Appears in *Computer Science Logic: Proceedings of the 18th International Workshop CSL 2004, Karpacz, Poland, September 20–24, 2004*, Lecture Notes in Computer Science 3210, pages 235–249. Springer-Verlag, 2004.

A Dependent Type Theory with Names and Binding

Ulrich Schöpp and Ian Stark

LFCS, School of Informatics, University of Edinburgh
JCMB, King's Buildings, Edinburgh EH9 3JZ

Abstract. We consider the problem of providing formal support for working with abstract syntax involving variable binders. Gabbay and Pitts have shown in their work on Fraenkel-Mostowski (FM) set theory how to address this through first-class names: in this paper we present a dependent type theory for programming and reasoning with such names. Our development is based on a categorical axiomatisation of names, with freshness as its central notion. An associated adjunction captures constructions known from FM theory: the freshness quantifier И, name-binding, and unique choice of fresh names. The Schanuel topos — the category underlying FM set theory — is an instance of this axiomatisation. Working from the categorical structure, we define a dependent type theory which it models. This uses bunches to integrate the monoidal structure corresponding to freshness, from which we define novel multiplicative dependent products Π^* and sums Σ^*, as well as a propositions-as-types generalisation H of the freshness quantifier.

1 Introduction

The handling of variable binding in abstract syntax is a recognised challenge for machine-assisted reasoning about programming languages and logics. The problem is that a significant part of the formalisation effort may go into dealing with issues that are normally suppressed in informal practice: namely that one is working with α-equivalence classes of terms rather than raw terms.

Gabbay and Pitts have shown that FM set theory supports a notion of names that can make precise the informal practise of using concrete names for α-equivalence classes. They give a number of useful constructions: abstract syntax with binders can be encoded as an inductive data type, there is a useful syntax-independent notion of name-freshness, and a freshness quantifier simplifies reasoning with names.

The approach of Gabbay and Pitts has been studied in a number of other settings, among which are the first-order Nominal Logic [18], the higher-order logic FM-HOL [6] as well as the programming language FreshML [19]. Related [9] to FM theory, the Theory of Contexts [11] provides an axiomatisation of reasoning with names in dependent type theory. The ideas underlying FM have also proved useful in other areas such as Spatial Logic [2] or programming with semi-structured data with hidden labels [1]. These approaches typically focus either on programming with names, or reasoning about them. The Theory of Contexts, for example, supports reasoning with names, but does not admit functions that compare names or which (locally) choose fresh names.

In this paper we take the first steps towards a dependent type theory incorporating FM concepts for both programming and reasoning with names. We introduce a dependent type theory, using as guidance the categorical structure of Schanuel topos, which is the category corresponding to FM set theory. In contrast to FM set theory, where swapping is the primitive notion for working with names, we take freshness as the central primitive of our type theory. This allows us to describe the constructions with names and binding in terms of universal constructions, and also avoids problems with extensional equality, which seems to be necessary for defining α-equivalence classes using swapping.

As the first contribution of the paper we introduce a *bunched* dependent type theory. Since freshness corresponds to a monoidal structure, bunches provide a natural way of integrating it into the type theory. Our bunched type theory may be seen as a generalisation of the $\alpha\lambda$-calculus of O'Hearn and Pym [17,20]. The $\alpha\lambda$-calculus is a simple type theory corresponding to a category which is both cartesian closed and monoidal closed. Our type theory extends this situation, but only in the additive direction: we consider a category which is *locally* cartesian closed as well as monoidal closed. In this structure, we can model a dependent type theory with two function spaces $\Pi x{:}A.\,B$ and $\Pi^* x{:}C.\,D$. The first comes from the locally cartesian closed structure and consists of normal dependent functions. The second, which is subject to the restriction that C is closed, comes from the monoidal closed structure and may be thought of as consisting of functions which are only defined on arguments $x : C$ that contain just *fresh* names. In particular, with a type of names $\mathbf{N}$, we can use $\Pi^* n{:}\mathbf{N}.\,D$ to model α-equivalence classes, which corresponds to the well-known approach of modelling α-equivalence classes as 'fresh functions' [7,4,9,5]. Another way of representing α-equivalence classes, as given in [7], is to consider them as pairs $n.x$ of a term x with a distinguished name n in such a way that the identity of n is hidden in the pair. This representation is also available in our type theory as fresh sum types Σ^*, dual to Π^*. The inhabitants of $\Sigma^* x{:}C.\,D$ may be thought of as pairs $M.N$ where $M : C$ and $N : D(M)$ and in which all the names in M have been hidden. To formulate Σ^*-types, we introduce a type $B^{*(M:A)}$, thought of as those elements of B which are free from all the names in the term $M : A$. These *freefrom* types are used to enforce that no use of a pair $M.N$ in $\Sigma^* x{:}C.\,D$ can reveal the hidden names.

As a second contribution of the paper, we give a new categorical axiomatisation of names and binding. The main feature of this axiomatisation is a propositions as types generalisation of the freshness quantifier of Gabbay and Pitts. To recall the freshness quantifier, consider quantifiers $\exists^* x{:}A.\,\varphi$ and $\forall^* x{:}A.\,\varphi$ expressing 'φ holds for some x containing only *fresh* names' and 'φ holds for any x containing only *fresh* names' respectively. The freshness quantifier И arises because, for the type of names $\mathbf{N}$, the propositions $\exists^* n{:}\mathbf{N}.\,\varphi$ and $\forall^* n{:}\mathbf{N}.\,\varphi$ are equivalent; and И $n.\,\varphi$ is used to denote either of them. We have a propositions-as-types correspondence between $\exists^*$ and Σ^* as well as between $\forall^*$ and Π^*, so one may generalise the equivalence of $\exists^* n : \mathbf{N}.\,\varphi$ and $\forall^* n : \mathbf{N}.\,\varphi$ to an isomorphism between $\Sigma^* n{:}\mathbf{N}.\,D$ and $\Pi^* n{:}\mathbf{N}.\,D$.

This motivates our categorical axiomatisation of names. The central concept is freshness, giving rise to a certain 'fresh weakening' functor W. The types Σ^* and Π^* are left and right adjoints to W. Names are given by an object $\mathbf{N}$ having decidable equality. Moreover, we require an isomorphism $\Sigma^*_{\mathbf{N}} \cong \Pi^*_{\mathbf{N}}$ generalising the freshness quantifier. We show that this structure includes not only the freshness quantifier, but also binding $(n.x)$ as in [7,16] as well as unique choice of fresh names $(\mathrm{new}\ n.\, M)$ as in FreshML [19].

The semantics leads us to a type theory with names and binding. Based on the isomorphism $\Sigma^*_{\mathbf{N}} \cong \Pi^*_{\mathbf{N}}$, we introduce hidden-name types $\mathrm{H}n.\, D$ as a generalisation of the freshness quantifier. We may think of the elements of $\mathrm{H}n.\, D$ as elements of $\Sigma^* n{:}\mathbf{N}.\, D$, i.e. pairs with hidden names, but also as elements of $\Pi^* n{:}\mathbf{N}.\, D$, i.e. functions taking only fresh names. In analogy to the freshness quantifier, which has the rules from both $\exists^*$ and $\forall^*$, the rules for H are those from both Σ^* and Π^*. This dual view of hidden-name types turns out to be useful for working with abstract syntax: it allows us to use both HOAS-style constructions and FM-style constructions at the same time.

2 A Bunched Dependent Type Theory

In this section we introduce a first-order bunched dependent type theory and identify the categorical structure corresponding to it. The type theory has the following forms of sequents: $(\vdash\ \Gamma\ \mathrm{Bunch})$ — Γ is a bunch, or context; $(\Gamma\ \vdash\ A\ \mathrm{Type})$ — A is a type in context Γ; $(\Gamma\ \vdash\ M\ :\ A)$ — M is a term of type A in context Γ; as well as corresponding sequents for definitional equalities.

