.3, this proof method is only possible because we constructed $\lhd_{\mathbf{ctx}}$ coinductively.

Remark 26.6.1. To prove $\lhd^\circ$ a congruence we use "***Howe's Method***", originally presented in [**36**] and [**35**], and it takes up the space between here and §28. I have not used this original presentation but **directly modelled the proof**

which follows on an application of Howe's method by Pitts in [64].[89]
◇

In theory the method is simple. We define another relation $\lhd^*$ using $\lhd^\circ$, show one-by-one that it inherits all the good properties of $\lhd^\circ$, but also that it is a congruence. We then use these good properties to show they are equal. This is important: most of the results to follow are just results about $\lhd^\circ$, repeated for $\lhd^*$ and the reader is invited to use this correspondence to understand the structure of the proof. The only results not lifted from the theory of $\lhd^\circ$ are T26.7.4 ($\lhd^* \circ \lhd^\circ \subseteq \lhd^*$) and of course L27.10 ($\lhd^*$ congruence).

It is quite easy really, but checking all the cases does take up a lot of space. I advise the casual reader not to take anything between now and §29 too seriously, since I would much prefer them awake.

26.7. The Relation $\lhd^*$.

Definition 26.7.1. *We define an auxiliary relation $\lhd^*$ inductively on possibly open terms-in-context as shown in Fig.40*$_{204}$ *and Fig.41*$_{205}$*.*

We need the following lemma in the case $[\mathtt{Atm}]\tau$ of T27.11.

Lemma 26.7.2 ($\lhd^*$ equivariant)**.** *If*

$$\Gamma \vdash s \lhd^* t : \tau$$

then

$$(a\ b)\cdot\Gamma \vdash (a\ b)\cdot s \lhd^* (a\ b)\cdot t : \tau.$$

Proof. Just like the proof of L26.4.4 only a little more complex. We know $(a\ b)\cdot\tau = \tau$ by examining the grammar that generated it in D22.1.1 and observing that atoms $a \in \mathbb{A}$ do not feature in it. □

Remark 26.7.3 (Overview)**.** We now embark on an extended development leading up to T27.15 ($\lhd^\circ = \lhd^*$). The important and difficult technical results are T27.11 ($\lhd^*$ on values) and then T27.14 ($\lhd^*$ evaluation box). Relatively simple but important technical results are T26.7.4 ($\lhd^* \circ \lhd^\circ \subseteq \lhd^*$) immediately below and T27.9 ($\lhd^*$ substitution properties). ◇

Theorem 26.7.4 ($\lhd^* \circ \lhd^\circ \subseteq \lhd^*$)**.** *For all t_1, t_2, t_3, and Γ, τ such that*

$$\Gamma \vdash t_i : \tau \qquad i = 1, 2, 3$$

[89] I have not only lifted Dr Pitts' method, but could never have controlled it without his guidance (it is *not* easy to understand, at least the first time). I take the opportunity to thank him.

(181) $\Gamma \vdash xc \lhd^* t : \sigma$ (**if** $\Gamma \vdash xc \lhd^\circ t : \sigma$)

(182) $\Gamma \vdash \mathtt{True} \lhd^* t : \mathtt{Bool}$ (**if** $\Gamma \vdash \mathtt{True} \lhd^\circ t : \mathtt{Bool}$)

(183) $\Gamma \vdash \mathtt{False} \lhd^* t : \mathtt{Bool}$ (**if** $\Gamma \vdash \mathtt{False} \lhd^\circ t : \mathtt{Bool}$)

(184) $$\frac{\Gamma \vdash V \lhd^* V' : \mathtt{Bool} \quad \Gamma \vdash t_1 \lhd^* t_1' : \tau \quad \Gamma \vdash t_2 \lhd^* t_2' : \tau}{\Gamma \vdash \mathtt{if}\ V\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 \lhd^* t : \tau}$$ (**if** $\Gamma \vdash \mathtt{if}\ V'\ \mathtt{then}\ t_1'\ \mathtt{else}\ t_2' \lhd^\circ t : \tau$)

(185) $\Gamma \vdash 0 \lhd^* t : \mathtt{Nat}$ (**if** $\Gamma \vdash 0 \lhd^\circ t : \mathtt{Nat}$)

(186) $$\frac{\Gamma \vdash V \lhd^* V' : \mathtt{Nat}}{\Gamma \vdash \mathtt{Succ}(V) \lhd^* t : \mathtt{Nat}}$$ (**if** $\Gamma \vdash \mathtt{Succ}(V') \lhd^\circ t : \mathtt{Nat}$)

(187) $$\frac{\Gamma \vdash V \lhd^* V' : \mathtt{Nat} \quad \Gamma \vdash t_0 \lhd^* t_0' : \tau \quad \Gamma, x : \mathtt{Nat} \# \overline{xc} \vdash t_f \lhd^* t_f' : \tau}{\Gamma \vdash \mathtt{Case}\ V\ \mathtt{of}\ \{0 \Rightarrow t_0, \mathtt{Succ}(x) \Rightarrow t_f\} \lhd^* t : \tau}$$

(**if** $\Gamma \vdash \mathtt{Case}\ V'\ \mathtt{of}\ \{0 \Rightarrow t_0', \mathtt{Succ}(x) \Rightarrow t_f'\} \lhd^\circ t : \tau$)

(188) $$\frac{\Gamma \vdash V_1 \lhd^* V_1' : \tau_1 \overline{xc} \quad \Gamma \vdash V_2 \lhd^* V_2' : \tau_2}{\Gamma \vdash (V_1, V_2) \lhd^* t : \tau_1 \times \tau_2}$$ (**if** $\Gamma \vdash (V_1', V_2') \lhd^\circ t : \tau_1 \times \tau_2$)

(189) $$\frac{\Gamma \vdash V \lhd^* V' : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{Fst}(V) \lhd^* t : \tau_1}$$ (**if** $\Gamma \vdash \mathtt{Fst}(V') \lhd^\circ t : \tau_1$)

(190) $$\frac{\Gamma \vdash V \lhd^* V' : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{Snd}(V) \lhd^* t : \tau_2}$$ (**if** $\Gamma \vdash \mathtt{Snd}(V') \lhd^\circ t : \tau_2$)

(191) $$\frac{\Gamma \vdash V \lhd^* V' : \mathtt{Atm}}{\Gamma \vdash \mathtt{Var}(V) \lhd^* t : \Lambda_\alpha}$$ (**if** $\Gamma \vdash \mathtt{Var}(V') \lhd^\circ t : \Lambda_\alpha$)

(192) $$\frac{\Gamma \vdash V \lhd^* V' : \Lambda_\alpha \times \Lambda_\alpha}{\Gamma \vdash \mathtt{App}(V) \lhd^* t : \Lambda_\alpha}$$ (**if** $\Gamma \vdash \mathtt{App}(V') \lhd^\circ t : \Lambda_\alpha$)

(193) $$\frac{\Gamma \vdash V \lhd^* V' : [\mathtt{Atm}]\Lambda_\alpha}{\Gamma \vdash \mathtt{Lam}(V) \lhd^* t : \Lambda_\alpha}$$ (**if** $\Gamma \vdash \mathtt{Lam}(V') \lhd^\circ t : \Lambda_\alpha$)

(194) $$\frac{\Gamma \vdash V \lhd^* V' : \tau \quad \Gamma \vdash V, V' \# \overline{xc} \quad \Gamma, a : \mathtt{Atm} \# \overline{xc} \vdash t_V \lhd^* t_V' : \tau \quad \Gamma, x : \Lambda_\alpha \times \Lambda_\alpha \# \overline{xc} \vdash t_A \lhd^* t_A' : \tau \quad \Gamma, x : [\mathtt{Atm}]\Lambda_\alpha \vdash t_L \lhd^* t_L' : \tau}{\Gamma \vdash \mathtt{Case}\ V\ \mathtt{of}\ \{\mathtt{Var}(x) \Rightarrow V_V, \mathtt{App}(x) \Rightarrow V_A, \mathtt{Lam}(x) \Rightarrow V_L\} \lhd^* t : \tau}$$

(**if** $\Gamma \vdash \mathtt{Case}\ V\ \mathtt{of}\ \{\mathtt{Var}(x) \Rightarrow V_V', \mathtt{App}(x) \Rightarrow V_A', \mathtt{Lam}(x) \Rightarrow V_L'\} \lhd^\circ t : \tau$)

FIGURE 40. D26.7.1 - $\lhd^*$ Defined 1

(195) $\Gamma \vdash \mathtt{Nil}_\tau \lhd^* t : \tau$ (**if** $\Gamma \vdash \mathtt{Nil}_{\lhd^\circ} t : \tau$)

(196) $$\frac{\Gamma \vdash V_h \lhd^* V_h' : \tau \quad \Gamma \vdash V_t \lhd^* V_t' : (\tau)\mathtt{List}}{\Gamma \vdash V_h :: V_t \lhd^* t' : (\tau)\mathtt{List}}$$ (**if** $\Gamma \vdash V_h' :: V_t' \lhd^\circ t' : (\tau)\mathtt{List}$)

(197) $$\frac{\Gamma \vdash V \lhd^* V' : \tau \quad \Gamma \vdash xc : \mathtt{Atm}}{\Gamma \vdash xc.V \lhd^* t : [\mathtt{Atm}]\tau}$$ (**if** $\Gamma \vdash xc.V' \lhd^\circ t : [\mathtt{Atm}]\tau$)

(198) $$\frac{\Gamma \vdash V \lhd^* V' : [\mathtt{Atm}]\tau \quad \Gamma \vdash V, V' \# xc}{\Gamma \vdash V @ xc \lhd^* t : \tau}$$ (**if** $\Gamma \vdash V' @ xc \lhd^\circ t : \tau$)

(199) $$\frac{\Gamma \# x : \mathtt{Atm} \vdash s \lhd^* s' : \tau \quad \Gamma \# x : \mathtt{Atm} \vdash s, s' \# x}{\Gamma \vdash \mathtt{fresh}\, x\, \mathbf{in}\ s \lhd^* t' : \tau}$$ (**if** $\Gamma \vdash \mathtt{fresh}\, x\, \mathbf{in}\ s' \lhd^\circ t' : \tau$)

(200) $$\frac{\begin{array}{c}\Gamma \vdash xc : \mathtt{Atm} \quad \Gamma \vdash yc : \mathtt{Atm} \\ \Gamma \vdash V_3 \lhd^* V_3' : \tau \quad \Gamma, xc \# yc \vdash V_4 \lhd^* V_4' : \tau\end{array}}{\Gamma \vdash \mathtt{if}\, xc = yc\, \mathtt{then}\ V_3\ \mathtt{else}\ V_4 \lhd^* t : \tau}$$

(**if** $\Gamma \vdash \mathtt{if}\, xc = yc\, \mathtt{then}\ V_3'\ \mathtt{else}\ V_4' \lhd^\circ t : \tau$)

(201) $$\frac{\Gamma \vdash xc : \mathtt{Atm} \quad \Gamma \vdash V_3 \lhd^* V_3' : \tau}{\Gamma \vdash \mathtt{if}\, xc = xc\, \mathtt{then}\ V_3\ \mathtt{else}\ V_4 \lhd^* t : \tau}$$

(**if** $\Gamma \vdash \mathtt{if}\, xc = xc\, \mathtt{then}\ V_3'\ \mathtt{else}\ V_4' \lhd^\circ t : \tau$)

(202) $$\frac{\Gamma, f : \tau' \to \tau, x : \tau' \vdash s \lhd^* s' : \tau}{\Gamma \vdash \mathtt{fix}\, f(x : \tau')\, \mathtt{in}\ s \lhd^* t : \tau' \to \tau}$$

(**if** $\Gamma \vdash \mathtt{fix}\, f(x : \tau')\, \mathtt{in}\ s' \lhd^\circ t : \tau' \to \tau$)

(203) $$\frac{\Gamma \vdash V_2 \lhd^* V_2' : \tau' \quad \Gamma \vdash V_1 \lhd^* V_1' : \tau' \to \tau}{\Gamma \vdash V_1\ V_2 \lhd^* t : \tau}$$ (**if** $\Gamma \vdash V_1'\ V_2' \lhd^\circ t : \tau$)

(204) $$\frac{\Gamma \vdash s_1 \lhd^* s_1' : \tau \quad \Gamma \vdash s_1, s_1' \# \overline{xc} \quad \Gamma, x : \tau \# \overline{xc} \vdash s_2 \lhd^* s_2' : \sigma}{\Gamma \vdash \mathtt{let}\ x = s_1\ \mathtt{in}\ s_2 \lhd^* t : \sigma}$$

(**if** $\Gamma \vdash \mathtt{let}\ x = s_1'\ \mathtt{in}\ s_2' \lhd^\circ t : \sigma$)

FIGURE 41. D26.7.1 - $\lhd^*$ Defined 2

it is the case that

$$(\Gamma \vdash t_1 \lhd^* t_2 : \tau) \ \wedge\ (\Gamma \vdash t_2 \lhd^\circ t_3 : \tau) \implies \Gamma \vdash t_1 \lhd^* t_3 : \tau.$$

PROOF. By induction on $\lhd^*$, using transitivity of $\lhd^\circ$ (R26.5.4). □

Lemma 26.7.5 ($\lhd^*$ Reflexive)**.** *For all t and Γ, τ, if $\Gamma \vdash t : \tau$ then*

$$\Gamma \vdash t \lhd^* t : \tau.$$

PROOF. By induction on the derivation of $\Gamma \vdash t : \tau$, using reflexivity of $\lhd^\circ$ (R26.5.4). □

Lemma 26.7.6 ($\lhd^\circ{\subseteq}\lhd^*$)**.** *If* $\Gamma \vdash t_1 \lhd^\circ t_2 : \tau$ *then* $\Gamma \vdash t_1 \lhd^* t_2 : \tau$.

PROOF. By combining T26.7.4 with L26.7.5. □

Corollary 26.7.7 ($\leq_{\mathbf{kl}}{\subseteq}\lhd^*$)**.** *L26.7.6, L26.5.6, and L26.4.8 imply*

$$\Gamma \vdash t_1 \leq_{\boldsymbol{kl}} t_2 : \tau \implies \Gamma \vdash t_1 \lhd^* t_2 : \tau.$$

Lemma 26.7.8. $\lhd^*$ *is a type-respecting relation (D26.1.2): for* Γ, τ *and terms* t *and* t'*, if*

$$\Gamma \vdash t \lhd^* t' : \tau$$

then

$$\Gamma \vdash t : \tau \quad \textit{and} \quad \Gamma \vdash t' : \tau.$$

PROOF. From the nature of the inductive rules defining $\lhd^*$, using the corresponding result, evident from D26.5.3, for $\lhd^\circ$. □

27. Proof of $\lhd^\circ{=}\lhd^*$ and hence $\lhd^\circ$ congruence

Theorem 27.9 ($\lhd^*$ Substitution Properties)**.** *For* U *and* U' values*, if*

$$\Gamma \vdash U \lhd^* U' : \tau, \quad \Gamma \vdash U, U' \# \overline{xc} \quad \textit{and} \quad \Gamma, u : \tau \# \overline{xc} \vdash t \lhd^* t' : \sigma$$

then

$$\Gamma \vdash t[U/u] \lhd^* t'[U'/u] : \sigma.$$

(U *and* U' *must be values to guarantee that* $t[U/u]$ *and* $t'[U'/u]$ *are terms in our reduced syntax, see T22.1.8.)*

PROOF. By induction on the derivation of $\Delta \vdash t \lhd^* t' : \tau$ (due to the syntax-directed nature of the rules this is similar to induction on the syntax of t) with fixed U, U' and inductive hypothesis

$$\forall \Gamma, u, \overline{xc}.\ \Big(\Delta \vdash t \lhd^* t' : \sigma \ \wedge\ \Delta = \Gamma, u : \tau \# \overline{xc} \implies \Gamma \vdash U \lhd^* U' : \tau \ \wedge\ \Gamma \vdash U, U' \# \overline{xc} \implies \Gamma \vdash t[U/u] \lhd^* t'[U'/u] : \sigma\Big).$$

In the proof below I shall elide Δ and the universal quantifier, assuming $\Delta = \Gamma, u : \tau \# \overline{xc}$ from the start, effectively working with an ***informal*** hypothesis of the form

$$\forall \Gamma, u, \overline{xc}.\ (\Gamma \vdash U \lhd^* U' : \tau \ \wedge\ \Gamma \vdash U, U' \# \overline{xc}) \Rightarrow \Gamma \vdash t[U/u] \lhd^* t'[U'/u] : \sigma.$$

We consider only a smattering of cases.

1• *Suppose* $t = xc$. Then the result follows directly from a corresponding result for $\lhd^\circ$, L26.5.11.

2• The cases $t = \mathtt{True}, \mathtt{False}, 0$ are trivial.

3• *Suppose* $t = \mathtt{Case}\ V\ \mathtt{of}\ \{0 \Rightarrow t_0, \mathtt{Succ}(x) \Rightarrow t_f\}$. We suppose that

$$\Gamma \vdash U \lhd^* U' : \tau, \quad \Gamma \vdash U, U' \# \overline{xc} \quad \text{and} \quad u : \tau \# \overline{xc}, \Gamma \vdash t \lhd^* t' : \sigma.$$

Using $(187)_{204}$ we know there are $t_0', t_f', \overline{yc}$ and V' such that

$$u : \tau \# \overline{xc}, \Gamma \vdash t_0 \lhd^* t_0' : \sigma,$$
$$u : \tau \# \overline{xc}, \Gamma, x : \mathtt{Nat} \# \overline{yc} \vdash t_f \lhd^* t_f' : \sigma \quad \text{and}$$
$$u : \tau \# \overline{xc}, \Gamma \vdash V \lhd^* V' : \sigma,$$

and that

(205) $$u : \tau \# \overline{xc}, \Gamma \vdash \mathtt{Case}\ V_0'\ \mathtt{of}\ \{0 \Rightarrow t_0', \mathtt{Succ}(x) \Rightarrow t_f'\} \lhd^\circ t' : \sigma.$$

By inductive hypothesis we conclude that

$$\Gamma \vdash V[U/u] \lhd^* V'[U'/u] : \sigma,$$
$$\Gamma \vdash t_0[U/u] \lhd^* t_0'[U'/u] : \sigma \quad \text{and}$$
$$\Gamma, x : \mathtt{Nat} \# \overline{yc} \vdash t_f[U/u] \lhd^* t_f'[U'/u] : \sigma$$

and from $(205)_{207}$ we can use the definition of $\lhd^\circ$ to deduce that

$$\Gamma \vdash (\mathtt{Case}\ V'\ \mathtt{of}\ \{0 \Rightarrow t_0', \mathtt{Succ}(x) \Rightarrow t_f'\})[U'/u] \lhd^\circ t'[U'/u] : \sigma.$$

The result follows by the definition of $\lhd^*$.

4• *Suppose* $t = \mathtt{fresh}\, x\ \textbf{\textit{in}}\ s$. Suppose that

$$\Gamma \vdash U \lhd^* U' : \tau, \quad \Gamma \vdash U, U' \# \overline{xc} \quad \text{and} \quad u : \tau \# \overline{xc}, \Gamma \vdash t \lhd^* t' : \sigma.$$

We resolve with $(199)_{205}$ and deduce the existence of an s' such that

$$\Gamma \# x : \mathtt{Atm} \vdash s \lhd^* s' : \sigma$$
$$\Gamma \# x : \mathtt{Atm} \vdash s, s' \# x$$
$$u : \tau \# \overline{xc}, \Gamma \vdash \mathtt{fresh}\, x\ \mathbf{in}\ s' \lhd^\circ t' : \sigma.$$

By induction hypothesis we know that

$$\Gamma \# x : \mathtt{Atm} \vdash s[U/u] \lhd^* s'[U'/u] : \sigma \quad \text{and} \quad \Gamma \# x : \mathtt{Atm} \vdash s[U/u], s'[U'/u] \# x$$

and by the definition of $\lhd^\circ$ we know that

$$\Gamma \vdash \mathtt{fresh}\, x\ \mathbf{in}\ s'[U'/u] \lhd^\circ t'[U'/u] : \sigma.$$

The result follows by the construction of $\lhd^*$ (D26.7.1).

5• *Suppose* $t = \mathtt{fix}\, f(x:\tau')\,\mathtt{in}\ s$. Suppose that

$$\Gamma \vdash U \lhd^* U' : \tau,\ \Gamma \vdash U, U' \# \overline{xc} \quad \text{and} \quad u : \tau \# \overline{xc}, \Gamma \vdash t \lhd^* t' : \tau' \to \tau.$$

We resolve with $(202)_{205}$ and see that

$$u : \tau \# \overline{xc}, \Gamma, f : \tau' \to \tau, x : \tau' \vdash s \lhd^* s' : \tau$$

and the side-condition of $(202)_{205}$ holds. We can now apply the induction hypothesis to conclude

$$\Gamma, f : \tau' \to \tau, x : \tau' \vdash s[U/u] \lhd^* s'[U'/u] : \tau,$$

and since the appropriate side-condition still holds for these new terms, we can deduce the required result.

□

Unlike $\lhd^\circ$, the relation $\lhd^*$ is a congruence:

Lemma 27.10 ($\lhd^*$ congruence)**.** *$\lhd^*$ is a congruence (N26.2.1); it is closed under the rules of Fig.37$_{193}$.*

Proof. By considering the rules of Fig.37_{193} one-by-one.

We give just one example, that of $(176)_{193}$. Suppose

$$\Gamma \# x : \mathtt{Atm} \vdash t \lhd^* t' : \tau \quad \text{and} \quad \Gamma \# x : \mathtt{Atm} \vdash t, t' \# x.$$

We also know from R26.5.4 ($\lhd^\circ$ reflexive) that

$$\Gamma \# x : \mathtt{Atm} \vdash t' \lhd^\circ t' : \tau.$$

We now observe that we can use $(199)_{205}$ to deduce that

$$\Gamma \vdash \mathtt{fresh}\, x\, \mathbf{in}\ t \lhd^* \mathtt{fresh}\, x\, \mathbf{in}\ t' : \tau.$$

The other cases are no harder. □

Theorem 27.11 ($\lhd^*$ on Closed Values)**.** *$\lhd^*$ restricted to closed terms satisfies all the lemmas of $\lhd$ given in Fig.39$_{198}$. In full this means:*

1. *For B one of* `True` *and* `False` *(so a value), if $\vdash B \lhd^* t' : \mathtt{Bool}$ then $t' \Downarrow B$.*
2. *For N a value, if $\vdash N \lhd^* t' : \mathtt{Nat}$ then $t' \Downarrow N$.*
3. *If $\vdash a.W \lhd^* t' : [\mathtt{Atm}]\tau$ then there are a', W' such that $t' \Downarrow a'.W'$ and for c new $\vdash (c\ a)\cdot W \lhd^* (c\ a')\cdot W' : \tau$.*
4. *For L a value, if $\vdash L \lhd^* t' : \Lambda_\alpha$ then there is a value $L' \equiv^{se} L$ such that $t' \Downarrow L'$.*

5. *If* $\vdash \mathtt{Nil}_\tau \lhd^* t' : (\tau)\mathtt{List}$ *then* $t' \Downarrow \mathtt{Nil}_\tau$.

   For $V_h{::}V_t$ *a value, if* $\vdash V_h{::}V_t \lhd^* t' : (\tau)\mathtt{List}$ *then there exist values* V'_h, V'_t *such that* $\vdash V_h \lhd^* V'_h : \tau$, $\vdash V_t \lhd^* V'_t : (\tau)\mathtt{List}$, *and* $t' \Downarrow V'_h{::}V'_t$.
6. *For* $P = (P_1, P_2)$ *a value, if* $\vdash P \lhd^* t' : \tau_1 \times \tau_2$ *then there exist values* P'_1, P'_2 *such that* $t' \Downarrow (P'_1, P'_2)$ *and* $\vdash P_i \lhd^* P'_i : \tau_i$ *for* $i = 1, 2$.
7. *For* $F = \mathtt{fix}\, f(x : \tau')\, \mathtt{in}\ s$, *if* $\vdash F \lhd^* t' : \tau' \to \tau$ *then there exists* $F' = \mathtt{fix}\, f(x : \tau')\, \mathtt{in}\ s'$ *such that* $t' \Downarrow F'$ *and for all* $U \in \boldsymbol{CVal}_{\tau'}$,

$$\vdash F\ U \lhd^* F'\ U : \tau.$$

PROOF. In view of L26.5.6 ($\lhd$ and $\lhd^\circ$ coincide on closed terms) we shall use $\lhd^\circ$ and $\lhd$ synonymously on closed terms.

1• *Type* `Bool`. Suppose $\vdash B = \mathtt{True} \lhd^* t' : \mathtt{Bool}$. We resolve with $(182)_{204}$ and deduce

$$\vdash \mathtt{True} \lhd^\circ t' : \mathtt{Bool}.$$

Since t' and `True` are closed terms it follows by L26.5.6 ($\lhd$ and $\lhd^\circ$ coincide on closed terms) that

$$\vdash \mathtt{True} \lhd t' : \mathtt{Bool}.$$

Finally by T26.4.7 we know that $t' \Downarrow \mathtt{True}$. The case $B = \mathtt{False}$ is similar.

2• *Type* `Nat`. We work by induction on the derivation of $\lhd^*$ using the hypothesis

$$\vdash N \lhd^* t : \mathtt{Nat} \implies t \Downarrow N.$$

We do only one case, that of $N = \mathtt{Succ}(M)$. Suppose

$$\vdash N \lhd^* t : \mathtt{Nat}.$$

It must be the case that for some M',

$$\vdash M \lhd^* M' : \mathtt{Nat} \quad \text{and} \quad \vdash \mathtt{Succ}(M') \lhd^\circ t : \mathtt{Nat}.$$

By induction hypothesis we know $M \Downarrow M'$ and, these both being values, $M = M'$. From T26.4.7 it follows that

$$t \Downarrow \mathtt{Succ}(M') = \mathtt{Succ}(M) = N$$

as required.

3• *Type* $[\mathtt{Atm}]\tau$. Suppose we have $\vdash a.W \lhd^* t' : [\mathtt{Atm}]\tau$. We follow back the deduction of $\lhd^*$ and deduce the existence of a W'' such that

$$\vdash W \lhd^* W'' : \tau \quad \text{and} \quad \vdash a.W'' \lhd^\circ t' : [\mathtt{Atm}]\tau.$$

We apply T26.4.7 ($\lhd$ on values) to this second fact and deduce the existence of a', W' and a new c such that

$$t' \Downarrow a'.W' \quad \text{and} \quad \vdash (c\ a)\cdot W'' \lhd^\circ (c\ a')\cdot W' : \tau.$$

We also know from L26.7.2 ($\lhd^*$ equivariant) and $\vdash W \lhd^* W'' : \tau$ deduced above that

$$\vdash (c\ a)\cdot W \lhd^* (c\ a)\cdot W'' : \tau.$$

We can now apply T26.7.4 ($\lhd^* \circ \lhd^\circ \subseteq \lhd^*$) to obtain the desired result.

4• *Type* Λ_α. By induction on $\lhd^*$ using the hypothesis

$$\vdash L \lhd^* t' : \Lambda_\alpha \implies \exists L' \equiv^{se} L.\ t' \Downarrow L'.$$

We do only one case, the more subtle one. Suppose

$$L = \mathtt{Lam}(M) \quad \text{and} \quad \Gamma \vdash L \lhd^* t' : \Lambda_\alpha.$$

Then by L26.7.8 and the structure of the typing rules we know

$$\vdash L : \Lambda_\alpha \quad \text{and} \quad M = a.U.$$

We resolve with $(193)_{204}$ and deduce that for some $M'' = a.U''$,

$$\vdash M \lhd^* M'' : [\mathtt{Atm}]\Lambda_\alpha \quad \text{and} \quad \vdash \mathtt{Lam}(M'') \lhd^\circ t' : \Lambda_\alpha.$$

We apply T26.4.7 to this second fact to conclude that for $L'' = \mathtt{Lam}(M'')$ and some L',

$$t \Downarrow L' \quad \text{and} \quad L' \equiv^{se} L''.$$

From the first fact on the other hand we can follow back $\lhd^*$ and see there is some $M''' = a.U'''$ such that

$$\vdash U \lhd^* U''' : \Lambda_\alpha \quad \text{and} \quad \vdash M''' \lhd^\circ M'' : [\mathtt{Atm}]\Lambda_\alpha$$

By induction hypothesis on this first fact $U \equiv^{se} U'''$ and therefore

$$L = \mathtt{Lam}(M) = \mathtt{Lam}(a.U) \equiv^{se} \mathtt{Lam}(a.U''') = \mathtt{Lam}(M''')$$

($\equiv^{se}$ is a congruence by construction). By the second fact and T26.4.7 we know

$$M'' \Downarrow M''' \quad \text{so by L25.4} \quad M'' = M''',$$

so

$$L \equiv^{se} L''.$$

$\equiv^{se}$ is transitive, so from $L \equiv^{se} L'' \equiv^{se} L'$ we deduce $L \equiv^{se} L'$, as required.