2.1 Bunches and Structural Rules

Bunches are built from the empty bunch $\Diamond$ using two kinds of extension. First, the familiar additive context extension from dependent type theory, which takes a bunch Γ to the bunch $\Gamma,\ x : A$. Second, a multiplicative extension taking two bunches Γ and Δ to a new bunch $\Gamma * \Delta$. This extension is non-dependent in that no dependency is allowed across the $*$. The bunch $\Gamma * \Delta$ should be thought of as the context Γ, Δ with the restriction that the names occurring in Γ are disjoint from those in Δ. For example, if Lam is a type which encodes object-level λ-terms, then the bunch $(x{:}\mathsf{Lam}, y{:}\mathsf{Lam}) * (z{:}\mathsf{Lam})$ declares three terms x, y and z with the property that the names (representing the free variables of the encoded terms) in x and y are disjoint from those in z.

$$\frac{}{\vdash \Diamond\ \mathrm{Bunch}} \qquad \frac{\Gamma \vdash A\ \mathrm{Type}}{\vdash \Gamma,\ x{:}A\ \mathrm{Bunch}}\ x \notin v(\Gamma) \qquad \frac{\vdash \Gamma\ \mathrm{Bunch} \quad \vdash \Delta\ \mathrm{Bunch}}{\vdash \Gamma * \Delta\ \mathrm{Bunch}}\ v(\Gamma) \cap v(\Delta) = \emptyset$$

In the side condition of these rules, we write $v(\Gamma)$ for the set of variables declared in Γ. We will frequently omit such side-conditions on the variable names, assuming tacitly that we encounter only bunches in which no variable is declared more than once.

We use the notation $\Gamma(\Delta)$ to indicate that Γ has a sub-bunch Δ, where sub-bunches are defined as follows: Δ is a sub-bunch of itself, and if Δ is a sub-bunch of Γ then it is also a subbunch of $(\Gamma,\ x : A)$, and $\Gamma * \Phi$, and $\Phi * \Gamma$. We write $\Gamma(\Phi)$ for the bunch which results from $\Gamma(\Delta)$ by replacing the (unique) occurrence of Δ in Γ with Φ.

Using this notation, we can formulate the structural rules:

$$(\mathrm{Proj})\ \frac{\Gamma \vdash A\ \mathrm{Type}}{\Gamma,\ x{:}A \vdash x : A}\ x \notin v(\Gamma)$$

$$\text{(Weak)}\ \frac{\mathit{\Gamma}(\mathit{\Delta}) \vdash \mathcal{J} \quad \mathit{\Delta} \vdash A \ \mathrm{Type}}{\mathit{\Gamma}(\mathit{\Delta},\, x\!:\!A) \vdash \mathcal{J}}\ x \notin v(\mathit{\Gamma}, \mathit{\Delta}) \qquad \text{(Subst)}\ \frac{\mathit{\Delta} \vdash M : A \qquad \mathit{\Gamma}(\mathit{\Delta},\, x\!:\!A) \vdash \mathcal{J}}{\mathit{\Gamma}(\mathit{\Delta})\,[M/x] \vdash \mathcal{J}\,[M/x]}$$

$$\text{(Unit)}\ \frac{\mathit{\Gamma}(\mathit{\Delta}) \vdash \mathcal{J}}{\mathit{\Gamma}(\mathit{\Delta} * \lozenge) \vdash \mathcal{J}} \qquad \text{(Swap)}\ \frac{\mathit{\Gamma}(\mathit{\Delta} * \mathit{\Phi}) \vdash \mathcal{J}}{\mathit{\Gamma}(\mathit{\Phi} * \mathit{\Delta}) \vdash \mathcal{J}} \qquad \text{(Assoc)}\ \frac{\mathit{\Gamma}((\mathit{\Delta} * \mathit{\Phi}) * \mathit{\Psi}) \vdash \mathcal{J}}{\mathit{\Gamma}(\mathit{\Delta} * (\mathit{\Phi} * \mathit{\Psi})) \vdash \mathcal{J}}$$

In these rules, we use $\mathcal{J}$ for an arbitrary judgement and double lines for bi-directional rules. We highlight the rule (Unit) which requires the empty bunch $\lozenge$ to be a unit for $*$, thus making $*$ affine. In particular, the multiplicative weakening rule

$$(*\text{-Weak})\ \frac{\mathit{\Gamma}(\mathit{\Delta}) \vdash \mathcal{J} \qquad \vdash \mathit{\Gamma}(\mathit{\Delta} * \mathit{\Phi})\ \mathrm{Bunch}}{\mathit{\Gamma}(\mathit{\Delta} * \mathit{\Phi}) \vdash \mathcal{J}}$$

becomes admissible by using (Unit) together with (Weak).

Semantically, the bunches and structural rules can be modelled by a comprehension category [12] that in addition has an affine (i.e. the unit is isomorphic to the terminal object) symmetric monoidal structure $*$ in its base. We model the additive context-extension $\mathit{\Gamma},\, x\!:\!A$ by the comprehension, and the multiplicative context-extension $\mathit{\Gamma} * \mathit{\Delta}$ by the monoidal product. To simplify the development, we make an additional assumption on the monoidal structure, given by the following definition [10].

Definition 1. *An* affine linear category *is a category $\mathbb{B}$ with finite products and an affine symmetric monoidal structure $*$ such that, for any two objects A and B of $\mathbb{B}$, the canonical map $\langle \pi_1, \pi_2 \rangle : A * B \to A \times B$ is a monomorphism.*

In most of the paper, we take a special comprehension category: the codomain fibration $cod : \mathbb{B}^{\to} \to \mathbb{B}$ for an affine linear category $\mathbb{B}$ having all pullbacks. Although technically the interpretation uses a corresponding split fibration to deal with well-known coherence issues [8], in the following we elide such details. We assume the reader to be familiar with the semantics of (first-order) dependent type theory, see e.g. [12,22,21].

2.2 Type Formers

In this section, we consider the types and terms, motivating them semantically. Starting from a codomain fibration $cod : \mathbb{B}^{\to} \to \mathbb{B}$ with an affine linear base $\mathbb{B}$, we step-by-step add more structure and introduce syntax based on it.

Type and term constants. Basic types and terms are given by constants. These can be formulated as usual. For example, a type constant T in context $\mathit{\Gamma}$ may be introduced as $(\mathit{\Gamma} \vdash T(\boldsymbol{x})\ \mathrm{Type})$, where $\boldsymbol{x}$ is the list of variables defined in $\mathit{\Gamma}$. That it is enough to annotate the constants just with the list of variables in $\mathit{\Gamma}$, ignoring any bunching structure, is a consequence of the assumption that the canonical map $A * B \rightarrowtail A \times B$ is a monomorphism.

Additive types (Σ, Π). Types found in Martin-Löf type theory can also be formulated as usual. In this paper, we use dependent sums and products, but others such as identity types can be added without problem. To model Π-types in the codomain fibration, we assume $\mathbb{B}$ to be locally cartesian closed [21,12].

Monoidal product (*). We add types $A{*}B$ which internalise the context multiplication $\Gamma * \Delta$. The type $A{*}B$ may be thought of as containing all pairs $\langle M, N\rangle$ in $A{\times}B$ for which the sets of names underlying M and N are disjoint.

$$\text{($*$-Ty)}\ \frac{\vdash A\ \mathrm{Type} \qquad \vdash B\ \mathrm{Type}}{\vdash A{*}B\ \mathrm{Type}} \qquad \text{($*$-I)}\ \frac{\vdash A{*}B\ \mathrm{Type} \qquad \Gamma \vdash M : A \qquad \Delta \vdash N : B}{\Gamma * \Delta \vdash M{*}N : A{*}B}$$

$$\text{($*$-E)}\ \frac{\Gamma(z{:}A{*}B) \vdash C\ \mathrm{Type} \qquad \Delta \vdash M : A{*}B \qquad \Gamma(x{:}A * y{:}B)\,[x{*}y/z] \vdash N : C\,[x{*}y/z]}{\Gamma(\Delta)\,[M/z] \vdash (\mathrm{let}\ M\ \mathrm{be}\ x{*}y\ \mathrm{in}\ N) : C\,[M/z]}$$

Note that the type $A{*}B$ requires both A and B to be closed. This is because of substitution, as $(A{*}B)[\sigma]$ and $(A[\sigma]{*}B[\sigma])$ would not always have isomorphic interpretations.