5• *Type* $(\tau)\texttt{List}$. The first case is similar to the boolean one. Consider the second. Suppose

$$\vdash V_h :: V_t \lhd^* t' : (\tau)\texttt{List}.$$

We resolve with $(196)_{205}$ and deduce there exist V_h'' and V_t'' such that

$$\vdash V_h \lhd^* V_h'' : \tau, \quad \vdash V_t \lhd^* V_t'' : (\tau)\texttt{List}$$

and

$$\vdash V_h'' :: V_t'' \lhd^\circ t' : (\tau)\texttt{List}.$$

By T26.4.7 we know there exist V_h', V_t' such that $V_h' \lhd V_h''$, $V_t' \lhd V_t''$, and $t' \Downarrow V_h' :: V_t'$. We now apply T26.7.4 to obtain the result.

6• *Type* $\tau_1 \times \tau_2$. As for the second case of list types above.

7• *Type* $\tau' \to \tau$. Suppose $\vdash F = \texttt{fix}\, f(x:\tau')\,\texttt{in}\ s \lhd^* t' : \tau' \to \tau$. We resolve with $(202)_{205}$ and deduce that there exists an s'' such that

$$f : \tau' \to \tau, x : \tau' \vdash s \lhd^* s'' : \tau$$

and

$$\vdash \texttt{fix}\, f(x:\tau')\,\texttt{in}\ s'' \lhd^\circ t' : \tau' \to \tau.$$

We apply L27.10 and deduce

$$\vdash \texttt{fix}\, f(x:\tau')\,\texttt{in}\ s \lhd^* \texttt{fix}\, f(x:\tau')\,\texttt{in}\ s'' : \tau' \to \tau.$$

We also use T26.4.7 to deduce that s' exists such that

$$t' \Downarrow \texttt{fix}\, f(x:\tau')\,\texttt{in}\ s'$$

and for all $U \in \mathbf{CVal}_{\tau'}$

$$\vdash (\texttt{fix}\, f(x:\tau')\,\texttt{in}\ s'')\,U \lhd^\circ (\texttt{fix}\, f(x:\tau')\,\texttt{in}\ s')\,U : \tau.$$

It only remains now to use the case of function application in L27.10 to also deduce that

$$\vdash (\texttt{fix}\, f(x:\tau')\,\texttt{in}\ s)\,U \lhd^* (\texttt{fix}\, f(x:\tau')\,\texttt{in}\ s'')\,U : \tau$$

and apply T26.7.4 ($\lhd^* \circ \lhd^\circ \subseteq \lhd^*$) to complete the result.

□

Corollary 27.12. *For a type τ a and values V', V, if*

$$\vdash V \lhd^* V' : \tau$$

then V and V' have the same top-level term-former.

PROOF. Directly from T27.11. □

The following is a technical result which bridges a slight mismatch between the form of Φ at function types in Fig.38_{196} and the form that would be most useful in the case of function application in T27.14. It is *not* a case of L27.10.

Lemma 27.13 (Technical Lemma)**.** *If*

$$\Gamma \vdash R = (\mathtt{fix}\, f(x:\tau')\,\mathtt{in}\ s) \lhd^* R' = (\mathtt{fix}\, f(x:\tau')\,\mathtt{in}\ s') : \tau' \to \tau$$

and

$$\Gamma \vdash U \lhd^* U' : \tau'$$

then

$$\Gamma \vdash (\mathtt{let}\ x = U, f = R\ \mathtt{in}\ s) \lhd^* (\mathtt{let}\ x = U', f = R'\ \mathtt{in}\ s') : \tau.$$

PROOF. We resolve with $(202)_{205}$ and conclude that there exists an s'' such that

$$\Gamma, f:\tau' \to \tau, x:\tau' \vdash s \lhd^* s'' : \tau$$

and

$$\Gamma \vdash R'' \lhd^\circ R' : \tau' \to \tau.$$

where $R'' = \mathtt{fix}\, f(x:\tau')\,\mathtt{in}\ s''$.

By a slightly sophisticated use of L26.5.11 we deduce that

$$\Gamma \vdash R''\ U'' \lhd^\circ R'\ U' : \tau,$$

and hence, using C26.5.10, that

$$\Gamma \vdash (\mathtt{let}\ x = U'', f = R''\ \mathtt{in}\ s'') \lhd^\circ (\mathtt{let}\ x = U', f = R'\ \mathtt{in}\ s') : \tau.$$

By L27.10 we can deduce that

$$\Gamma \vdash (\mathtt{let}\ x = U, f = R\ \mathtt{in}\ s) \lhd^* (\mathtt{let}\ x = U'', f = (\mathtt{fix}\, f(x:\tau')\,\mathtt{in}\ s'')\ \mathtt{in}\ s'') : \tau.$$

We apply T26.7.4 ($\lhd^* \circ \lhd^\circ \subseteq \lhd^*$) to complete the result. □

Recall C26.4.11 and $(180)_{200}$. $\lhd^*$ restricted to closed values satisfies a similar property:

Theorem 27.14 ($\lhd^*$ Evaluation Box)**.** *For t a closed term and V a closed value, if $t \Downarrow V$ then for all t', if $\vdash t \lhd^* t' : \tau$ then there exist V' such that*

$$t' \Downarrow V' \quad \textit{and} \quad \vdash V \lhd^* V' : \tau.$$

PROOF. By induction on $t \Downarrow V$. In the light of L26.5.6 ($\lhd$ and $\lhd^\circ$ coincide on open terms) we shall use the judgements $\vdash t \lhd^\circ t' : \tau$ and $t \lhd t'$ synonymously.

1• *If* $t = V$ there is nothing to prove.

2• *Suppose* $t = \mathtt{if\,True\,then}\ t_1\ \mathtt{else}\ t_2$ *and suppose* $t \Downarrow V$. Therefore $t_1 \Downarrow V$. Suppose also $\vdash t \lhd^* t' : \tau$ and $\vdash V : \tau$. From $(184)_{204}$ we see there exist t_1', t_2', U' such that

$$(\vdash t_1 \lhd^* t_1' : \tau) \ \wedge\ (\vdash t_2 \lhd^* t_2' : \tau) \ \wedge\ (\vdash \mathtt{True} \lhd^* U' : \tau)$$

and

$$\vdash \mathtt{if}\ U'\ \mathtt{then}\ t_1'\ \mathtt{else}\ t_2' \lhd^\circ t' : \tau.$$

By C27.12 we conclude that $U' = \mathtt{True}$.

Now we know $t_1 \Downarrow V$ and $\vdash t_1 \lhd^* t_1' : \tau$, so we can apply the inductive hypothesis to deduce that there is some V_1' such that $t_1' \Downarrow V_1'$ and $\vdash V \lhd^* V_1' : \tau$.

Furthermore we know

$$\vdash t_1' \lhd^\circ \mathtt{if\,True\,then}\ t_1'\ \mathtt{else}\ t_2' \lhd^\circ t' : \tau \quad \text{and} \quad t_1' \Downarrow V_1'$$

so we know by C26.4.11 that $t' \Downarrow V'$ for a V' such that $\vdash V_1' \lhd^\circ V' : \tau$. We now use T26.7.4 to complete the result.

3• *The case* $t = \mathtt{if\,False\,then}\ t_1\ \mathtt{else}\ t_2$ is similar.

4• *We skip to the case* $t = \mathtt{Case}\ U\ \mathtt{of}\ \{0 \Rightarrow t_0, \mathtt{Succ}(x) \Rightarrow t_f\}$ where $U = \mathtt{Succ}(W)$. Suppose

$$t \Downarrow V \quad \text{and} \quad \vdash t \lhd^* t' : \tau.$$

So t_0', t_f' and U' must exist such that

$$\vdash \mathtt{Case}\ U'\ \mathtt{of}\ \left\{0 \Rightarrow t_0', \mathtt{Succ}(x) \Rightarrow V_f'\right\} \lhd^\circ t' : \tau$$

and

$$\begin{aligned} &(\vdash t_0 \lhd^* t_0' : \tau) && \wedge \\ &(x : \mathtt{Nat} \# \overline{xc} \vdash t_f \lhd^* t_f' : \tau) && \wedge \\ &(\vdash U \lhd^* U' : \mathtt{Nat}). \end{aligned}$$

We apply T27.11 to this third fact and deduce that $U' = U$.

Now we know $t \Downarrow V$, so we follow the evaluation relation and see that

$$\mathtt{let}\ x = W\ \mathtt{in}\ t_f \ \Downarrow\ V.$$

We use L27.10 to deduce

$$\vdash \mathtt{let}\ x = W\ \mathtt{in}\ t_f \lhd^* \mathtt{let}\ x = W\ \mathtt{in}\ t'_f : \tau,$$

(recall that $U = \mathtt{Succ}(W) = U'$) and using the induction hypothesis we know there exists V'_1 such that $\mathtt{let}\ x = W\ \mathtt{in}\ t'_f \Downarrow V'_1$, and $\vdash V \lhd^* V'_1 : \tau$.

Following the same pattern as the previous case, we know

$$\vdash \mathtt{let}\ x = W\ \mathtt{in}\ t'_f \leq_{\mathbf{kl}} \mathtt{Case}\ U\ \mathtt{of}\ \{0 \Rightarrow t'_0, \mathtt{Succ}(x) \Rightarrow V'_f\} \lhd^\circ t' : \tau$$

and using L26.4.8 ($\leq_{\mathbf{kl}} \subseteq \lhd$) and R26.5.4 ($\lhd^\circ$ transitive) we deduce

$$\vdash \mathtt{let}\ x = W\ \mathtt{in}\ t'_f \lhd^\circ t' : \tau.$$

From this, $\mathtt{let}\ x = W\ \mathtt{in}\ t'_f \Downarrow V'_1$, and C26.4.11 we conclude there exists a V' such that $t' \Downarrow V'$ and

$$\vdash V \lhd^* V'_1 \lhd^\circ V' : \tau.$$

We apply T26.7.4 to obtain the desired result.

5 • *Suppose*

$$t = \mathtt{Case}\ U\ \mathtt{of}\ \{\mathtt{Var}(a) \Rightarrow t_V, \mathtt{App}(x) \Rightarrow t_A, \mathtt{Lam}(a) \Rightarrow t_L\}$$

where $U = \mathtt{Lam}(a.\, W)$ and suppose that for some V and t',

$$t \Downarrow V \quad \text{and} \quad \vdash t \lhd^* t' : \tau.$$

We follow back $\lhd^*$ and deduce that there exist t''_V, t''_A, t''_L and U'' such that

$$\vdash \mathtt{Case}\,\mathtt{Lam}(U'')\ \mathtt{of}\ \{\mathtt{Var}(a) \Rightarrow t''_V, \mathtt{App}(x) \Rightarrow t''_A, \mathtt{Lam}(x) \Rightarrow t''_L\} \lhd^\circ t' : \tau$$

and

$$\begin{aligned}
&(a : \mathtt{Atm} \# \overline{xc} \vdash t_V \lhd^* t''_V : \tau) && \wedge \\
&(x : \Lambda_\alpha \times \Lambda_\alpha \# \overline{xc} \vdash t_A \lhd^* t''_A : \tau) && \wedge \\
&(x : [\mathtt{Atm}]\Lambda_\alpha \# \overline{xc} \vdash t_L \lhd^* t''_L : \tau) && \wedge \\
&(\vdash U \lhd^* U'' : \Lambda_\alpha) && \wedge \\
&(\vdash U, U'' \# \overline{xc}).
\end{aligned} \tag{206}$$

We can now use L27.10 to deduce that

$$\vdash \mathtt{let}\ x = U\ \mathtt{in}\ t_L \lhd^* \mathtt{let}\ x = U''\ \mathtt{in}\ t''_L : \tau.$$

We know from the evaluation relation that $\mathtt{let}\ x = U\ \mathtt{in}\ t_L \Downarrow V$, so we apply the inductive hypothesis to conclude that for some V'',

$$\mathtt{let}\ x = U''\ \mathtt{in}\ t''_L \Downarrow V'' \quad \text{and} \quad \vdash V \lhd^* V'' : \tau.$$

Now we know $\texttt{let}\ x = U''\ \texttt{in}\ t''_L \lhd t'$ from L26.4.8, and the fact that U and U'' have the same top-level term-former (C27.12). We can apply C26.4.11 to $\texttt{let}\ x = U''\ \texttt{in}\ t''_L \Downarrow V''$ and $\texttt{let}\ x = U''\ \texttt{in}\ t''_L \lhd t'$ to deduce the existence of V' such that

$$t' \Downarrow V' \quad \text{and} \quad \vdash V'' \lhd^\circ V' : \tau.$$

We now apply T26.7.4 to obtain the desired result.[90]

6• *Suppose* $t = R\ U$ where $R = (\texttt{fix}\, f(x:\tau')\ \texttt{in}\ s)$. We assume

$$t \Downarrow V \quad \text{and} \quad \vdash t \lhd^* t' : \tau.$$

From this second fact we deduce values $R'' = \texttt{fix}\, f(x:\tau')\ \texttt{in}\ s''$ and U'' exist such that

$$\vdash \texttt{fix}\, f(x:\tau')\ \texttt{in}\ s \lhd^* R'' : \tau' \to \tau \quad \text{and} \quad \vdash U \lhd^* U'' : \tau',$$

and

$$\vdash R''\ U'' \lhd^\circ t' : \tau.$$

We use L27.13 (the hard work of this case is hidden in that lemma) to deduce that

$$\vdash (\texttt{let}\ x = U, f = R\ \texttt{in}\ s) \lhd^* (\texttt{let}\ x = U'', f = R''\ \texttt{in}\ s'') : \tau.$$

From $t \Downarrow V$ we can deduce that

$$(\texttt{let}\ x = U, f = R\ \texttt{in}\ s) \Downarrow V.$$

By inductive hypothesis therefore we know that there is some V'' such that

$$(\texttt{let}\ x = U'', f = R''\ \texttt{in}\ s'') \Downarrow V'' \quad \text{and} \quad \vdash V \lhd^* V'' : \tau.$$

Since $\leq_{\mathbf{kl}}$ respects β-equivalence (L26.4.8) we know from $R''\,U'' \lhd t'$ that

$$(\texttt{let}\ x = U'', f = R''\ \texttt{in}\ s'') \lhd t'$$

and hence, by C26.4.11 we know there is a value V' such that

$$t' \Downarrow V' \quad \text{and} \quad V'' \lhd V'.$$

It remains only to apply T26.7.4 ($\lhd^* \circ \lhd^\circ \subseteq \lhd^\circ$) to complete the result.

[90] Are you reading this? Incredible! Here's a joke for your entertainment: "Which is worse, ignorance or apathy? Who knows? Who cares?"