Since the rule ($*$-Weak) is admissible, we can derive an inclusion $\imath_{A,B}$ of type $A{*}B \to A{\times}B$, given by the term $\imath_{A,B} =_{\mathrm{df}} \lambda p : A{*}B.\,(\mathrm{let}\ p\ \mathrm{be}\ x{*}y\ \mathrm{in}\ \langle x, y\rangle)$. Using this, we can state the equations for the monoidal product:

$$\text{($*$-β)}\ \frac{\Gamma \vdash \mathrm{let}\ M{*}N\ \mathrm{be}\ x{*}y\ \mathrm{in}\ R : C}{\Gamma \vdash (\mathrm{let}\ M{*}N\ \mathrm{be}\ x{*}y\ \mathrm{in}\ R) = R\,[M/x]\,[N/y] : C}$$

$$\text{($*$-η)}\ \frac{\Delta \vdash M : A{*}B \qquad \Gamma(z{:}A{*}B) \vdash N : C}{\Gamma(\Delta)[M/z] \vdash N\,[M/z] = \mathrm{let}\ M\ \mathrm{be}\ x{*}y\ \mathrm{in}\ (N\,[x{*}y/z]) : C[M/z]}$$

$$\text{(Inject)}\ \frac{\Gamma \vdash M : A{*}B \qquad \Gamma \vdash N : A{*}B \qquad \Gamma \vdash \imath_{A,B}(M) = \imath_{A,B}(N) : A{\times}B}{\Gamma \vdash M = N : A{*}B}$$

Fresh dependent products (Π^*). We now make the further assumption on $\mathbb{B}$ that, for each object A in $\mathbb{B}$, the functor $-{*}A$ preserves pullbacks and has a right adjoint $A \mathrel{-\!\!*} -$.

This gives rise the following situation. Let $gl(- * A)$ be the fibration defined by change-of-base as in the left square below. Let $W_A : \mathbb{B}^{\to} \to \mathbb{B}/(- * A)$ be the functor which maps an object $f : B \to G$ to $f * A : B * A \to G * A$. The assumption that $- * A$ preserves pullbacks amounts to saying that W_A is a *fibred* functor from *cod* to $gl(-{*}A)$. Moreover, it follows that W_A has a fibred right adjoint $\Pi^*_A : \mathbb{B}/(- * A) \to \mathbb{B}^{\to}$, see e.g. [14]. Explicitly, Π^*_A maps an object $g : C \to G * A$ to the the morphism $\Pi^*_A g$ as in the pullback on the right.

$$\begin{tikzcd}
\mathbb{B}/(- * A) \arrow[r] \arrow[d, "gl(-*A)"'] \arrow[dr, phantom, very near start, "\lrcorner"] & \mathbb{B}^{\to} \arrow[d, "cod"] \\
\mathbb{B} \arrow[r, "-*A"'] & \mathbb{B}
\end{tikzcd}
\qquad
\begin{tikzcd}
\Pi^*_A C \arrow[r] \arrow[d, "\Pi^*_A g"'] \arrow[dr, phantom, very near start, "\lrcorner"] & A \mathrel{-\!\!*} C \arrow[d, "A \mathrel{-\!\!*} g"] \\
G \arrow[r, "\eta"'] & A \mathrel{-\!\!*} (G * A)
\end{tikzcd}$$

Proposition 1. *For any object A of $\mathbb{B}$, the functor W_A as defined above has a fibred right adjoint Π^*_A if and only if $A{*}-$ preserves pullbacks and has a right adjoint $A \mathrel{-\!\!*} -$.*

In this way, we can recast the monoidal closed structure in terms of a fibred adjunction, and introduce syntax for the fibred adjunction as follows.

$$\text{(Π^*-Ty)}\ \frac{\Gamma * x{:}A \vdash B\ \mathrm{Type}}{\Gamma \vdash \Pi^* x{:}A.\,B\ \mathrm{Type}}$$

$$(\Pi^*\text{-I})\,\frac{\mathit{\Gamma} * x{:}A \vdash M : B}{\mathit{\Gamma} \vdash \lambda^* x{:}A.\,M \;:\; \Pi^* x{:}A.\,B} \qquad (\Pi^*\text{-E})\,\frac{\mathit{\Gamma} \vdash M : \Pi^* x{:}A.\,B \qquad \mathit{\Delta} \vdash N : A}{\mathit{\Gamma} * \mathit{\Delta} \vdash M @ N : B\,[N/x]}$$

$$(\Pi^*\text{-}\beta)\,\frac{\mathit{\Gamma} * x{:}A \vdash M : B \qquad \mathit{\Delta} \vdash N : A}{\mathit{\Gamma} * \mathit{\Delta} \vdash (\lambda^* x{:}A.\,M) @ N = M\,[N/x] : B\,[N/x]}$$

$$(\Pi^*\text{-}\eta)\,\frac{\mathit{\Gamma} \vdash M : \Pi^* x{:}A.\,B}{\mathit{\Gamma} \vdash \lambda^* x{:}A.\,(M @ x) = M : \Pi^* x{:}A.\,B}$$

Notice that the fresh dependent product $\Pi^* x{:}A.\,B$ is only well-formed for closed types A, as bunching does not allow dependency across the $*$ in the bunch $\mathit{\Gamma} * x{:}A$.

The rules of Π^* derive from the adjoint correspondence

$$\frac{1_{G*A} = W_A(1_G) \to C \qquad \text{in } \mathbb{B}/(G * A)}{\overline{\;1_G \to \Pi^*_A(C) \qquad \text{in } \mathbb{B}/G\;}},$$

since morphisms $1_G \to D$ in $\mathbb{B}/G$ correspond to terms in context G. Here, 1_G denotes the terminal object in $\mathbb{B}/G$. That Π^*_A is a *fibred* right adjoint means that substitution behaves as expected, that is we have $(\Pi^* x{:}A.\,B)[M/y] = \Pi^* x{:}A.\,(B[M/y])$ as well as $(\lambda^* x{:}A.\,N)[M/y] = \lambda^* x{:}A.\,(N[M/y])$.

Freefrom types ($A^{*(N:B)}$). Having considered a fibred right adjoint Π^*_A to W_A, it is natural to ask for a fibred left adjoint Σ^*_A to W_A. To add syntax for such a left adjoint, we need to account for a one-to-one correspondence between maps $B \to W_A(C)$ in $\mathbb{B}/(G*A)$ and $\Sigma^*_A(B) \to C$ in $\mathbb{B}/G$. Hence, we need a syntactic equivalent for the map $B \to W_A(C)$, and so must introduce syntax for $W_A(C)$. Note that this is not necessary for Π^*, since there we only need the value of $W_A(1_G)$, which is 1_{G*A}.

We introduce types $B^{*(M:A)}$ as a syntax for working with $W_A(B)$. Intuitively, the type $B^{*(M:A)}$ comprises all those $p : B*A$ whose second component $\pi_2(p)$ is $M : A$. The functor W_A may then be understood as a 'fresh weakening' functor, taking the type $(\mathit{\Gamma} \vdash B \text{ Type})$ to $(\mathit{\Gamma} * x{:}A \vdash B^{*(x:A)} \text{ Type})$. Here, type A is necessarily closed, while B may in general depend on $\mathit{\Gamma}$. However, in the present paper we avoid the complexity of managing substitution in B by restricting to closed freefrom types:

$$(\text{F-Ty})\,\frac{\vdash A \text{ Type} \qquad \vdash B \text{ Type} \qquad \mathit{\Delta} \vdash N : A}{\mathit{\Delta} \vdash B^{*(N:A)} \text{ Type}}$$

$$(\text{F-I})\,\frac{\vdash A, B \text{ Type} \qquad \mathit{\Gamma} \vdash M : B \qquad \mathit{\Delta} \vdash N : A}{\mathit{\Gamma} * \mathit{\Delta} \vdash M^{*N} : B^{*(N:A)}}$$

$$(\text{F-E})\,\frac{\mathit{\Gamma}(x{:}A,\; z{:}B^{*(x:A)}) \vdash C \text{ Type} \quad \mathit{\Delta} \vdash M : B^{*(N:A)} \quad \mathit{\Gamma}(y{:}B * x{:}A) \vdash R : C[y^{*x}/z]}{\mathit{\Gamma}(\mathit{\Delta})[N/x][M/z] \vdash \text{let } M \text{ be } y^{*x} \text{ in } R : C[N/x][M/z]}$$

The equations[1], in which $\mathit{\Gamma} \vdash Q : B^{*(N:A)}$, are:

$$\begin{array}{ll} (\beta) & \text{let } M^{*N} \text{ be } y^{*x} \text{ in } R = R[N/x][M/y] \\ (\eta) & \text{let } Q \text{ be } y^{*x} \text{ in } R[y^{*x}/z] = R[N/x][Q/z] \end{array}$$

[1] For brevity, from now on, we omit the contexts and typeability assumptions in the formulation of equations. Nevertheless, all equations are to be understood as equations-in-context, formulated under suitable typeability assumptions.

Furthermore, we add a constant to 'join' two elements of freefrom types.

$$\text{(F-join)}\ \frac{\mathit{\Gamma} \vdash M : A^{*(R:C)} \qquad \mathit{\Gamma} \vdash N : B^{*(R:C)}}{\mathit{\Gamma} \vdash \mathrm{join}_{A,B,C}(M,N) : (A \times B)^{*(R:C)}}$$

This constant is part of the syntax for W_A, arising from the fact that W_A is a *fibred* functor, equivalently that $- * A$ preserves pullbacks. It makes available the important property of freshness that if two objects x and y are fresh for some z then so is the pair $\langle x, y\rangle$. The behaviour of join is described by the equations

$$\begin{aligned} &\mathrm{let\ join}_{A,B,C}(M,N)\ \mathrm{be}\ y^{*x}\ \mathrm{in}\ (\pi_1\ y)^{*x} = M,\\ &\mathrm{let\ join}_{A,B,C}(M,N)\ \mathrm{be}\ y^{*x}\ \mathrm{in}\ (\pi_2\ y)^{*x} = N. \end{aligned}$$

The semantic interpretation of (F-Ty) is given by the following diagram.

$$\begin{tikzcd} \bullet \arrow[r] \arrow[d, "B^{*(N:A)}"'] \arrow[dr, phantom, "\lrcorner", very near start] & B * A \arrow[r, equal] \arrow[d, "\pi_2"] \arrow[dr, phantom, "\lrcorner", very near start] & B * A \arrow[d, "! * A"] \\ \mathit{\Delta} \arrow[r, "N"'] & A \arrow[r, "\cong"'] & 1 * A \end{tikzcd}$$

To see how this corresponds to W_A, recall that a closed type B in context $\mathit{\Gamma}$ corresponds to the projection $\pi_B : \mathit{\Gamma} \times B \to \mathit{\Gamma}$. Using pullback-preservation of $- * A$, the following square is easily seen to be a pullback.

$$\begin{tikzcd} (\mathit{\Gamma} \times B) * A \arrow[r, "\pi_2 * A"] \arrow[d, "\pi_B * A"'] \arrow[dr, phantom, "\lrcorner", very near start] & B * A \arrow[d, "\pi_2"] \\ \mathit{\Gamma} * A \arrow[r, "\pi_2"'] & A \end{tikzcd}$$

Since the bottom row of this diagram corresponds to the term $\mathit{\Gamma} * x : A \vdash x : A$, this means that $(\mathit{\Gamma} * x : A \vdash B^{*(x:A)}\ \mathrm{Type})$ receives an interpretation isomorphic to $\pi_B * A$, which, by definition, is just $W_A(\pi_B)$.

Fresh dependent sums (Σ^*). We now assume that W_A has a fibred left adjoint Σ^*_A. Using freefrom types as syntax for W_A, this gives rise to the following rules for Σ^*_A.

$$(\Sigma^*\text{-Ty})\ \frac{\mathit{\Gamma} * x : A \vdash B\ \mathrm{Type}}{\mathit{\Gamma} \vdash \Sigma^* x{:}A.\, B\ \mathrm{Type}}$$

$$(\Sigma^*\text{-I})\ \frac{x : A \vdash B\ \mathrm{Type} \qquad \mathit{\Gamma} \vdash M : A \qquad \mathit{\Gamma} \vdash N : B[M/x]}{\mathit{\Gamma} \vdash \mathrm{bind}(M,N) : (\Sigma^* x{:}A.\, B)^{*(M:A)}}$$

$$(\Sigma^*\text{-E})\ \frac{\mathit{\Gamma} \vdash M : \Sigma^* x{:}A.\, B \qquad (\mathit{\Gamma} * x : A),\, y : B \vdash N : C^{*(x:A)}}{\mathit{\Gamma} \vdash \mathrm{let}\ M\ \mathrm{be}\ x.y\ \mathrm{in}\ N : C}$$

$$M.N =_{\mathrm{df}} (\mathrm{let\ bind}(M,N) \mathrm{be}\ u^{*m}\ \mathrm{in}\ u)$$

These rules are best explained using the intended model of names. The term $\mathrm{bind}(M,N)$ in (Σ^*-I) may be understood as the pair $\langle M, N\rangle$ with all the names in M made private, together with a proof that the names in M are indeed fresh for the pair. The abbreviation $M.N$ is a short-hand for the pair without the proof of freshness. The introduction rule (Σ^*-I) has a freefrom type in its conclusion because the constructor $\mathrm{bind}(N, M)$

comes from the unit $\eta : B \to W_A \Sigma^*_A B$ of the adjunction, whose codomain $W_A \Sigma^*_A B$ is the semantic equivalent of $(\Sigma^* x{:}A.\, B)^{*(x:A)}$. The elimination rule ($\Sigma^*$-E) formalises the intuition that an element M of type $\Sigma^* x{:}A.\, B$ is a pair with name-hiding. For this intuition to be valid, it should only be possible to use the components of the pair M in such a way that none of the hidden names is revealed. In (Σ^*-E) this is achieved using freefrom types: the term N has type $C^{*(x:A)}$, and such a term can be understood as an element of C whose value does not depend on the names in x.

The equations, in which $(\Gamma * x : A), y : B \vdash R : C^{*(x:A)}$ and $\Gamma, z : \Sigma^* x{:}A.\, B \vdash Q : D$, follow from the triangular identities for the adjunction $\Sigma^*_A \dashv W_A$.

$$(\beta) \qquad \text{let bind}(M, N) \text{be } z^{*u} \text{ in } (\text{let } z \text{ be } x.y \text{ in } R)^{*u} = R[M/x][N/y]$$
$$(\eta) \qquad \text{let } M \text{ be } x.y \text{ in } (\text{let bind}(x, y) \text{be } z^{*x} \text{ in } Q^{*x}) = Q[M/z]$$

We remark that the restriction on freefrom types that B must be closed in $B^{*(M:A)}$ makes the rules for Σ^* incomplete. For example, we have to restrict (Σ^*-I) so that B can only depend on x. More general rules are possible with unrestricted freefrom types.