7• *Suppose* $t = \texttt{fresh}\, x\, \textbf{\textit{in}}\, s$. Suppose also that

$$t \Downarrow V \quad \text{and} \quad \vdash t \lhd^* t' : \tau.$$

From this second fact we deduce the existence of an s'' such that

$$x : \texttt{Atm} \vdash s \lhd^* s'' : \tau, \quad x : \texttt{Atm} \vdash s, s' \# x$$

and

$$\vdash \texttt{fresh}\, x\, \textbf{in}\, s' \lhd^\circ t' : \tau.$$

Now we follow back the evaluation relation and deduce that for a new c,

$$s[c/x] \Downarrow V.$$

For this new c it is also the case, by T27.9, that

$$\vdash s[c/x] \lhd^* s''[c/x] : \tau \quad \text{and} \quad \vdash s[c/x], s''[c/x] \# c.$$

So by induction hypothesis there is some V'' such that

$$s''[c/x] \Downarrow V'' \quad \text{and} \quad \vdash V \lhd^* V'' : \tau.$$

Now we also know that $s''[c/x] \lhd \texttt{fresh}\, x\, \textbf{in}\, s'' \lhd t'$, so we can apply C26.4.11 as in the previous cases and deduce the existence of a V' such that $V'' \lhd V'$ and $t' \Downarrow V'$, which gives the result using T26.7.4 as in the previous cases.

8• *Suppose* $t = V_h :: V_t$ *and* $t \Downarrow V$. Since t is already a value, $t = V$. Suppose further

$$\vdash t \lhd^* t' : (\tau)\texttt{List}.$$

Then there exist V_h'', V_t'' such that

$$\vdash V_h \lhd^* V_h'' : \tau \quad \text{and} \quad \vdash V_f'' \lhd^* V_t'' : (\tau)\texttt{List}$$

and

$$\vdash V_h'' :: V_t'' \lhd^\circ t' : (\tau)\texttt{List}.$$

We use L27.10 to deduce $\vdash t = V_h :: V_t \lhd^* V_h'' :: V_t'' : (\tau)\texttt{List}$ and proceed using C26.4.11 and T26.7.4 as in the other cases.

9• *Suppose* $t = U @ c$ for U a value. Suppose further that

$$t \Downarrow V \quad \text{and} \quad \vdash t \lhd^* t' : \tau.$$

We follow back $\lhd^*$ and deduce the existence of a U'' such that

$$\vdash U \lhd^* U'' : [\texttt{Atm}]\tau \quad \text{and} \quad \vdash U'' @ c \lhd^\circ t' : \tau.$$

We use L26.7.8 and L26.5.5 and the typing rules for concretion to deduce that $\vdash U, U''\#c$.

We can now apply L27.10 to deduce $\vdash U@c \lhd^* U''@c : \tau$ and then proceed using C26.4.11 and T26.7.4 as in the other cases.

□

Theorem 27.15 ($\lhd^\circ{=}\lhd^*$)**.** *The relation $\lhd^\circ$ coincides with $\lhd^*$.*

Proof. We have already shown $\lhd^\circ{\subseteq}\lhd^*$ in L26.7.6, so we need only establish the reverse inclusion. It suffices to show this inclusion on closed terms only, because $\lhd^\circ$ is defined in terms of $\lhd$ relating closures of terms with appropriate $\mathcal{V}$ (D26.5.3), and $\lhd^*$ enjoys the same substitutive properties (T27.9).

So we need to show $\lhd^*{\subseteq}\lhd$ on closed terms. To do this it suffices to show that $\lhd^*$ restricted to closed terms is a post-fixed point of the operator Φ used in Fig.38_{196} to coinductively generate $\lhd$. This follows from T27.11 and T27.14. □

28. Proof of $\lhd^\circ{=}\lhd_{\mathbf{ctx}}$

Lemma 28.16 ($\lhd^\circ{\subseteq}\lhd_{\mathbf{ctx}}$)**.** *$\lhd_{ctx}$ contains $\lhd^\circ$.*

Proof. Since $\lhd_{\mathbf{ctx}}$ is the largest adequate relation (D26.1.6) satisfying the rules of Fig.37_{193} (congruence properties) we need only show that $\lhd^\circ$ has the same properties.

$\lhd^\circ$ is adequate from L26.5.7. $\lhd^\circ$ is a congruence from L27.10 ($\lhd^*$ congruence) and T27.15 ($\lhd^*{=}\lhd^\circ$). Hence $\lhd^\circ{\subseteq}\lhd_{\mathbf{ctx}}$. □

Lemma 28.17 ($\leq^\circ_{\mathbf{kl}}{\subseteq}\lhd_{\mathbf{ctx}}$)**.** *$\leq^\circ_{kl}$ (D26.5.8) is a subrelation of $\lhd_{ctx}$.*

Proof. By L28.16 ($\lhd^\circ{\subseteq}\lhd_{\mathbf{ctx}}$) and L26.5.9 ($\leq^\circ_{\mathbf{kl}}{\subseteq}\lhd^\circ$). □

Corollary 28.18. *If $\Gamma \vdash s =^\circ_{kl} s' : \tau$ then*

$$\Gamma \vdash s \lhd_{ctx} t : \tau \iff \Gamma \vdash s' \lhd_{ctx} t : \tau,$$

and similarly if $t =^\circ_{kl} t'$.

Proof. L28.17 ($\leq^\circ_{\mathbf{kl}}{\subseteq}\lhd_{\mathbf{ctx}}$) and L26.2.4. □

Now for a useful corollary of C28.18:

Lemma 28.19. *Write $\lhd_{ctx\text{-}cl}$ for $\lhd_{ctx}$ restricted to closed terms. Then $\lhd^\circ_{ctx\text{-}cl}$, the open extension of $\lhd_{ctx}$ (D26.5.1), is equal to $\lhd_{ctx}$.*

Proof. The only difficulty in this proof is the number of definitions and lemmas we must unpack. We prove the two inclusions $\lhd^\circ_{\mathbf{ctx\text{-}cl}}{\subseteq}\lhd_{\mathbf{ctx}}$ and $\lhd_{\mathbf{ctx}}{\subseteq}\lhd^\circ_{\mathbf{ctx\text{-}cl}}$ separately.

By L26.5.2, $\lhd^\circ_{\mathbf{ctx\text{-}cl}}$ coincides with $\lhd_{\mathbf{ctx}}$ on closed terms, so it is adequate (D26.1.6) because $\lhd_{\mathbf{ctx}}$ is. From the definition of an open extension in terms of closures $\mathcal{V}$ we see that $\lhd^\circ_{\mathbf{ctx\text{-}cl}}$ is a congruence (N26.2.1). Since $\lhd_{\mathbf{ctx}}$ is the largest adequate congruence, we know $\lhd^\circ_{\mathbf{ctx\text{-}cl}}{\subseteq}\lhd_{\mathbf{ctx}}$.

For $\lhd_{\mathbf{ctx}}{\subseteq}\lhd^\circ_{\mathbf{ctx\text{-}cl}}$ we use the power of $\mathtt{let}\ x = -\ \mathtt{in}\ -$ to carry out 'by hand' the instantiations of free variables executed by a closure $\mathcal{V}$. Suppose Γ is some context with $\Gamma_{\mathbf{typ}} = \{(x_1, \sigma_1), \ldots, (x_n, \sigma_n)\}$ and

$$\Gamma \vdash s \lhd_{\mathbf{ctx}} t : \tau$$

and suppose $\mathcal{V} = (V_1, \ldots, V_n)$ is a Γ-closure (D26.1.1). By applying the rules of Fig.37_{193} repeatedly we can deduce that

$$\Gamma \vdash (\mathtt{let}\ x_n = V_n, \ldots, x_1 = V_1\ \mathtt{in}\ s) \lhd_{\mathbf{ctx}} (\mathtt{let}\ x_n = V_n, \ldots, x_1 = V_1\ \mathtt{in}\ t) : \tau.$$

By C28.18 we have

$$\Gamma \vdash s\mathcal{V} \lhd_{\mathbf{ctx}} t\mathcal{V} : \tau.$$

But $\lhd_{\mathbf{ctx}}{=}\lhd_{\mathbf{ctx\text{-}cl}}$ on the closed terms $s\mathcal{V}$ and $t\mathcal{V}$. $\mathcal{V}$ was an arbitrary closure so we can quantify over it. Thus if $\Gamma \vdash s \lhd_{\mathbf{ctx}} t : \tau$ then for all Γ-closures $\Gamma \vdash s\mathcal{V} \lhd_{\mathbf{ctx\text{-}cl}} t\mathcal{V} : \tau$, and this means precisely

$$\Gamma \vdash s \lhd^\circ_{\mathbf{ctx\text{-}cl}} t : \tau.$$

So $\lhd_{\mathbf{ctx}}{\subseteq}\lhd^\circ_{\mathbf{ctx\text{-}cl}}$ as required. □

This gives us what we were after:

Lemma 28.20. *If $\lhd_{ctx}$ restricted to closed terms is a subset of $\lhd$ then $\lhd_{ctx}{\subseteq}\lhd^\circ$.*

Proof. Recall from L28.19 that we write $\lhd_{\mathbf{ctx\text{-}cl}}$ for the restriction of $\lhd_{\mathbf{ctx}}$ to closed terms and $\lhd^\circ_{\mathbf{ctx\text{-}cl}}{=}\lhd_{\mathbf{ctx}}$. From L26.5.2, if $\lhd_{\mathbf{ctx\text{-}cl}}{\subseteq}\lhd$ then $\lhd_{\mathbf{ctx}}{=}\lhd^\circ_{\mathbf{ctx\text{-}cl}}{\subseteq}\lhd^\circ$. □

Notation 28.21. *In the case that two terms s, t are closed we can use N26.1.4 to write $\vdash s \lhd_{ctx} t : \tau$ as $s \lhd_{ctx} t$. In the light of this and the results above we shall abandon the notation $\lhd_{ctx\text{-}cl}$ and write $s \lhd_{ctx\text{-}cl} t$ as $s \lhd_{ctx} t$.*

Lemma 28.22. *Every type τ is populated by a closed value; there exists some $V \in \boldsymbol{CTerms}$ such that*

$$\vdash V : \tau$$

PROOF. By induction on τ.[91] □

Lemma 28.23. *If $s, t \in$ **CTerms** are closed terms and*

$$s \lhd_{ctx} t \quad \textit{and} \quad s \Downarrow U$$

then there is some V such that

$$t \Downarrow V \quad \textit{and} \quad U \lhd_{ctx} V.$$

PROOF. It follows from the rules in Fig.37_{193} that

$$s \lhd_{\mathbf{ctx}} t \implies \texttt{let}\ x = s\ \texttt{in True} \lhd_{\mathbf{ctx}} \texttt{let}\ x = t\ \texttt{in True}.$$

By construction $\lhd_{\mathbf{ctx}}$ is adequate (D26.1.6) so $\texttt{let}\ x = s\ \texttt{in True} \Downarrow \texttt{True}$ (and it does) then $\texttt{let}\ x = t\ \texttt{in True} \Downarrow \texttt{True}$. It follows from the evaluation rule for $(\texttt{let}\ x = -\ \texttt{in}\ -)$ $(148)_{188}$ in Fig.36_{188} that t evaluates to some value V. Then $U \lhd_{\mathbf{ctx}} V$ from C28.18. □

Corollary 28.24 ($\lhd_{\mathbf{ctx}}{=}\lhd^\circ$)**.** *$\lhd_{ctx}$ and $\lhd^\circ$ coincide.*

PROOF. $\lhd_{\mathbf{ctx}} \subseteq \lhd^\circ$ is L28.16. To show $\lhd_{\mathbf{ctx}} \subseteq \lhd^\circ$ it suffices by L28.20 to show that $\lhd_{\mathbf{ctx}}$ restricted to closed terms (call this restriction $\lhd_{\mathbf{ctx}}$, see N28.21) is a subset of $\lhd$. We can show this by proving that $\lhd_{\mathbf{ctx}}$ is a postfixed point of Φ as defined in Fig.38_{196}. Since all terms we consider are closed we use N26.1.4 to write $\vdash s \lhd_{\mathbf{ctx}} t : \tau$ as $s \lhd_{\mathbf{ctx}} t$.

Suppose $s \lhd_{\mathbf{ctx}} t$. We seek to prove $(s, t) \in \Phi(\lhd_{\mathbf{ctx}})$. If s does not evaluate then $(s, t) \in \Phi(\lhd_{\mathbf{ctx}})$ automatically from the definition of Φ. So we suppose $s \Downarrow U$ and work by induction on U.

1• *Suppose $s \Downarrow \texttt{True}$.* Since $\lhd_{\mathbf{ctx}}$ is adequate (D26.1.6) it follows that $t \Downarrow \texttt{True}$. Similarly for `False`.

2• *Suppose $s \Downarrow 0, \texttt{Succ}(N)$ for $\vdash N : \texttt{Nat}$.* Since $\lhd_{\mathbf{ctx}}$ is adequate, $t \Downarrow N$. This settles the cases $s \Downarrow 0$ and $s \Downarrow \texttt{Succ}(U)$.

3• *Suppose $s \Downarrow c$ for $c \in$ **AtmC**.* We use L28.23 and deduce there is some $d \in$ **AtmC** such that $t \Downarrow d$ and $c \lhd_{\mathbf{ctx}} d$. We now apply the rules of Fig.37_{193} and deduce

$$\texttt{if}\ c = c\ \texttt{then True else False} \lhd_{\mathbf{ctx}} \texttt{if}\ d = c\ \texttt{then True else False}.$$

$\lhd_{\mathbf{ctx}}$ is by definition adequate so if the LHS evaluates to `True`, and it does, then so must the RHS. From this it follows from the structure of the evaluation rules that $c = d$, as required.

[91] Note in passing that if we had fixedpoints at every type (we only have fixedpoints at function types) this result would be trivial, we would just use $\mathit{fix}\, x : \tau.x$.

4• *Supposes* $\Downarrow U = U_1 :: U_2$. We use L28.23 and deduce there is a value $V = V_1 :: V_2$ or $V = \mathtt{Nil}_\tau$ such that $t \Downarrow V$ and $U \lhd_{\mathbf{ctx}} V$. Using Fig.37_{193} we deduce

$$\mathtt{Case}\ U\ \mathtt{of}\ \{\mathtt{Nil}_\tau \Rightarrow B, x_1 :: x_2 \Rightarrow x_1\} \lhd_{\mathbf{ctx}} \mathtt{Case}\ V\ \mathtt{of}\ \{\mathtt{Nil}_\tau \Rightarrow B, x_1 :: x_2 \Rightarrow x_1\}$$

where B is some value of type τ (L28.22).

We use L28.23 again, combined with the evaluation rules, to deduce that $V = V_1 :: V_2$ and

$$U_1 \lhd_{\mathbf{ctx}} V_1.$$

We may similarly deduce that $U_2 \lhd_{\mathbf{ctx}} V_2$.

The cases $U = \mathtt{Nil}_\tau$ and $U = \mathtt{Var}(U'), \mathtt{App}(U'), \mathtt{Lam}(U')$ are similar and easier.