2.3 Examples and Applications

As a simple example, we show that one can go from $\Pi x{:}A.\, B$ to $\Pi^* x{:}A.\, B$, as is the case in the affine $\alpha\lambda$-calculus.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{f : \Pi x{:}A.\, B \vdash f : \Pi x{:}A.\, B}\,(\text{Proj})}{(f : \Pi x{:}A.\, B) * \Diamond \vdash f : \Pi x{:}A.\, B}\,(\text{Unit})}{(f : \Pi x{:}A.\, B) * x : A \vdash f : \Pi x{:}A.\, B}\,(\text{Weak}) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x : A \vdash x : A}\,(\text{Proj})}{x : A * \Diamond \vdash x : A}\,(\text{Unit})}{\Diamond * x : A \vdash x : A}\,(\text{Swap})}{(f : \Pi x{:}A.\, B) * x : A \vdash x : A}\,(\text{Weak})}{(f : \Pi x{:}A.\, B) * x : A \vdash f\ x : B}\,(\Pi\text{-E})}{f : \Pi x{:}A.\, B \vdash \lambda^* x : A.\, f\ x : \Pi^* x{:}A.\, B}\,(\Pi^*\text{-I})$$

With type dependency and freefrom types, we can express freshness assumptions more precisely than with simply-typed bunches alone. For example, the freshness assertions in the context $x : A,\ y : A,\ u : A^{*(x:A)},\ v : A^{*(\langle x,y \rangle : A \times A)}$ cannot be expressed with simply-typed bunches. On the other hand, the only way the freshness information in freefrom types $B^{*(M:A)}$ can ever be used is via bunches. We then have to ask the question if this is enough to derive useful statements involving freefrom types.

A useful set of rules for working with freefrom types appears in the type system of FreshML [19], which may be seen as a simply typed system with restricted freefrom types. Rules similar to those in FreshML are admissible in our system, thus allowing us to work with freefrom types in the style of FreshML. The main use of freshness in FreshML is for abstraction types (α-equivalence classes) and for the choice of fresh names ($\text{new}\ n.\, M$). Since we will see below that both constructions arise as instances of Π^* and Σ^*, we expect to have at our disposal at least the uses of names and binding as found in FreshML.

Furthermore, with dependent types we can also work with types that are not available in FreshML. For example, assume an inductive type L of lists of names. By structural recursion, we can define a function remove of type $\Pi n{:}\mathbf{N}.\, (\mathsf{L} {\to} \mathsf{L}^{*(n:\mathbf{N})})$ taking a name n and a list l to the list which results by removing n from l. As can be seen from the type, remove also provides a proof that n is fresh for the resulting list. Such freshness information is crucial for defining functions out of α-equivalence classes, to guarantee that the definition is independent of the choice of representative. An example of this, the function computing the free variables of a term, is given in Sec. 3.1 below.

2.4 Models

We summarise the structure required of a category $\mathbb{B}$ so that its codomain fibration models all of the syntax. The interpretation itself also requires this structure to be split, but due to space restrictions we omit the details of the interpretation.

Definition 2. *An affine linear category $\mathbb{B}$ is a* model of the bunched dependent type theory *if it is locally cartesian closed, and if, for each object A in $\mathbb{B}$, the functor W_A as defined above is a fibred functor from cod to $gl(-*A)$ having both fibred left and right adjoint $\Sigma^*_A \dashv W_A \dashv \Pi^*_A$.*

We have seen that the fibred adjunction $W_A \dashv \Pi^*_A$ can be formulated in terms of the monoidal structure. We know of no such non-fibred restatement for $\Sigma^*_A \dashv W_A$.

3 Names and Binding

In this section we consider how the bunched type theory can be used for working with names and binding. To this end, we consider a particular model of the type theory, the Schanuel topos $\mathbb{S}$, which is being widely used as a universe in which to work with names and binding. The Schanuel topos may be thought of as a category of sets involving names. For lack of space, we cannot present it in any detail; the reader is referred to e.g. [7] for its use for names and binding, and to e.g. [15,13,16] for categorical presentations. For the type theory we use the following categorical structure of $\mathbb{S}$.

Proposition 2. *The Schanuel topos $\mathbb{S}$ is a model of the bunched type theory having the following additional structure.*

1. *Finite coproducts which are stable under pullback.*
2. *An object $\mathbf{N}$ for which $[\delta, \imath] : \mathbf{N} + (\mathbf{N} * \mathbf{N}) \to (\mathbf{N} \times \mathbf{N})$ is an isomorphism. Here δ is the diagonal map and $\imath$ is the canonical monomorphism.*
3. *A vertical natural isomorphism $i : \Sigma^*_{\mathbf{N}} \to \Pi^*_{\mathbf{N}}$ such that the triangle below commutes.*

$$\begin{tikzcd} W_{\mathbf{N}}\Sigma^*_{\mathbf{N}} \arrow[rr, "W_{\mathbf{N}}(i)"] & & W_{\mathbf{N}}\Pi^*_{\mathbf{N}} \\ & \mathrm{Id} \arrow[ul, "\eta"] \arrow[ur, "\varepsilon"'] & \end{tikzcd}$$

   Here η is the unit of $\Sigma^*_{\mathbf{N}} \dashv W_{\mathbf{N}}$ and ε is the counit of $W_{\mathbf{N}} \dashv \Pi^*_{\mathbf{N}}$.
4. *For each object A and each monomorphism $m : B \rightarrowtail C$, the commuting square below is a pullback.*

$$\begin{tikzcd} B * A \arrow[r, "\pi_1"] \arrow[d, "m*A"'] \arrow[dr, phantom, "\lrcorner", very near start] & B \arrow[d, "m"] \\ C * A \arrow[r, "\pi_1"'] & C \end{tikzcd}$$

In the rest of this section we explain the structure in this proposition and how it can be integrated in the type theory. We argue informally towards the relation of the above structure to constructions in FM set theory.

As a model of the bunched type theory, $\mathbb{S}$ has both Σ^* and Π^* types. The fresh sums $\Sigma^* x{:}A.\, B$ may be constructed by taking certain equivalence classes of pairs $\langle M, N\rangle$ with $M : A$ and $N : B[M/x]$. Fresh products $\Pi^* x{:}A.\, B$ may be constructed as certain partial functions from A to B. This underpins the view of $\Sigma^* x{:}A.\, B$ and $\Pi^* x{:}A.\, B$ as non-standard sums and products. The difference from the standard sums and products is determined only by the names in A. For a type A that does not contain names, such as the natural numbers, the non-standard sums and products agree with the standard ones.

In Prop. 2.2 we ask for an object $\mathbf{N}$ of names with the property that any two names are either equal, i.e. a single element of $\mathbf{N}$, or they are fresh, i.e. an element of $\mathbf{N}{*}\mathbf{N}$. Thus, names have decidable equality, with two names being different precisely when they are fresh. This object of names plays the same role as the set of atoms $\mathbb{A}$ in FM set theory. We omit the rules for the type of names and its decidable equality, but remark that stable coproducts are used in the formulation of the term for deciding the equality.

Prop. 2.3 concerns the structure of the types $\Sigma^* n{:}\mathbf{N}.\, B$ and $\Pi^* n{:}\mathbf{N}.\, B$. Both types can be used for encoding of α-equivalence classes. An element $n.x$ of type $\Sigma^* n{:}\mathbf{N}.\, B$ is, by construction, an equivalence class and may be understood as the α-equivalence class of x with respect to n. This encoding of α-equivalence classes agrees with that of FM set theory. Indeed, for a closed type B, the construction of $\Sigma^* n{:}\mathbf{N}.\, B$ is (essentially) the same as that of the abstraction set $[\mathbb{A}]B$ of FM set theory. In the work on FM sets, it was also observed that α-equivalence classes may be constructed as partial functions from $\mathbf{N}$ to B. This construction is captured by the type $\Pi^* n{:}\mathbf{N}.\, B$. Therefore, $\Sigma^* n{:}\mathbf{N}.\, B$ and $\Pi^* n{:}\mathbf{N}.\, B$ are different encodings of the same α-equivalence classes, which means that the types should be isomorphic. This explains the isomorphism in Prop. 2.3. The isomorphism is useful for working with α-equivalence classes, as it allows us, for example, to form an α-equivalence class as a pair $n.x$ in $\Sigma^* n{:}\mathbf{N}.\, B$, and then to use it as a function in $\Pi^* n{:}\mathbf{N}.\, B$ to instantiate it at some other name $(n.M)@m$. We give further examples of this in Sec. 3.1, see also [7].