5• *Suppose* $s \Downarrow U = (U_1, U_2)$. We use exactly the same argument as in the previous case. We apply L28.23 to deduce there exists a $V = (V_1, V_2)$ such that $t \Downarrow V$. We then use Fig.37_{193} we deduce

$$\mathtt{Fst}(U) \lhd_{\mathbf{ctx}} \mathtt{Fst}(V).$$

From this, using L28.23 once more and the evaluation rules, it follows that

$$U_1 \lhd_{\mathbf{ctx}} V_1.$$

We may similarly deduce that $U_2 \lhd_{\mathbf{ctx}} V_2$.

Recall that we are proving that $\lhd_{\mathbf{ctx}}$ is a post-fixed point of Φ defined in Fig.38_{196}. The pattern of proof is precisely the same as in the previous cases, so from now on we simply give the appropriate 'context', built using the rules of Fig.37_{193}, that verifies the conditions of Fig.38_{196} for each possibility for U.

6• *Suppose* $s \Downarrow U = \mathtt{fix} f(x : \tau)\ \mathtt{in}\ s'$. For all $W \in \mathbf{CVal}_\tau$ we have

$$UW \lhd_{\mathbf{ctx}} VW.$$

7• *Suppose* $s \Downarrow U = a.U'$. For new c and $V = b.V'$,

$$U@c \lhd_{\mathbf{ctx}} V@c.$$

We then apply L28.23 and deduce

$$(c\ a)\cdot U' \lhd_{\mathbf{ctx}} (c\ b)\cdot V'$$

as required.

□

29. The Sanity Clause, proved

Let us go back now and finish what we started in §21:

Theorem 29.25 (The Sanity Clause (with proof))**.**

- *The syntax of* FreshML *is defined in §22.1.*
- *The typing rules are defined in §24.1.*
- *The evaluation rules are defined in §25.*
- *The contextual preorder $\lhd_{\boldsymbol{ctx}}$ is defined in §26.2.*
- *α-equivalence on closed values of the type Λ_α (recall, representing possibly open terms of the untyped λ-calculus), written $\equiv^{se}$, is defined in D26.3.1.*

Given this, for all closed values $U, V \in \boldsymbol{CTerms}$ such that $\vdash U, V : \Lambda_\alpha$,

$$U \lhd_{\boldsymbol{ctx}} V \iff U \equiv^{se} V.$$

Proof. C28.24 says $\lhd_{\mathbf{ctx}} = \lhd^\circ$. L26.5.6 says $\lhd^\circ = \lhd$ on closed terms, an in particular on closed values. From the case of the type Λ_α in T26.4.7 we have

$$U \lhd V \iff U \equiv^{se} V$$

which gives us the result. □

FreshML is intended as a prototypical example of a language with the features that FM (Chapter II) inspires, as well as providing the object of a particularly large case-study in inductive reasoning in FM. The Sanity Clause brings this all together by proving, using FM, that a language it has inspired really does represent it in a 'sane' way.

30. Questions

30.1. ML-style evaluation? FM set theory has no constants of type $\mathbb{A}$ and FreshML has constants $a, b, c \in \mathbf{AtmC}$ of type `Atm`. So we may wonder:

> "We use 'equivariance' in L26.7.2 yet T9.1.6 (equivariance FM) depends—simplistically put—on there being no constant symbols in $\mathbb{A}$ in the language of FM. Isn't there something wrong here?"

The technical answer is:

> "No. FreshML is modelled as an object-language inside FM set theory, and therefore has no connection with the language of FM itself."

But the relation of FreshML and FM is a little incestuous. After all, we began our discussion of FreshML in §21 by suggesting terms could denote functions in FM set theory. If the denotation of the FreshML constants of $a, b, c \in \mathbf{AtmC}$ is not FM *constants* $a, b, c \in \mathbf{AtmC}$—then what is it? The technical answer above

says, correctly, that they are FM *variables* ranging over the set **AtmC** just as x, y, z range over **Var**.

Still, we *could* eliminate syntactic constants symbols $a, b, c \in \mathbf{AtmC}$, they would be replaced by syntactic variables $x, y, z{:}\mathtt{Atm}$ in the typing context. $(163)_{188}$ would change because $c \in \mathbf{AtmC}$ is outlawed. Evaluation would carry an ML-style evaluation context associating to some variables $x \in \mathbf{Var}$ constants $a \in \mathbf{AtmC}$, and our notion of 'closed value' would change accordingly.

But FreshML was never designed from programming. It was designed for proving the Sanity Clause (§29). I did not want labelled evaluations and 'closed' values that actually contain free variables in my proofs.

30.2. Combine typing and apartness? We defined apartness judgements first and then used them to define typing judgements (§23 and §24). Why not combine them and define them together? I.e. why not take the judgement

$$\Gamma \vdash t : \tau \# \overline{xc}$$

as primitive, not shorthand (Item 1 of N24.1.2).

We can do this but it causes problems. Because $\lhd^*$, $\lhd$, $\lhd^\circ$, $\leq_{\mathbf{kl}}$ and $\leq^\circ_{\mathbf{kl}}$ are all defined on typeable pairs of terms of the same type, if apartness and pure typing judgements are defined together we carry apartness judgements $\#\overline{xc}$ in all the inductive and coinductive reasoning on types. Now we can always strengthen our inductive hypothesis and universally quantify over $\overline{xc}$ where necessary. I have tried this. However, the 'black lace' effect of a hundred '$\#\overline{xc}$'s on every page was most unpleasant, and since I have proved what I need without doing this, it is also demonstrably unnecessary.

30.3. Incorporate apartness into types? We could extend the typing system to include types like $\tau\#\overline{xc} \to \sigma\#\overline{yc}$. We could then type

$$\vdash F = (\mathtt{fix}\, f(x:\tau)\, \mathtt{in}\ a.x) : \tau \to \big([\mathtt{Atm}]\tau\#\{a\}\big)$$

and use this to deduce

$$x : \tau \vdash F(x)\#a.$$

We might similarly be able to type 'silly' (but acceptable) functions like $(116)_{177}$ and not-so-silly functions like the pretty-printing function mentioned just below $(116)_{177}$.

I have tried designing such a type system, it turned out to be rather complicated. We should be mindful of this idea for future languages, if only to understand why it might be impractical. In $(\mathtt{Subst}'(t))_{186}$ I hint at how this complexity may be unnecessary after all.

30.4. Define # 'co-inductively'? That is, instead of taking the basic apartness judgement to be $t\#x$, why not take it to be $\neg(t\#x)$ or—loosely speaking because t is in FreshML not FM—$x \in \mathbf{Supp}(t)$.

The reader is welcome to consider Fig.32_{169} and Fig.33_{170} and think how we might reformulate them in terms of these 'negative' apartness judgements. Let us write $a \boxplus x$ for $a \in \mathbf{Supp}(x)$. Should we change apartness contexts to $\Gamma_{\boxplus}$, finite sets of ordered pairs written $x \boxplus y$, so an apartness judgement $\Gamma_{\boxplus} \vdash t \boxplus xc$ means "if $y \in \mathbf{Supp}(x)$ for all $(x, y) \in \Gamma_{\boxplus}$ then $xc \in \mathbf{Supp}(t)$"?

We might even keep $\Gamma_{\#}$ but make apartness judgements of the form $\Gamma_{\#} \vdash t \boxplus xc$, meaning "if $x\#y$ for all $(x, y) \in \Gamma_{\#}$ then $xc \in \mathbf{Supp}(t)$".

I shall make just these two remarks:

- The natural thing to have on the right seems to be #. Consider the condition $\Gamma_{\#} \vdash yc\#xc$ in $(98)_{169}$.
- The most natural thing to have on the left seems also to be #. Consider the contexts $\Gamma_{\#}\#x$ in $(99)_{169}$ and $\Gamma_{\#} \cup xc\#yc$ in $(100)_{169}$.

31. FreshML and automation

In conclusion there are a few points I would like to make about the Sanity Clause (T29.25) and automation, particularly in Isabelle/FM (Chapter III).

1. Chapter IV *cries out* to be automated. It is long, it is complex, it has a lot of relatively trivial cases. Has the reader checked every line of this proof? If so, can they guarantee that they are correct? *What?* That is *my* responsibility? And why believe me? So the size and yes, monotony of this proof invites automation.

   The reader is also invited to observe that (thanks to FM) every proof we have seen is by strict induction or coinduction (as appropriate) on a (co)inductively defined set.[92] By 'strictly' I mean that I never used *any* subsidiary 'renaming', 'weakening' or 'congruence' lemmas. For example, we prove T23.1.12 but never use it.[93] The proofs are always the same: apply the (co)inductive principle and then resolve away the subgoals until there is nothing left. In other words this whole proof is no more than standard resolution-based theorem-proving, but carried out on paper instead of a computer.

[92] Except naturally for those having nothing to do with inductively defined sets such as L23.1.9, C23.1.10 and L24.1.5, or L26.5.2. Except also for L26.4.4, which we could prove by co-induction only FM gives it to us 'for free'.

[93] The one 'renaming' lemma we do use is L26.4.4 but it describes α-equivalence on values of type Λ_α, which represent λ-terms *not* up to α-equivalence. In a strong sense this is the whole point of FreshML.

Note also that this uniformity is characteristic of syntax *without* variable binding, but has been conspicuous by its absence from developments of syntax *with* binding. The fact that the proof has been boring and easily automated is therefore very good. See Item 3 below.

2. *Why is the proof so long?*
   1. The representation of these (co)inductive proofs on paper is not particularly compact. For example the cases in the proof of T27.11, when written in full, occupy about half a page each—but they are nothing more than "resolve against Eq_1 to produce two subgoals. Eliminate the first by resolving against Eq_2 and the second by resolving against Eq_3", and so on.
   2. We (often) have to write out the proof-state at each step. A theorem-proving environment would do that for us. For example $(206)_{214}$.
   3. Paper does not have `ALLGOALS`, `REPEAT`, `ORELSE` or `blast_tac` so that part of themselves which the proofs of the cases have in common cannot be distilled to a single "`by (ALLGOALS` *tactic*`)`" at the head of a proof (followed by specialised code to clean up the debris). Instead we must redescribe the tactic for each case or omit the proof (with an excuse like "proof just like the last case" or "proof trivial").

      E.g. consider the proofs of T27.9 or the more complicated T27.14. In this latter proof there is a telling sentence half-way through case 4: "Following the same pattern as the previous case". In my experience the cases of these large inductive proofs do have a general 'shape' in common which paper expresses worse than computers.

3. *If this proof is "standard resolution-based theorem proving", why is more of the literature not automated (e.g.* [**64**]*,* [**65**]*,* [**42**]*)?*
   They did not have FM, which gives us structural inductive principles. Without it we either dishonestly rename bound variables or do induction on the length of a derivation rather than its structure. Both are messy. With 'neat' proofs like those of Chapter IV, who needs 'messy'?

4. *Could this proof be carried out using de Bruijn datatypes of syntax (§33.1)? After all, the de Bruijn datatypes have semantics isomorphic to the* FM *datatypes.*
   We commented in *p.*13 that it is not enough that a datatype be isomorphic to what we want, it must have the right inductive structure. The structural inductive principle for a de Bruijn datatype does not correspond to 'taking subterms', which complicates inductive hypotheses.

5. *Could this proof be carried out in HOAS?*
   Not in its present form. We would have to abandon set theory and Isabelle to avoid exotic terms ([**49**, §11.2.2, p.132 or Example 11.8, p.135] or [**12**, §5, p.8]). The *real* point is that HOAS does not give inductive reasoning principles (though see [**15**]). Cf. §33.2.

   Of course we might be able to carry out a *different* proof of the Sanity Clause (something like [**34**][94])—but HOAS has nothing like the facilities necessary to support *this* proof.

6. *Why have I not implemented Chapter IV in* FM*?*
   I know what has to be done to implement this proof, see R19.3.1 and R20.3, but I also know I do not have the time to do it in this thesis. With the modifications of R20.3 in place an implementation would not even be particularly difficult. I have already developed all the underlying results (e.g. I have mechanised §13.2, a nontrivial task which I do not discuss) and designed the proof in detail on paper. I did so with an Isabelle/FM implementation specifically in mind, not only in the strict structural-induction style of my proofs (see Item 1 of this list) but also in some of the fine detail of the inductive sets' design.

   I shall indulge myself and show one of these details. This one is rather amusing, perhaps the reader noticed it. $(162)_{188}$ is of the form

   $$\text{“}\frac{V' = (b\ a)\cdot V}{(a.V)@b \Downarrow V'} \quad (a.V\#b)\text{”}$$ and not “$(a.V)@b \Downarrow (b\ a)\cdot V \quad (a.V\#b)$”.

   The second version is superficially 'simpler' but were the reader to program it into a resolution-based system he or she would quickly discover that it refuses to intro-resolve with anything that does not have conclusion with RHS of the form $(b\ a)\cdot V$. This is not at all what we meant, of course.

[94] Note, one could argue that [**34**] is an 'investigation' rather than a 'proof'. Discussion omitted.

Chapter V

Conclusions

32. Commentary on FM

32.1. FM's great problem: inaccessible.

Remark 32.1.1. At the end of §5.1 I said that FM is 'new', but insisted it is not 'exotic', 'difficult' or 'strange'. On the contrary, FM seems to capture our intuitions about binding very well.

But that is not the whole story. FM does demand a dedicated system, and even if this system's *use* is easy its initial *construction* need not be.

What does that mean? For example to 'do' FM-style datatypes with binding in a set theory I needed to construct FM set theory first. ZF was *not* up to the job. That was not too bad, but when I wanted to automate this in Isabelle I had to implement Isabelle/FM first—and that took a whole year. Of course this job is behind us and the rest of the world can benefit from my work without shedding a tear for *me*, but the list goes on: What if the reader wants COQ or HOL, not ZF? I have nothing to offer (for the moment). Similarly, before we write programs on FM-style datatypes we have to implement a language that supports them.[95] And so on. This contrasts with other most other approaches to syntax with binding (§33), which can be used for the most part on any current system. ◇

The following terminology will be useful. I shall use it for the rest of this document mostly without comment.

Notation 32.1.2 (Accessible)**.** *I shall call an approach to binding* ***inaccessible*** *if we cannot use it right now on our average system, and* ***accessible*** *if we can. Thus* FM *is inaccessible and de Bruijn nameless terms (§33.1) are.*

So a drawback to FM is that it is inaccessible.

The following merely reiterates R4.14, only this time we have the development of FM behind us.

Notation 32.1.3 (Name-carrying, nameless, nameful)**.** *Call a term a* ***nameless term*** *if bound variables really are bound (e.g. de Bruijn terms, §33.1, or* FM*-style terms). Call a term a* ***name-carrying term*** *when bound variables are not bound and their names are therefore 'carried' in the abstract syntax of the term (e.g. §33.4).* FM *datatypes allow us to freely invent new names for bound variables and hence treat them as if they did have names. We call this a '****nameful****' reasoning style.*

Datatypes with nameless terms are usually setwise isomorphic to corresponding name-carrying datatypes up to α-equivalence. Recall from R4.14 that FM

[95] This has not yet been done, work includes Chapter IV and [**66**].

gives us all the benefits of nameless terms while still allowing us to reason in a nameful manner, as if about a name-carrying datatype.[96]

32.2. Usefulness of FM**.** In this subsection I shall consider various situations in which FM is more, or less, useful. Suppose that ...

1• ... *we want syntax as a means to an end.*

For example, in [**53**] Milner and Parrow define the syntax of and a transition system for π-calculus (incidentally, using name-carrying terms) and then prove results about them. They do not care about the syntax as such, it is simply a way of capturing "on paper" the object "in nature" which they want to study. Similarly in say, [**7**].

These researchers would conceivably be quite content to use FM-logic as an algebraic system for reasoning about bound names. They certainly do not have to understand the minutiae of FM sets to do this, any more than one has to understand the minutiae of ZF to use FOL.

И accurately captures our intuitions about manipulating bound names and a manifestation of this is that reasoning in FM-logics seems almost indistinguishable on paper from the material currently produced by researchers.[97] It seems plausible to me that researchers could use FM-logic with a minimum of fuss, and produce material comprehensible to those who do not know FM.

2• ... *we want to write a program to manipulate syntax-with-binding.*

As in the previous scenario FM is a means to an end. Assuming that an FM-style language becomes available (and this is the big 'if'; there are none at the moment, see R32.1.1), the programmer can just write his or her program.

The reader can see example programs written informally in the kind of language I envisage in §4. Programs in a weaker but rigorously constructed language are in §22.2 and §24.2.

There *are* subtleties to FM-style languages. But in my opinion—and after looking at the examples I would hope the reader might agree I may be right—a practical programmer would quickly get the idea.

The real world is a messier place than the theory to which most of this thesis is devoted. I discuss how FM-style languages might contribute to the one application of syntax with which I am most familiar, namely theorem-proving systems, in the course of §33.4.

[96] Though there are necessary limits to this emulation, see R33.4.1.

[97] Concrete evidence supporting this claim appears in this thesis. The mathematics of Chapter IV is conducted entirely FM-style (see R21.4) yet proofs like that of T24.1.7, L24.1.8, T27.11 look traditional.

3• *... syntax itself is our object of study.*

Here, certainly, a researcher would have to become familiar with the technical details of FM in whatever universe the researcher prefers, let us stick with sets for the sake of argument.

The development of FM sets in Chapter II is certainly long and technical (cf. Item 3 of §3). Of course a really rigorous development of *any* underlying universe is technical. Yet that misses the point. We are not doing set theory for its own sake, we just want to give semantics to syntax. Thus what interests us is ultimately *not* the precise set-theoretic implementation of abstraction-sets, any more than that of pair-sets or lists. These implementations are quite arbitrary *except* that they must have the right algebraic and logical *properties*, because it is these properties that interest us. So I would urge the reader not to worry too much about the length of Chapter II. The real issue is the beauty, or at least the utility, of the results we prove about our complicated constructions. And they come out rather well.[98]

32.3. FM **not panacea.** I believe in FM, it has been very successful. Yet as successful as it is, sometimes it may just be irrelevant. So let me mention just two examples that on just fine without FM.

If we only ever need to substitute for closed terms, capture of free variables—and α-equivalence—is not a problem. In particular substitution for closed values/terms is all we need for proving some properties of run-time systems, for example type-soundness. According to Tobias Nipkow ([**32**]), the Bali project [**71**] has verified type-soundness for a simplified but significant subset of Java, and they never had to think about α-equivalence once. Java is imperative and so one imagines it would be quite well-suited to this because binding plays a little less of a rôle than it would in a more functional language. Nevertheless, their subset has methods and objects and local variables are bound in them. But because run-time behaviour is their object of interest they avoid most of the problems which FM addresses.

This is not to say that the Bali project need not use FM. Its abstraction types might simplify the handling of bound variables. Then again, maybe they would not; the designers might want to label bound variables with names for pretty-printing as discussed in and around R33.4.1, and as discussed there, that *might* effectively bring us back where we started to ZF-style name-carrying terms. Then again, maybe not.

[98] **At the time of going to press** there is a draft paper [**6**]. There the authors import what they need of FM techniques and logics to apparently satisfactory effect, and without having to build all of FM set theory. So it really does seem to work.

Another example of an application that would not need FM is a theorem-prover called SATCHMO, [**4**], brought to my attention by Francois Bry. The system is 12 lines of Prolog (Bry apologised for the length and explained it would be shorter but for pretty-printing) and took two years to write. Bry assures me that SATCHMO makes sophisticated non-evident use of Prolog; it is a genuine system, not just a front-end. However, it hijacks Prolog's syntactic system and therefore has no need for its own datatype of terms,[99] and sidesteps this entire thesis.

It is not that I see a shortage of applications for FM. On the contrary, I think it will be very useful. I merely wish to emphasise that FM is a good solution, but only to a particular problem.

33. Other approaches and the literature

First I shall reference a few articles and web sites. [**62**] is a survey of 'logical frameworks' with some emphasis on the issue of syntax. Logical Frameworks have a homepage [**56**] with a useful link [**73**] to researches in the area. Incidentally "logical framework" is used in a more specific sense than "theorem-proving environment", see the homepage. In concrete terms this means that [**56**] does not include COQ, among others. However, there is a link there to the COQ homepage [**27**] anyway. I should also mention a relatively early paper [**75**] by Stoughton. In the introduction he discusses the problem of substitution and binding and gives references.

Recall the terminology 'accessible', 'inaccessible', 'name-free term' and 'name-carrying term' established in N32.1.2 in N32.1.3.

Now we consider the various other approaches to syntax with binding.

33.1. De Bruijn. The idea was originally presented in de Bruijn's [**11**]. The de Bruijn-style type of untyped λ-terms is

$$\mu X.\texttt{Var of } \mathbb{N} + \texttt{Lam of } X + \texttt{App of } X \times X. \tag{207}$$

There are many variants but the usual idea is that $\texttt{Var}(n)$ represents the $(n-i)$th variable name if there are i occurrences of `Lam` above this $\texttt{Var}(n)$ in the abstract syntax of the term for $i \leq n$, and represents the variable bound by the nth `Lam` up from $\texttt{Var}(n)$ otherwise. Thus n encodes either a pointer to a particular binding operator `Lam` or a reference to a free variable.

[99] Unlike Isabelle, say, which hijacks the ML command-line but uses it as a semi-imperative meta-language to manipulate an ML datatype of terms.

De Bruijn techniques are accessible (N32.1.2) and produce nameless terms with semantics isomorphic to the syntax-up-to-α-equivalence. They are widely used in programming applications, program-verifications, and more.

Remark 33.1.1. However de Bruijn datatypes have a rather revolting inductive structure which makes it difficult to write programs using them (cf. `Nat` and `oNat` on *p.*13). For example, if we want to unpack the body of an abstraction (e.g. remove λ in a term representing $\lambda x.t$), we have to write a program to go through the term relabelling all the indices (as seen in Fig.1_{20}). ◇

So advantages are:

- Terms are nameless, we do not have to worry about respecting α-equivalence and a class of bugs is ruled out.
- For a *given* application there are only a finite number of functions we need out of the datatype (possibly just substitution, 'body of a λ-term', 'top-level term-former', 'is-well-formed' and a couple more). We may cry and beat the walls while we define them, but then it's over and we can get on with things.
- This is true in verification as well. There the functions on syntax are defined by and limited to those declared in the specification of the program.

... and disadvantages are:

- People can get used to almost anything including this, but still, programs come out messy. See R33.1.1 and Fig.1_{20}.
- This messiness is not just a cosmetic issue. When we unpack a body of an abstraction we have to relabel variables as described in R33.1.1. With huge terms (such as arise in hardware verification) this causes efficiency problems. For this reason HOL-light ([**29**]) uses name-carrying terms and not de Bruijn, we shall return to this in §33.4.
- In a theoretical treatment of syntax the goal may be not only to define a function but to then study it. If the definition of the function is revolting so is its analysis (see R33.1.2). Ironically de Bruijn introduced his indices in [**11**] precisely to reason about nameless terms and a particular relation on them (Church-Rosser for the λ-calculus). However, the twisted inductive principles of de Bruijn seem too much for modern researchers. They prefer either name-carrying techniques (as in [**53**]) or HOAS (as in [**34**], see also §33.2).

Remark 33.1.2. However, Hirschkoff's recent thesis [**24**] (in French, resuméd to some degree in [**25**]) uses de Bruijn-style to execute a complete analysis of the π-calculus and bisimulation equivalence in COQ. Apparently 75% of the author's

COQ lemmas control de Bruijn indices. The man must have incredible patience. I am sure no-one wants to go through what he must have gone through again. But if they use de Bruijn, they will. ◇

I shall discuss de Bruijn further when I compare it to name-carrying techniques in §33.4.

I mention in conclusion that Bird and Paterson in [**2**] build a de Bruijn style datatype with a clever structure which avoids many of the problems discussed above. It has its disadvantages but the point is that sheer cunning can get good results. See in particular [**2**, §4, p.10].

33.2. HOAS. HOAS stands for Higher-Order Abstract Syntax. Miculan's thesis [**49**] is an excellent document and [**49**, Chapter 11, p.125] is devoted to a clear discussion of HOAS. A roughly similar but more concise, advanced discussion is in [**12**]. [**51**] is a clear historical and technical account of HOAS techniques.[100] [**52**] is a paper discussing logics for reasoning on HOAS and includes references for that part of the field. I also recommend a survey of Logical Frameworks [**62**], in particular [**62**, §5, p.8] is a discussion of HOAS with plenty of references. [**45**] is not a survey article in itself but [**45**, §5, p.8] is a survey.[101]

HOAS is a large field to which I cannot do justice here. I shall briefly mention the basic idea, otherwise I refer the non-expert reader to [**49**, Chapter 11, p.125] as a good exposition with plenty of examples. From now on I shall assume the reader is familiar with HOAS.

Briefly, HOAS is when we interpret binding term-formers as functions ('meta-level abstractions', as it is often put). The datatype of syntax of λ-terms becomes

$$\mu X.\texttt{Lam of } (X \to X) + \texttt{App of } X \times X. \tag{208}$$

[**49**, §11.2.1] describes in the vocabulary of COQ ([**27**]) how things go horribly wrong with this approach. We see the same problem in sets: the function space $X \to X$ is too large and the fixedpoint does not exist. [**49**, Example 11.6, p.131] is a case where the approach of $(208)_{233}$ does work, but often it does not. One attempt at solution is to hypothesise a type of atoms **Var** and redeclare our datatype as

$$\mu X.\texttt{Lam of } \mathbf{Var} \to X + \texttt{App of } X \times X. \tag{209}$$

Features of HOAS usually portrayed in a positive light are:

1. Substitution is handled by the meta-language.

[100] Which is very up-to-date. So much so in fact that it is unpublished at the time of writing. It is available on the net.

[101] No misprint, the page and section numbers really are the same in both.

2. α-equivalence is handled by the meta-language.
3. Object-level variables disappear and are replaced by meta-level variables.

This does not impress me.

1• *HOAS datatypes do not have inductive structure and do not yield structural recursion principles.*

The types $\mathbf{Var} \to X$ and $X \to X$ do not have inductive structure. This means the datatypes of $(208)_{233}$ and $(209)_{233}$ have no structural inductive principles. Functions out of them cannot be defined by structural induction.

For an example of how researchers try to get around this see Despeyroux and Hirschowitz [**14**]. This does not seem to quite work. A later attempt is Despeyroux, Pfenning and Schürmann [**15**] (short version) and [**13**] (long version). I would very much like to dissect their paper but there is no space here. One manifestation of the problems HOAS has with recursion is that we cannot write down simple pattern-matching iterative definitions like those of §4 or even §22.2 because of the functional type in the recursive definition. Despeyroux et al therefore try to sidestep pattern-matching. Instead they 'replace' the constructors of the initial datatype (e.g. λ-terms) with constructors of the target type (e.g. natural numbers), which produces an 'iterative' function from the first (λ-terms) to the second (natural numbers). (This is their example *cntvar*.)

- It is complicated.
- The authors' λ-calculus is not given any underlying semantics (aside from itself), which is unsatisfactory. Without an underlying (or if the reader prefers, *alternative*) semantics, how do we know there isn't a much better λ-calculus just round the corner. There probably is. How do they set about finding it? [**26**] independently addresses this point and produces presheaf semantics for the work in [**13**].[102]

2• *Therefore, is difficult to reason about and define functions on HOAS datatypes.*

Surely not? In [**13**] and [**15**] Despeyroux et al present a methodology for recursion on HOAS datatypes and [**26**] gives it a semantics. Is this not just what we wanted? Well, I already mentioned that [**13**] is not *quite* recursion as such, it is something a little different to do with 'replacement' rather than 'pattern-matching'. Yet I myself am anxious to convince the reader that

> "FM is not *quite* ZF, but close enough to be useful and anyway it's quite harmless so don't worry about it"

[102] However, the construction is ad hoc in the sense that presheaf categories can give semantics for almost everything. In a sense, a system that explains everything, *tells us* nothing.

—so perhaps I had better be open-minded about eccentricities in other people's work. And indeed, I do not object to the 'exotic' primitive recursion of Despeyroux et al. It's perfectly reasonable, it just comes out complicated.

Abstraction types $[\mathbb{A}]X$ have *far* better algebraic properties than function-types $X \to X$ (see C9.6.9) and FM delivers completely standard iterative definitions without a hint of a struggle (see §10 for details, §5.2 for an overview). Thus in [**15**, p.8 and p.9] the informal iterative definitions of two functions *plus* and *cntvar*, which serve in [**15**] as the *beginning* of the rest of their paper, would in FM just be the definitions of the desired functions: for example I define a function `FVlist` similar to *cntvar* in $(\texttt{FVlist})_{166}$. Instead of the number of free variables it returns a list of them. If the reader can pardon the concrete syntax, it is clearly just the informal definition of *cntvar* given in [**15**]. Notice that no pure HOAS system can ever define `FVlist` because in HOAS variables have no object-level names.

Similarly FreshML (Chapter IV) is a typed λ-calculus corresponding to (though completely different from and almost *melodramatically* simpler than) that of [**15**]. It is *derived from* from FM-sets, not plucked out of thin air and then given semantics as is the case for [**15**] and [**26**].

We move on from [**15**]. Honsell, Miculan and Scagnetto place a different emphasis in [**34**]. They are concerned with practical reasoning on a particular HOAS datatype of terms for the π-calculus. I know this paper quite well because I mirrored some of it on paper in FM and even began to implement my paper development in Isabelle/FM. As they say in their conclusions, the great benefit of HOAS is that it gives them nameless terms but without the trouble de Bruijn causes (cf. §33.1 and in particular R33.1.2). The price is that their datatype does *not* have inductive principles, so they axiomatise the properties and functions they need (e.g. "is in the free names of").