We integrate the isomorphism i in the type theory by means of hidden-name types $\mathrm{H}n.\, B$ which are isomorphic to both $\Sigma^* n{:}\mathbf{N}.\, B$ and $\Pi^* n{:}\mathbf{N}.\, B$. The rules for $\mathrm{H}n.\, B$ are those from both Σ^* and Π^*, giving H a self-dual nature.

$$\text{(H-Ty)}\ \frac{\Gamma * n{:}\mathbf{N} \vdash B \ \mathrm{Type}}{\Gamma \vdash \mathrm{H}n.\, B \ \mathrm{Type}}$$

$$\text{(H-I1)}\ \frac{\Gamma *\ n{:}\mathbf{N} \vdash M : B}{\Gamma \vdash \lambda^*_{\mathrm{H}} n.\, M : \mathrm{H}n.\, B} \qquad \text{(H-E1)}\ \frac{\Gamma \vdash M : \mathrm{H}n.\, B \qquad \Delta \vdash N : \mathbf{N}}{\Gamma * \Delta \vdash M @_{\mathrm{H}} N : B\,[N/n]}$$

$$\text{(H-I2)}\ \frac{n{:}\mathbf{N} \vdash B \ \mathrm{Type} \qquad \Gamma \vdash M : \mathbf{N} \qquad \Gamma \vdash N : B[M/n]}{\Gamma \vdash \mathrm{bind}_{\mathrm{H}}(M, N) : (\mathrm{H}n.\, B)^{*(M:\mathbf{N})}}$$

$$\text{(H-E2)}\ \frac{\Gamma \vdash M : \mathrm{H}n.\, B \qquad (\Gamma * n{:}\mathbf{N}),\, y{:}B \vdash N : C^{*(n:\mathbf{N})}}{\Gamma \vdash \mathrm{let}\ M\ \mathrm{be}\ n._{\mathrm{H}} y\ \mathrm{in}\ N : C}$$

$$M._{\mathrm{H}} N =_{\mathrm{df}} (\mathrm{let}\ \mathrm{bind}_{\mathrm{H}}(M, N) \mathrm{be}\ u^{*m}\ \mathrm{in}\ u)$$

The type $\mathrm{H}n.\, B$ may be interpreted as either $\Sigma^*_{\mathbf{N}} B$ or $\Pi^*_{\mathbf{N}} B$. In the first case, the interpretation of $\lambda^*_{\mathrm{H}} n.\, M$ and $M @_{\mathrm{H}} N$ is given by $i^{-1}(\lambda^* n : \mathbf{N}.\, M)$ and $(i(M))@N$

respectively. With this interpretation, (β) and (η)-equations for H derive from those for Σ^* and Π^*. A further equation, which we omit, arises from the naturality of i.

$$\begin{array}{lll} (\beta 1) & (\lambda^*_{\mathrm{H}} n.\, M) @_{\mathrm{H}} N = M\,[N/n] & \\ (\eta 1) & \lambda^*_{\mathrm{H}} n.\, (M @_{\mathrm{H}} n) = M & n \notin \mathrm{FV}(M) \\ (\beta 2) & \text{let bind}_{\mathrm{H}}(M, N) \text{be } z^{*u} \text{ in } (\text{let } z \text{ be } x.y \text{ in } R)^{*u} = R[M/x][N/y] & \\ (\eta 2) & \text{let } M \text{ be } x._{\mathrm{H}} y \text{ in } (\text{let bind}_{\mathrm{H}}(x, y) \text{be } z^{*x} \text{ in } Q^{*x}) = Q[M/z] & \end{array}$$

The commuting diagram in Prop. 2.3 provides two additional equations, which explain (to some extent) the interaction between the two roles of $\mathrm{H}n.\, B$ as $\Sigma^* n{:}\mathbf{N}.\, B$ and $\Pi^* n{:}\mathbf{N}.\, B$. The equations are formulated in context $\Gamma * n : \mathbf{N}$.

$$\begin{array}{ll} (\beta 3) & \text{let bind}_{\mathrm{H}}(n, N) \text{be } x^{*m} \text{ in } x @_{\mathrm{H}} m = N \\ (\eta 3) & \text{bind}_{\mathrm{H}}(n, \text{let } M \text{be } x^{*m} \text{ in } x @_{\mathrm{H}} m) = M \end{array}$$

From Prop. 2.4 it follows that hidden-name types are in propositions as types correspondence with the freshness quantifier И of Gabbay and Pitts. Consider the logic of subobjects of $\mathbb{S}$. From the fibred adjunction $\Sigma^*_A \dashv W_A \dashv \Pi^*_A$ we can derive a fibred adjunction $\exists^*_A \dashv W^S_A \dashv \forall^*_A$ on $\mathrm{Sub}(\mathbb{S})$, where W^S_A is the endofunctor on $\mathrm{Sub}(\mathbb{S})$ mapping a subobject $m : B \rightarrowtail C$ to $m * A : B * A \rightarrowtail C * A$ (note that $- * A$ preserves pullbacks, and so also monos). Prop. 2.4 then means that W^S_A is nothing but substitution along the projection $\pi_1 : (-) * A \to (-)$. Thus, the propositions as types analogues $\exists^*_A$ of Σ^*_A and $\forall^*_A$ of Π^*_A arise in terms of ordinary quantification along this projection. In the particular case where A is $\mathbf{N}$, it follows from $\Sigma^*_{\mathbf{N}} \cong \Pi^*_{\mathbf{N}}$ that $\exists^*_{\mathbf{N}} = \forall^*_{\mathbf{N}}$. We have thus shown that, along the projection $\pi_1 : (-) * \mathbf{N} \to \mathbf{N}$, the existential and the universal quantifier agree, and it may be seen [16] that this amounts the the freshness quantifier И, i.e. И $= \exists^*_{\mathbf{N}} = \forall^*_{\mathbf{N}}$. As hidden-name types correspond to both $\exists^*_{\mathbf{N}}$ and $\forall^*_{\mathbf{N}}$, they thus correspond to И.

3.1 Examples and Applications

Unique choice of fresh names. For programming with names and binders, it is useful to have the ability to generate fresh names. In FreshML, one can write a term $(\text{new } n.\, M)$, which is thought of as the unique value of M for an arbitrary freshly chosen name n. The existence of such a unique value can be guaranteed by a freshness condition on M. Using our notation, the introduction rule for new may be written as follows.

$$\frac{\Gamma * n : \mathbf{N} \vdash M : C^{*(n:\mathbf{N})}}{\Gamma \vdash \text{new } n.\, M : C}$$

This is derivable in our system by means of the following derivation, in which we write 1 for the unit type with unique element $\diamond : 1$.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma * n : \mathbf{N} \vdash \diamond : 1\end{array}}{\Gamma \vdash \lambda^*_{\mathrm{H}} n.\, \diamond : \mathrm{H}n.\, 1} \qquad \dfrac{\dfrac{\Gamma * n : \mathbf{N} \vdash M : C^{*(n:\mathbf{N})}}{(\Gamma * n : \mathbf{N}), u : 1 \vdash M : C^{*(n:\mathbf{N})}}\text{(Weak)}}{\Gamma,\ z : \mathrm{H}n.\, 1 \vdash \text{let } z \text{ be } n.u \text{ in } M : C}\text{(Weak), (H-E2)}}{\Gamma \vdash \text{let } (\lambda^*_{\mathrm{H}} n.\, \diamond) \text{ be } n.u \text{ in } M : C}\text{(Subst)}$$

We use $(\mathrm{new}\ n.\ M)$ as an abbreviation for the term in the conclusion of this derivation.

In this way, we are using the fact that $\mathrm{H}n.\ 1$ is inhabitated to obtain a supply of fresh names. This generalises the situation in FM set theory or the Theory of Contexts, where one uses the truth of the proposition $(\text{И}\, n.\ \top)$ as a supply of fresh names for reasoning.