It is interesting that their axioms are reminiscent of certain useful properties of FM, although in FM they are all derived, ultimately from the 'FM axiom' $(\text{Fresh})_{35}$. E.g. `unsat` on p.20 corresponds to C9.2.5.

3• *HOAS leads to complicated reasoning and programming principles.*

Never mind programming, what about logic? HOAS terms are at least second-order, so we cannot reason about them with First Order Logic (like we do in FM). Thus HOAS generates a need for new logics with enough strength at higher orders to manipulate the terms, but not so much that they become too powerful. After 'logic' comes 'logic programming', e.g. unification. If the logic

is too powerful unification becomes undecidable—how *do* we unify terms of type $X \to X$ or $\mathbf{Var} \to X$?

Relatively recent research in this area includes McDowell's thesis [**44**], McDowell and Miller's [**45**] and the same authors' most recent paper [**46**].

FM-abstraction types are first-order objects and FM-logic (§5.1) has been quite sufficient to manipulate them logically. Some discussion of this is in §12. The logic of FM is also *classical*. The logic $FO\lambda^{\Delta\mathbb{N}}$ of the papers referenced above is not.

4 • *HOAS has exotic terms.*

This problem is peculiar to HOAS. See and [**49**, §11.2.2, p.132 and §11.3.1 p.135] (elementary) or [**14**, §2.2, p.6] (more advanced). This is a tremendous problem for HOAS for various reasons. I mention just one of them: they put a lot of junk in a HOAS datatype that should not be there. If we want to reason about this datatype we need some way of eliminating exotic terms from consideration. Why?

- Our theorem may well not be true of them.
- They are not constructed using the term-formers of the datatype and as such are part of the 'no structural recursion' problem discussed in Item 1.

Now let us consider some of the so-called 'advantages' of HOAS listed on *p.*233.

5 • *Object-level variables disappear.*

Observe in $(208)_{233}$ and $(209)_{233}$ that there is no `Var` term-former. If we want open terms in HOAS we model them as functions $f : \mathtt{term}^n \to \mathtt{term}$, see [**14**, §1 p.3].[103] I do not like this. It is just another complication.

6 • *Substitution is handled by the meta-language.*

I think the reason this is pleasing is that, were it not the case (and without recursion) HOAS would be completely stuck. This is no blessing, it is utter damnation averted. For other approaches which give inductive principles of one kind or another, substitution is just one of a number of interesting functions we might define out of a datatype.

Sometimes HOAS gives us substitution even when we do not want it. For example, in the π-calculus we want to bind channel names *without* allowing name-for-process substitution. This is discussed in [**49**, Chapter 11, p.128, Examples 11.1 and 11.2].

[103] This takes the philosophy of HOAS, to model object level variables using meta-level variables, to its logical conclusion.

7• *α-equivalence is handled by the meta-language.*

Yes. HOAS terms are nameless!

This brings the list to an end. The reader should be clear precisely *what* I am criticising. It is standard to give (for example) $\forall$ and $\exists$ on a type X semantics as higher-order functions

$$\forall, \exists \colon (X \to b) \to b$$

where b is the type of truth values. Thus $\forall x \in X.\phi$ is interpreted as $\forall_X(\lambda x : X.\phi)$. *This* idea is very successful (it goes back to Church's famous paper [**8**]). Note that it concerns the semantics of binders. HOAS concerns the semantics of the *syntax* of binders. The latter is under scrutiny here. Let me illustrate this with an example. The Isabelle/FOL universal quantifier `All` is typed as `('a => o) => o`. This is a classic higher-order encoding of the quantifier. However, the syntactic term-former we manipulate is just a function from `term` to `term` with associated typing conditions. True higher-order *syntax* would be if `All` took an ML-function from `term` to `term` and returned a term in `term`.

Remark 33.2.1 (A little philosophy)**.** One of the important techniques of science is to 'explain' phenomena in terms of 'simpler' concepts. Thus 'AN OBJECT IS BLUE' is 'explained' in terms of wavelengths of light reflected. FM 'explains' binding in terms of $\mathbb{A}$ and $И$. De Bruijn 'explains' it in terms of pointers to variables. HOAS is different. Its great claim is that it 'explains' variables in terms of variables in the meta-language—but is this not suspiciously like 'explaining' 'AN OBJECT IS BLUE' by saying 'THE OBJECT IS COMPOSED OF BLUE PARTICLES'? This paragraph may be just words, but see [**49**, §11.1, p.127] for a concrete manifestation of what I am worrying about. ◇

In spite of all I have said, sometimes, with the right language and the right meta-language, HOAS can work. For example in her thesis [**19**],[104] Gardner shows that terms of FOL are in bijection with terms of ELF$^+$ (a logical framework she introduces in her thesis, see [**19**, Chapter 3, p.45]) in $\beta\eta$-normal form (i.e. what I would probably call 'values'), see [**19**, T5.1.9, p.101]. Furthermore the *proofs* of FOL can be represented with complete accuracy in the framework [**19**, T5.2.7, p.119]. Similar results hold of HOL (E5.1.10 and E5.2.8 respectively). Even so in C5.1.8, E5.2.6, or T5.2.13 Gardner discusses how the results can break down for other logical systems. I would refer the interested reader to Miculan's thesis [**49**]. It contains what seems to be a systematic programme of attempts to represent

[104] Thanks to Randy Pollack for drawing my attention to this work.

various logics in CIC, the results of which enquiry are summarised in [**49**, §18.1, p.200].

33.3. Combinators. Since α-conversion causes so much trouble, why not use combinators ([**10**])?

We can. Camilleri and Melham decided to do just that in [**5**].

If we just want to avoid sinking into that particular bog which this thesis hopes to drain and landscape over, fine. But as a philosophical position it is untenable. It commits us to reasoning using combinatory logics, programming using, say, combinatory versions of C and ML—perhaps even abolishing the words 'every', 'a few', 'all', and 'some' from the English language (and similar) and replacing them by combinatory versions.

Assuming that interest in traditional methods and languages persists, the problem of syntax with α-equivalence is there to be solved.

33.4. Name-carrying terms. Recall the terminology 'name-carrying' and 'nameless' from N32.1.3. In name-carrying techniques abstraction types are interpreted by $\mathbb{N} \times X$ or $\mathbb{A} \times X$ (or similar).

For an excellently clear exposition on the difficulties traditional name-carrying terms have with α-equivalence I refer the reader to the slides of a talk by Pollack [**74**] and also Stoughton's [**75**, §1].

I consider practical applications first. In §33.1 and particularly R33.1.1 I mentioned that de Bruijn-style terms can suffer efficiency problems because we cannot break up terms without relabelling indices. In name-carrying terms this is not a problem, the body of `Lam(x,M)` is just `M`. Name-carrying has its own problems:

- Equality testing, matching, or unification are up to α-equivalence (with associated computational cost).
- The datatype must be wrapped in an abstract datatype (with some fixed collection of basic interface functions) to preclude programs which may not respect α-equivalence. This reduces programming flexibility and ...
- ... the interface theorems are complicated, to handle renaming of bound names. Bugs are possible and a computational price guaranteed.

So in practical applications there is a trade-off between de Bruijn and name-carrying terms. Isabelle uses de Bruijn, I am told that HOL88 (the original HOL) used name-carrying terms, HOL90 switched to de Bruijn. Apparently HOL98 went back to de Bruijn but was worked on by people from the COQ system and now uses explicit substitutions in a way I have not researched. HOL-Light uses name-carrying terms.

Remark 33.4.1 (Nameless terms and pretty-printing)**.** However, users like to see terms printed with explicit variable names so even the systems that ostensibly use 'nameless de Bruijn' often label the bound variables with names anyway. De Bruijn guarantees we respect α-equivalence, and the labels help us pretty-print. ◇

Does FM have anything to contribute? FM-style languages simply give the 'up to α-equivalence' of de Bruijn but with the added extra of 'natural recursive principles' and 'N and **fresh**'. Investigations in FM-style programming suggests it works excellently (for informal examples see §4, for examples in a rigorously defined language see §22.2. See also [**66**]). A project has been applied for to implement an FM-style language based on ML.

Thus one can expect programs in FM to be easier to handle than de Bruijn, and if we want name-carrying terms we can adopt the trick mentioned above of using nameless terms with extra name-carrying labels. Such programs should be easier to write, debug, and indeed to formally verify. Note that if we do use extra name-carrying labels we re-introduce the problems of α-conversion and all the tedious programming involved. However, this problem is now at a more superficial level in the sense that it is with 'pretty-printing labels' rather than the core underlying representation (which is FM, nameless, and correct).

Would these programs run faster? Not necessarily. The FM-language will still use an underlying representation of elements of a datatype, probably similar to those manipulated directly in current systems. It might (or might not) be that compiled FM code would not be very different from what we have now. Compared to a name-carrying technique an FM-version would presumably be more efficient because the underlying datatype could be left exposed (not wrapped in any computationally expensive abstract datatype).[105]

I suspect that programmers have got used to not being able to do certain things and that FM languages will generate their own demand in ways I could not necessarily foresee now.

Now we leave programming and turn to theory. For most researchers syntax is a means to an end. They usually use name-carrying terms ([**53**]) or name-carrying terms up to α-equivalence ([**7**]) and work with representatives. Up to a point the two options are equivalent because in the former technique the authors rename bound variables. In both techniques proofs tend to be by induction on the length of terms.

[105]Thanks to John Harrison ([**30**]) and Konrad Slind ([**31**]) for educating me about HOL-light and HOL.

Besides being a nuisance these proofs have been known to wrong-foot authors, though I believe today's generation has learnt some caution in this regard. For example [**76**, p.161-166] is a proof of a substitution lemma which is really by induction on length but claimed to be by structural induction.

If a paper has one particular object of study, e.g. 'bisimulation' in [**53**], the authors just have to prove (by induction on length of terms) that their object of interest is appropriately '***blind***' to renaming bound variables, at which point they can continue their development essentially by structural recursion/induction, renaming where necessary. For an example see [**53**, §3.1, p.19], just after Theorem 1,

> " ... we shall freely identify alpha-convertible agents ... ".

If the paper studies many different objects or a whole class of them, things are harder. For example Chapter IV involves the relations $\lhd$, $\lhd^\circ$, $\leq_{\mathbf{kl}}$, $\leq^\circ_{\mathbf{kl}}$, $\lhd_{\mathbf{ctx}}$ and $\lhd^*$ (plus spear-carriers). Without FM we would presumably have had to prove 'blindness' for each.

For such researchers FM provides the FM-style logic. One such was used in this thesis in Chapter II to construct FM set theory and in Chapter IV to prove an operational extensionality result T29.25 for an example language (itself an FM-style programming language). Some of its features are sketched in §5.1. A summary is:

- We have the option of treating terms as name-carrying or nameless as convenient. From the former we have all the benefits of induction on terms and writing terms with explicit names. From the latter we have the option of writing terms without names (for when we do no care what they are), and a guarantee we can never violate α-equivalence. We discuss this during §12.1 and the second half of §12.2.
- The logic has excellent properties which justify reasoning steps whose validity might not otherwise be obvious (e.g. R23.1.13).
- Reasoning using FM logic looks and feels exactly like what practitioners do anyway (once they have proved their technical 'blindness' results, see above). In the second half of §12.2 I show how we can phrase FM rules in a traditional way. All of Chapter IV is carried out in FM and it looks completely standard—except for when I occasionally take advantage of the extra power of FM to short-circuit what would otherwise be a few pages of mathematics (cf. §12.5, in particular R12.5.1).

33.5. McKinna and Pollack. Still on the subject of name-carrying terms, one approach that deserves special mention is the work of McKinna and Pollack,

see [**74**] (slides of a talk) and [**47**]. They pursue name-carrying terms in ZF to their logical conclusion. In [**47**] they present, by working through a particular example, a general methodology for carrying out structural induction on terms up to α-equivalence.

The trick that makes it work is an echo of T9.4.6 (duality of $\mathsf{И}$ between $\forall$ and $\exists$). They prove it by induction on term length for each individual datatype. Their methodology is a serious practical tool: in conversation, Pollack told me he had used his method to verify a typechecker in LEGO, a development of 6000 lemmas which took one-and-a-half years to carry out. See [**69**] (there a figure of 3000 is quoted, I presume it has grown since 1995).

The unique property of McKinna's and Pollack's work among non-FM approaches is that, like FM, they try to remain faithful to how we really seem to handle syntax with binding in the real world. The fact that they come back from this analysis with structures reminiscent of FM vindicates this document, and indeed this document vindicates theirs.

33.6. Fiore, Plotkin and Turi. Fiore, Plotkin and Turi have done work on presheaf models of abstract syntax with binders. See [**17**]. On *p.*234 (just after the itemised list) I comment that presheaves have such rich structure, you can make them do anything you like if you are clever enough. The challenge is to find a presheaf with a good model of syntax with binding. This is what [**17**] tries to do.

On the subject of presheaves, there is a mathematical presentation of FM in presheaf style, see Gabbay and Pitts [**18**, §6], which I do not discuss in this thesis. In fact, I made a conscious decision to present my material in the 'lightest' style I can and it seemed to me that presheaves are 'heavier' than sets. In [**17**] Fiore et al present their material in a relatively sophisticated mathematical style and this may generate the impression that it is completely different from the material in this thesis. However, the two approaches are quite similar. Many constructions in both correspond. Their δ operator ([**17**, p.196]) corresponds in its behaviour and construction to my $[\mathbb{A}](-)$ (D9.6.1). I owe the general idea of binding signatures ([**17**, §2]) to them. Their treatment of substitution ([**17**, §3]) is more abstract than mine but amounts to much the same object as I defined in D12.6.2.

I suggest that my underlying universe is nicer than theirs, in particular FM logic is classical whereas the internal logic of their category is not. They also do not have $\mathsf{И}$.

33.7. The axiomatic approach. It deserves mention that various authors have tried 'axiomatising' α-equivalence in one way or another. The authors do not try to 'explain' binding in terms of this axiomatisation. Either:

1. they want to systematically analyse the behaviour of binding, or
2. they are in COQ (or some similar system), want to make some definition work, and find they need this or that axiom to do so.

Gordon and Melham's [**21**] is a good example of case 1 above, carried out in the context of HOL ([**28**]). We can see how they do not 'explain' binding because they do not 'explain' renaming of bound variables, they just avoid it; see their §1.2. Wahid Taha brought to my attention some work, see [**16**], by Pašalić, Sheard and Taha, where they try to capture binding behaviour in the framework of a λ-calculus. There are several examples of case 2 above, in particular Honsell, Miculan and Sagnetto do precisely this in [**34**] as already commented on in Item 2 of §33.2.

I should also mention a relatively old paper by Stoughton, [**75**] (I get the impression it was rather ahead of its time). This axiomatises simultaneous substitution. Miller has tried extending standard ML with abstraction types to achieve the same effect as FreshML, see [**50**]. As in case 2 above, the idea is not to explain anything, just to get things done.

34. Accomplishments of this thesis

1. I define and develop the theory of a set-universe FM (Chapter II). It enables purely inductive definitions of datatypes of syntax with α-conversion by standard initial algebra methods, so the datatypes have standard recursive/iterative/inductive principles.

2. This universe is elementary in that it *looks* just like ZF. In particular it does not force its (considerable) extra power on practitioners (e.g. through unfamiliar notation, strange ways of doing things, etc.) if they do not want to use it.

3. The new set-universe and its associated logic accurately and simply capture our natural intuitions about syntax with binding.

4. Changing the subject, I present an alpha-stage Isabelle implementation of FM set theory, called Isabelle/FM (Chapter III).

5. I include a discussion of the various monsters I slayed in the course of the implementation. Some of them are pure proof-engineering and have nothing to do with FM as such. Some concern how FM interacted with mechanised proof.

6. Isabelle/FM includes nearly all the theory of Isabelle/ZF (as well as the extra FM structure), including facilities for declaring datatypes of syntax—only this time, thanks to the extra structure, with binding.

7. I have conducted several experiments defining datatypes of syntax up to binding and explored functions and predicates defined on them by their inductive structure. From all this I come to definite conclusions and proposals about the design of a beta-version of Isabelle/FM, and FM-style mechanised mathematics in general (§20).

8. Changing the subject once more, I demonstrate how an FM-style programming language might look (§4) and rigorously define a toy programming language as a first step on this path (Chapter IV).

9. I demonstrate the behaviour of this toy language (§22.2, §24.2) and show how it allows us to program inventing fresh names for bound variables at our convenience. It all works out very naturally.

10. Chapter IV proves a technical correctness result (T21.9) designed to reassure us that we have understood the subtle points mentioned above.

11. This technical correctness result is carried out using FM-set theory and logic. FM really does make the proofs easier and cleaner.

35. Future work

1. Continue the investigation of FM underlying universes by pursuing FM sets further and developing FM-HOL and an FM dependently typed universe (à la Martin-Löf).
2. Bring Isabelle/FM to beta-version so people could use it (§20). Possibly implement other mechanised versions of FM such as FM-HOL.
3. Develop the underlying theory for (and implement a version of) the FM-style programming language which this document nibbles at and lays the technical groundwork for (based on ML, a project has been applied for).
4. Further demonstrate the advantages of FM for paper mathematics by carrying out real mathematics using FM.
5. Lots more.

And that, gentle reader, was my thesis.

Murdoch J. Gabbay
10 August 2000
Cambridge University, England

— The End —

List of Figures

Bibliography

1. H. P. Barendregt, *The lambda calculus: Its syntax and semantics (revised ed.)*, Studies in Logic and the Foundations of Mathematics, vol. 103, North-Holland, Amsterdam, 1984.
2. Richard Bird and Ross Paterson, *De Bruijn notation as a nested datatype*, Journal of Functional Programming **9** (1999), no. 1, 77–91.
3. Francis Borceux, *Handbook of categorical algebra I,II,III*, CUP, 1994.
4. Francois Bry and Rainer Manthey, *Satchmo: A theorem prover implemented in prolog*, 9th Int. Conf. on Automated Deduction (CADE), Argonne, IL, Springer-Verlag LNCS 310, May 1988, pp. 415–434.
5. Juanito Camilleri and Tom Melham, *Reasoning with inductively defined relations in the hol theorem prover*, Tech. Report 265, University of Cambridge Computer Laboratory, August 1992.
6. Luca Cardelli and Andrew D. Gordon, *Logical properties of name restriction (not yet published)*, Microsoft Research, Cambridge.
7. G. L. Cattani and P. Sewell, *Models for name-passing processes: Interleaving and causal (extended abstract)*, Fifteenth Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, Washington, 2000.
8. A. Church, *A formulation of the simple theory of types*, Journal of Symbolic Logic **5** (1940), 56–68.
9. Thierry Coquand and Gérard Huet, *The Calculus of Constructions*, Information and Computation **76** (1988), no. 2/3, 95–120.
10. H. B. Curry and R. Feys, *Combinatory logic*, vol. 1, North Holland, 1958, (Second edition, 1968).
11. N. G. de Bruijn, *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*, Indag. Math. **34** (1972), 381–392.
12. J. Despeyroux, A. Felty, and A. Hirschowitz, *Higher-order abstract syntax in Coq*, Typed Lambda Calculus and Applications, 2nd International Conference (M. Dezani-Ciancaglini and G. D. Plotkin, eds.), Lecture Notes in Computer Science, vol. 902, Springer-Verlag, Berlin, 1995, pp. 124–138.
13. J. Despeyroux, F. Pfenning, and C. Schürmann, *Primitive recursion for higher-order abstract syntax*, Technical Report CMU-CS-96-172, Carnagie Mellon University, September 1996.
14. Joëlle Despeyroux and André Hirschowitz, *Higher-order abstract syntax with induction in Coq*, Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning (Kiev, Ukraine) (Frank Pfenning, ed.), Springer-Verlag LNAI 822, July 1994, pp. 159–173.
15. Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann, *Primitive recursion for higher-order abstract syntax*, Proceedings of the Third International Conference on Typed Lambda

Calculus and Applications (TLCA'97) (Nancy, France) (R. Hindley, ed.), Springer-Verlag LNCS, April 1997, An extended version is [**13**], pp. 147–163.

16. Walid Taha Emir Pasalic, Tim Sheard, *An untyped cbv operational semantics and equational theory of datatypes with binders (technical development)*, Tech. Report CSE-00-007, OGI Computer Science and Engineering, 1999.
17. M. P. Fiore, G. D. Plotkin, and D. Turi, *Abstract syntax and variable binding*, 14th Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, Washington, 1999, pp. 193–202.
18. M. J. Gabbay and A. M. Pitts, *A new approach to abstract syntax involving binders*, 14th Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, Washington, 1999, pp. 214–224.
19. Philippa Gardner, *Representing logics in type theory*, Ph.D. thesis, University of Edinburgh, July 1992, Available as Technical Report CST-93-92.
20. J. Goguen, J. Thatcher, , and E. Wagner, *An initial algebra approach to the specification, correctness and implementation of abstract data types*, Current Trends in Programming Methodology **IV** (1978), 80–149.
21. A. D. Gordon and T. Melham, *Five axioms of alpha-conversion*, Theorem Proving in Higher Order Logics: 9th Interational Conference, TPHOLs'96, Lecture Notes in Computer Science, vol. 1125, Springer-Verlag, Berlin, 1996, pp. 173–191.
22. C. A. Gunter, *Semantics of programming languages: Structures and techniques*, Foundations of Computing, MIT Press, 1992.
23. P. R. Halmos, *Naïve set theory*, Springer Verlag, March 1987.
24. D. Hirschkoff, *Mise en œuvre de preuves de bisimulation*, Ph.D. thesis, École Nationale des Ponts et Chaussées, January 1999, in French.
25. Daniel Hirschkoff, *A full formalization of pi-calculus theory in the Calculus of Constructions*, Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97) (Murray Hill, New Jersey) (E. Gunter and A. Felty, eds.), August 1997, pp. 153–169.
26. M. Hofmann, *Semantical analysis of higher-order abstract syntax*, 14th Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, Washington, 1999, pp. 204–213.
27. COQ Homepage, `http://coq.inria.fr/`.
28. HOL Homepage, `http://www.cl.cam.ac.uk/Research/HVG/HOL/HOL.html#index`.
29. HOL-Light Homepage, `http://www.cl.cam.ac.uk/users/jrh/hol-light/index.html`.
30. John Harrison Homepage, `http://www.cl.cam.ac.uk/users/jrh/`.
31. Konrad Slind Homepage, `http://www.cl.cam.ac.uk/users/kxs/`.
32. Tobias Nipkow Homepage, `http://www4.informatik.tu-muenchen.de/~nipkow/`.
33. Twelf Homepage, `http://www.cs.cmu.edu/~twelf`.
34. F. Honsell, M. Miculan, and I. Scagnetto, *π-calculus in (co)inductive type theory*, Tech. report, Dipartimento di Matematica e Informatica, Università degli Studi di Udine, 1998.
35. D. J. Howe, *Proving congruence of bisimulation in functional programming languages*, Information and Computation **124** (1996), no. 2, 103–112.

36. Douglas J. Howe, *Equality in lazy computation systems*, Proceedings, Fourth Annual Symposium on Logic in Computer Science (Asilomar Conference Center, Pacific Grove, California), IEEE Computer Society Press, 5–8 June 1989, pp. 198–203.
37. Isabelle Cambridge Homepage, `http://www.cl.cam.ac.uk/research/hvg/isabelle/cambridge.html`.
38. T. J. Jech, *About the axiom of choice*, Handbook of Mathematical Logic (J. Barwise, ed.), North-Holland, 1977, pp. 345–370.
39. T. J. Jech, *Set theory*, Academic Press, 1978.
40. P. T. Johnstone, *Notes on logic and set theory*, CUP Mathematical Textbooks, 1987.
41. L. Lamport and L. C. Paulson, *Should your specification language be typed?*, Tech. Report 147, Digital SRC, 1998.
42. Søren Bøgh Lassen, *Relational reasoning about functions and nondeterminism*, Ph.D. thesis, Department of Computer Science, University of Aarhus, 1998.
43. Saunders Mac Lane, *Categories for the working mathematician*, Springer, 1971.
44. Raymond McDowell, *Reasoning in a logic with definitions and induction*, Ph.D. thesis, University of Pennsylvania, 1997.
45. Raymond McDowell and Dale Miller, *A logic for reasoning with higher-order abstract syntax*, 12th Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, Washington, 1997, pp. 434–445.
46. ———, *A logic for reasoning with higher-order abstract syntax: An extended abstract*, Proceedings of the Twelfth Annual Symposium on Logic in Computer Science (Warsaw, Poland) (Glynn Winskel, ed.), June 1997, pp. 434–445.
47. James McKinna and Robert Pollack, *Some lambda calculus and type theory formalized*, Tech. report, 1997.
48. Thomas F. Melham, *Using recursive types to reason about hardware in higher-order logic*, Tech. Report TR135, Cambridge University Computer Lab, April 1990.
49. Marino Miculan, *Encoding logical theories of programs*, Ph.D. thesis, Università di Pisa-Genova-Udine, Thesis TD-7/97, March 1997.
50. D. Miller, *An extension to ML to handle bound variables in data structures: Preliminary report*, Proceedings of the Logical Frameworks BRA Workshop, 1990.
51. Dale Miller, *Abstract syntax for variable binders*, CL2000, Springer-Verlag, 2000, To appear.
52. Dale Miller and Catuscia Palamidessi, *Foundational aspects of syntax*, Computing Surveys **31** (1999).
53. R. Milner, J. Parrow, and D. Walker, *A calculus of mobile processes (parts I and II)*, Information and Computation **100** (1992), 1–77.
54. R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The definition of standard ML (revised)*, MIT Press, 1997.
55. J. C. Mitchell and G. D. Plotkin, *Abstract types have existential types*, ACM Transactions on Programming Languages and Systems **10** (1988), 470–502.
56. Logical Frameworks Page, `http://www.cs.cmu.edu/~fp/lfs.html`.
57. L. C. Paulson, *ML for the working programmer, 2nd ed.*, Cambridge University Press, 1996.
58. Lawrence C. Paulson, *Isabelle reference manual, Isabelle99 edition*, Part of the Isabelle Distribution (see [**37**]).

59. ———, *Isabelle's logics: FOL and ZF, Isabelle99 edition*, Previously "Isabelle's Object-Logics", Isabelle98-1 Edition. Part of the Isabelle Distribution (see [**37**]).
60. ———, *A fixedpoint approach to implementing (co)inductive definitions*, Proceedings of the 12th International Conference on Automated Deduction (Nancy, France) (Alan Bundy, ed.), Springer-Verlag LNAI 814, June 1994, pp. 148–161.
61. Simon Peyton-Jones, `http://research.microsoft.com/~simonpj/`.
62. Frank Pfenning, *The practice of logical frameworks*, Proceedings of the Colloquium on Trees in Algebra and Programming (Linköping, Sweden) (Hélène Kirchner, ed.), Springer-Verlag LNCS 1059, April 1996, Invited talk, pp. 119–134.
63. A. M. Pitts, *A note on logical relations between semantics and syntax*, Logic Journal of the Interest Group in Pure and Applied Logics **5** (1997), no. 4, 589–601.
64. ———, *Operationally-based theories of program equivalence*, Semantics and Logics of Computation (P. Dybjer and A. M. Pitts, eds.), Publications of the Newton Institute, Cambridge University Press, 1997, pp. 241–298.
65. ———, *Parametric polymorphism and operational equivalence*, Mathematical Structures in Computer Science **10** (2000), 1–39.
66. A. M. Pitts and M. J. Gabbay, *A metalanguage for programming with bound names modulo renaming*, Mathematics of Program Construction, MPC2000, Proceedings, Ponte de Lima, Portugal, July 2000 (R. Backhouse and J. N. Oliveira, eds.), Lecture Notes in Computer Science, vol. ?, Springer-Verlag, Heidelberg, 2000, pp. ?–?
67. A. M. Pitts and I. D. B. Stark, *Observable properties of higher order functions that dynamically create local names, or: What's new?*, Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993, Lecture Notes in Computer Science, vol. 711, Springer-Verlag, Berlin, 1993, pp. 122–141.
68. Andrew M. Pitts, `http://www.cl.cam.ac.uk/~ap`.
69. R. Pollack, *A verified typechecker*, Proceedings of the Second International Conference on Typed Lambda Calculi and Applications (M. Dezani-Ciancaglini and G. Plotkin, eds.), TLCA'95, 1995.
70. Robert Pollack, `http://www.dcs.ed.ac.uk/home/rap/`.
71. Bali Project, `http://www4.informatik.tu-muenchen.de/~isabelle/bali/`.
72. Willard Van Orman Quine, *Set theory and its logic, revised ed.*, Harvard University Press, Cambridge, MA, 1963.
73. Logical Frameworks Researchers, `http://www.cs.cmu.edu/~fp/lfs-people.html`.
74. Robert and James McKinna, *Names, binding and substitution*, February 1998, Unpublished slides of a talk, available as `bindingTalk.ps` via [**70**].
75. A. Stoughton, *Substitution revisited*, Theoretical Computer Science **59** (1988), no. 3, 317–325.
76. Joseph E. Stoy, *Denotational semantics: The Scott-Strachey approach to programming language theory*, The MIT Press, 1977.
77. Benjamin Werner, *Une théorie des constructions inductives*, Ph.D. thesis, Université de Paris VII, 1994.
78. Glynn Winskel, *The formal semantics of programming languages: An introduction*, MIT Press, 1993.