Abstract Syntax with Variable Binding. A key application of names and binding is for working with abstract syntax involving variable binders. We encode abstract syntax as an inductive type, using hidden-name types $\mathrm{H}n.\ A$ for object-level binders. The duality of H offers two styles of working with abstract syntax: viewing H as Π^* allows us to work in the style of weak Higher Order Abstract Syntax (wHOAS) [3,11], and viewing H as Σ^* supports the style of FM set theory. In the rest of this section, we give examples illustrating the advantages of both views as well as showing the benefits of mixing the two styles.

We take the syntax of the untyped λ-calculus as an example, encoding it as an inductive type Lam with three constructors: $\mathsf{var} : \mathbf{N} \to \mathsf{Lam}$, $\mathsf{app} : (\mathsf{Lam} \times \mathsf{Lam}) \to \mathsf{Lam}$ and $\mathsf{lam} : (\mathrm{H}n.\ \mathsf{Lam}) \to \mathsf{Lam}$. For example, the term $\lambda x.\ \lambda y.\ (x\ y)$ can be encoded as $\mathsf{lam}(\lambda^*_{\mathrm{H}} x.\ \mathsf{lam}(\lambda^*_{\mathrm{H}} y.\ \mathsf{app}(\mathsf{var}(x), \mathsf{var}(y))))$. In a context with two different names x and y, it may also be encoded as $\mathsf{lam}(x._{\mathrm{H}}\mathsf{lam}(y._{\mathrm{H}}\mathsf{app}(\mathsf{var}(x), \mathsf{var}(y))))$.

Semantically, Lam corresponds to an initial algebra, which lets us define functions by structural recursion. The following recursion principle follows from the initial algebra when $\mathrm{H}n.\ \mathsf{Lam}$ is viewed as $\Pi^* n{:}\mathbf{N}.\ \mathsf{Lam}$.

$$\frac{\begin{array}{l} x{:}\mathsf{Lam} \vdash A(x)\ \mathrm{Type} \\ \Gamma \vdash f : \Pi n{:}\mathbf{N}.\ A(\mathsf{var}(n)) \\ \Gamma \vdash g : \Pi M, N{:}\mathsf{Lam}.\ A(M) \to A(N) \to A(\mathsf{app}(M,N)) \\ \Gamma \vdash h : \Pi M{:}(\mathrm{H}n.\ \mathsf{Lam}).\ (\mathrm{H}n.\ A(M@_{\mathrm{H}}n)) \to A(\mathsf{lam}(M)) \end{array}}{\Gamma \vdash \mathsf{rec}(f,g,h) : \Pi M{:}\mathsf{Lam}.\ A(M)}$$

with equations (in which we write rec for $\mathsf{rec}(f,g,h)$)

$$\begin{aligned} \mathsf{rec}\ \mathsf{var}(n) &= f\ n \\ \mathsf{rec}\ \mathsf{app}(M,N) &= g\ M\ N\ (\mathsf{rec}\ M)\ (\mathsf{rec}\ N) \\ \mathsf{rec}\ \mathsf{lam}(M) &= h\ M\ (\lambda^*_{\mathrm{H}} n.\ (\mathsf{rec}\ (M@_{\mathrm{H}}n))). \end{aligned}$$

For a closed type A, this structural recursion produces a unique function $\mathsf{Lam} \to A$ for given functions $f{:}\mathbf{N} \to A$, $g{:}\mathsf{Lam} \to \mathsf{Lam} \to A \to A \to A$ and $h{:}(\mathrm{H}n.\ \mathsf{Lam}) \to (\mathrm{H}n.\ A) \to A$. In FM set theory one has an apparently different recursion principle, where instead of h one is essentially given a function $k : \mathrm{H}n.\ \mathsf{Lam} \to A \to A^{*(n:\mathbf{N})}$. The above recursion principle is also applicable in this case, since from k we can define $h =_{\mathrm{df}} \lambda u : (\mathrm{H}n.\ \mathsf{Lam}).\ \lambda v : (\mathrm{H}n.\ A).\ \mathrm{new}\ n.\ ((k@_{\mathrm{H}}n)\ (u@_{\mathrm{H}}n)\ (v@_{\mathrm{H}}n))$. In this way, we get a second recursion operator $\mathsf{rec}'(f,g,k)$ with the following equation for the lam-case: $(\mathsf{rec}'(f,g,k)\ \mathsf{lam}(M)) = \mathrm{new}\ n.\ ((k@_{\mathrm{H}}n)\ (M@_{\mathrm{H}}n)\ (\mathsf{rec}'(f,g,k)\ (M@_{\mathrm{H}}n)))$.

As a first example of a recursively defined function, we define capture-avoiding substitution in the style of wHOAS and compare the definition to an FM-style encoding.

Given $m{:}\mathbf{N}$ and $R{:}\mathsf{Lam}$, we can use rec to define $\mathsf{subst} : \mathsf{Lam} \to \mathsf{Lam}$ satisfying

$$\begin{aligned}
\mathsf{subst}(\mathsf{var}(n)) &= \mathrm{ifeq}\ \langle m, n\rangle\ \mathrm{then}\ n.\, R\ \mathrm{else}\ n.\, \mathsf{var}(n)\\
\mathsf{subst}(\mathsf{app}(M, N)) &= \mathsf{app}(\mathsf{subst}(M), \mathsf{subst}(N))\\
\mathsf{subst}(\mathsf{lam}(M)) &= \mathsf{lam}(\lambda^*_{\mathrm{H}} n.\, \mathsf{subst}(M @_{\mathrm{H}} n)).
\end{aligned}$$

This definition uses only the view of H as Π^* and is similar in spirit to wHOAS definitions. We can also define substitution in FM-style using rec'. For the lam-case, we then have $\mathsf{subst}(\mathsf{lam}(M)) = \mathrm{new}\ n.\, (\mathrm{let}\ \mathrm{bind}_{\mathrm{H}}(n, \mathsf{subst}(M @_{\mathrm{H}} n))\mathrm{be}\ w^{*n}\ \mathrm{in}\ (\mathsf{lam}(w))^{*n})$. However, this definition is more complex than the first one, since it involves a unique choice of fresh names via new. In the first definition we could do without the choice of a fresh name by using λ^*_{H} to 'rebind' the fresh name n.

As a second example, we define the function computing the free variables of a term. This example makes essential use of the view of H as Σ^*. We assume an inductive type L of lists of names, together with suitably defined functions $\mathsf{singleton} : \mathbf{N} \to \mathsf{L}$, $\mathsf{concat} : \mathsf{L} \to \mathsf{L} \to \mathsf{L}$, and $\mathsf{remove} : \Pi n{:}\mathbf{N}.\, (\mathsf{L} \to \mathsf{L}^{*(n:\mathbf{N})})$. Using rec, we can define $\mathsf{fv} : \mathsf{Lam} \to \mathsf{L}$ to satisfy the equations

$$\begin{aligned}
\mathsf{fv}(\mathsf{var}(n)) &= \mathsf{singleton}(n)\\
\mathsf{fv}(\mathsf{app}(M, N)) &= \mathsf{concat}(\mathsf{fv}(M), \mathsf{fv}(N))\\
\mathsf{fv}(\mathsf{lam}(M)) &= \mathrm{let}\ (\lambda^*_{\mathrm{H}} n.\, \mathsf{fv}(M @ n))\ \mathrm{be}\ n._{\mathrm{H}} y\ \mathrm{in}\ (\mathsf{remove}\ n\ y)
\end{aligned}$$

This example demonstrates how let-terms can be used for 'pattern matching' elements of $\mathrm{H}n.\, A$. A similar pattern matching appears in FreshML. Moreover, the example shows that it is useful to mix the views of H as Π^* and Σ^*.

Note that, in the equation for lam, the subterm $(\mathsf{remove}\ n\ y)$ has type $\mathsf{L}^{*(n:\mathbf{N})}$, and that this freshness information is necessary for the let to be typeable. Intuitively, this is because the choice of representative $n.y$ must not affect the computation. Dependency in the type of remove is therefore essential for the pattern matching in the definition of fv. Without dependency we could write remove with type $\mathbf{N} \to \mathsf{L} \to \mathsf{L}$, but then fv as above would not be typeable. Indeed, this problem arises in FreshML, where fv cannot be defined using a remove function of this type (Nevertheless, fv can be defined in FreshML).

Again, we can use rec' to give an alternative definition of fv so that it satisfies the equation $\mathsf{fv}(\mathsf{lam}(M)) = \mathrm{new}\ n.\, (\mathsf{remove}\ n\ (\mathsf{fv}(M @ n)))$. Note that, by means of new, this encoding also uses the view of H as Σ^*, and this is in fact essential. The Theory of Contexts, for example, axiomatises a 'is not free in'-predicate rather than defining fv.

4 Discussion and Further Work

We have introduced a bunched dependent type theory that integrates FM concepts for working with names and binding.

One decision in the design of the bunches was to allow dependency for additive context extension but to forbid any dependency for multiplicative context extension. There are other possibilities for combining bunches and dependency. Pym [20, §15.15], for example, outlines a bunched dependent calculus allowing more dependency. The problem with using this for names and binding, which has lead us to the current design, is that it would require to generalise the monoidal product $*$ to a monoidal product on the slices of $\mathbb{S}$, and there seems to be no sensible way of doing this.

We stress that, although the examples in this paper concentrate on programming, reasoning with names and binding can also be accommodated in the type theory. Indeed, it is possible to define a higher-order logic over the dependent type theory [12, §11]. In addition to the usual logical connectives, this logic also features the multiplicative quantifiers $\exists^*$ and $\forall^*$, similar to $\forall_{\mathrm{new}}$ and $\exists_{\mathrm{new}}$ from **BI** [20], as well as the freshness quantifier И. This higher-order logic supports reasoning with names similar to the Theory of Contexts. For example, the Theory of Contexts has an 'extensionality' axiom, which may be expressed as $\Gamma \mid \exists^* n\!:\!\mathbf{N}.\,(M@_{\mathrm{H}}n =_A N@_{\mathrm{H}}n) \vdash (M =_{\mathrm{H}n.\,A} N)$, where M and N have type $\mathrm{H}n.\,A$ and $=_A$ denotes Leibniz equality. Making essential use of the equation $(\eta 3)$, this sequent is derivable in the logic. In another direction, one may also ask how the logic relates to Nominal Logic [18]. For this it is necessary to consider swapping, an essential ingredient of Nominal Logic that is absent from the type theory. We briefly discuss the possibilities of adding swapping below.

Another possibility for reasoning is to use dependent types to encode propositions as types. Alongside the usual encodings of $\forall$ as Π and $\exists$ as Σ, one can encode $\forall^*$ as Π^*, $\exists^*$ as Σ^*, and И as H. Although such an encoding is possible, the use of $\exists^*$ is very restricted, because the rules for Σ^* use types of the form $\varphi^{*(n:\mathbf{N})}$, and, at least in this paper, we allow such types only when φ is closed. Considering a higher-order logic is a way of side-stepping this problem, since, because of Prop. 2.4, we have an equivalence of $\varphi^{*(n:\mathbf{N})}$ and φ, so that freefrom types can be avoided altogether in the logic.

Although we have based our type theory on freshness rather than swapping, we nevertheless think that swapping can be useful in type theory. Swapping can be added to the type theory as a special kind of explicit substitution, as is done in [1,23]. One application of swapping is to make available more information about the isomorphism $\Sigma^*_{\mathbf{N}} \cong \Pi^*_{\mathbf{N}}$ than is given by the commuting triangle in Prop. 2.3. The triangle only explains the instantiation of $n._{\mathrm{H}}x$ at n. With swapping, we can explain the instantiation of $n._{\mathrm{H}}x$ at names other than n by adding the equation $(n._{\mathrm{H}}x)@_{\mathrm{H}}m = (m\ n) \cdot M$. Furthermore, with swapping, we should get a logic close to Nominal Logic; see also [16].

Regarding the categorical semantics of the type theory, it is natural to ask how it compares to other categorical approaches to names and binding. Besides the Schanuel topos, two other categories used frequently [9,4,5, ...] for names and binding are $\mathrm{Set}^{\mathbb{V}}$, where $\mathbb{V}$ is the category of finite cardinals and all functions between them, and $\mathrm{Set}^{\mathbb{I}}$, where $\mathbb{I}$ is the category of finite cardinals and injections. However, neither category has all of the structure of Prop. 2. In $\mathrm{Set}^{\mathbb{V}}$ names do not have decidable equality, whereas $\mathrm{Set}^{\mathbb{I}}$ does not have a freshness quantifier and not all the canonical maps $A * B \to A \times B$ are monomorphic. In this light, Prop. 2 should be viewed as identifying the categorical structure underlying the work with names and binding, while for particular applications it may well be sufficient to have only some of this structure. Another example of such a substructure is Menni's axiomatisation of binders [16]. Nevertheless, there are categories other than the Schanuel topos having the structure of Prop. 2. One such category is a variation of the Schanuel topos in which the elements are allowed to contain countably many names rather than just finitely many, see [18, p.13]. There is also a realisability category having almost all of the structure of Prop. 2, the only restriction being that the type $\Sigma^* x{:}A.\, B$ can only be formed when A belongs to a certain restricted class of types (which includes all types with decidable equality). Moreover, this category models an impredicative universe, so that it should provide the basis for a bunched calculus of constructions.

There are many directions for further work. First, an immediate point requiring further work is the restriction that $B^{*(M:A)}$ can only be formed for closed B. Second, the proof theory of the bunched type theory needs further work. Also, variants such as a non-affine version of the type theory should be possible. Finally, algorithmic questions such as the decidability of type-checking should be considered.

Acknowledgements. We would like to thank Alex Simpson and John Power for interesting discussions on this work.

References

1. L. Cardelli, P. Gardner, and G. Ghelli. Manipulating trees with hidden labels. In *Proceedings of FOSSACS'03*, volume 2620 of *LNCS*. Springer, 2003.
2. L. Cardelli and A. Gordon. Logical properties of name restriction. In *Proceedings of TLCA'01*, volume 2044 of *LNCS*. Springer, 2001.
3. J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In *Proceedings of TLCA'95*, 1995.
4. M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proceedings of LICS99*, 1999.
5. M. Fiore and D. Turi. Semantics of name and value passing. In *Proceedings of LICS01*, 2001.
6. M. Gabbay. FM-HOL, a higher-order theory of names. In *Workshop on Thirty Five years of Automath*, 2002.
7. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
8. M. Hofmann. On the interpretation of type theory in locally cartesian closed categories. In *Proceedings of CSL94*, volume 933 of *LNCS*. Springer, 1994.
9. M. Hofmann. Semantical analysis of higher-order abstract syntax. In *Proceedings of LICS99*, 1999.
10. M. Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 104(1–3):113–166, 2000.
11. F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning about nominal algebras in HOAS. In *Proceedings of ICALP01*, 2001.
12. B. Jacobs. *Categorical Logic and Type Theory*. Elsevier Science, 1999.
13. P.T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium*. Oxford University Press, 2002.
14. P. Lietz. A fibrational theory of geometric morphisms. Master's thesis, TU Darmstadt, May 1998.
15. S. MacLane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer-Verlag, 1992.
16. M. Menni. About И-quantifiers. *Applied Categorical Structures*, 11(5):421–445, 2003.
17. P. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, 2003.

18. A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
19. A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *Proceedings of MPC2000*, volume 1837 of *LNCS*. Springer, 2000.
20. D. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Kluwer Academic Publishers, 1999.
21. R.A.G. Seely. Locally cartesian closed categories and type theory. In *Math. Proc. Cambridge Philos. Soc.*, volume 95, pages 33–48, 1984.
22. P. Taylor. *Practical Foundations of Mathematics*. Cambridge University Press, 1999.
23. C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. In *Proccedings of CSL'03*, volume 2803 of *LNCS*. Springer, 2003.