A APPENDIX

In this appendix we wish to guard the reader against misconceptions about the relationship between nominal unification and higher-order pattern unification, such as the following one that was expressed by an anonymous referee:

> *...the [nominal unification] algorithm is a minor variant of a well-known algorithm for unification of higher-order patterns in the sense of Miller. There, freshness constraints are encoded directly in the term language instead of being carried around separately. The algorithm here can most easily be seen to be a minor variant of prior ones by comparison with [Dowek et al.]*

Miller's higher-order pattern unification algorithms solves equations between λ-terms that satisfy the pattern restriction (see [20]). This algorithm solves such equations modulo $=_{\alpha\beta_0\eta}$-equivalence (where β_0 is a restricted form of β—namely $(\lambda x.M)y = M[y/x]$ with y being a variable in the λ-calculus). If an equation is solvable with respect to this equivalence, then Miller's algorithm produces a *capture-avoiding* substitution. In contrast, nominal unification solves equations between terms that are, roughly speaking, first-order terms plus 'nominal' abstractions, and looks for solutions of such equations with respect to the $\approx$-equivalence defined in Figure 2. Moreover, a solution produced by nominal unification is a *possibly-capturing* substitution *and* a set of freshness constraints. Below we shed more light on the differences between these notions.

- The difference between capture-avoiding substitutions and possibly-capturing substitutions can be seen in the following example: consider the λ-term $\lambda a.X$ and the substitution $[X := a]$; applying this substitution in a possibly-capturing manner yields

  $$\lambda a.a \tag{15}$$

  whereas applying it in a capture-avoiding manner yields

  $$\lambda a'.a \tag{16}$$

  in which the binder a has been renamed to a'. This seemingly minor renaming has however drastic consequences: the terms in (15) and (16) are not α-equivalent! We have illustrated in the Introduction that possibly-capturing substitutions occur often when dealing with schematic inference rules, which are universal in the theory of programming languages.
- Despite having defined our $\approx$-equivalence to capture the traditional notion of α-equivalence (see Theorem 1 and Remark 1), $\approx$ and $=_\alpha$ do *not* coincide. They coincide only on ground terms. To give again the example from the paper, the equation

  $$\lambda a.X =_\alpha \lambda b.X$$

  holds, while

  $$\varnothing \vdash \mathtt{fn}\, a.X \approx \mathtt{fn}\, b.X$$

  does *not* hold. This is a crucial point: if latter equation were to hold, then a possibly-capturing substitution could replace X by a term containing free occurrences of a, equating two terms which are not α-equivalent. So even if Miller's algorithm is restricted to solve unification problems modulo only $=_\alpha$ (instead of $=_{\alpha\beta_0\eta}$), nominal unification and higher-order pattern unification disagree on the equivalence relation that underlies the unification process. Our $\approx$-equivalence is a non-conservative modification of $=_\alpha$ on non-ground terms; its major feature is that, unlike $=_\alpha$, it is respected by possibly-capturing substitutions (see Lemma 2).

- Nominal unification solves freshness problems of the form $a\ \#?\ t$. Such freshness problems have no *direct* equivalent in Miller's algorithm. Also note that they do not only arise when abstractions are unified, but can be arbitrarily chosen. They are a useful programming construct as illustrated in the Conclusion by the `type`-program in which the freshness constraint `x#Gamma` is added to the third clause ensuring that the side-condition of the corresponding inference rule holds. Note that `x#Gamma` does not arise from unifying a goal with the head of this clause—it is an independent constraint given explicitly by the programmer.

The conclusion we draw from this comparison is that it does not make sense to view nominal unification as a variant of the higher-order pattern unification: the algorithms solve problems from different domains (equational problems of λ-terms modulo $\alpha\beta_0\eta$-equivalence vs. equational problems of first-order terms with binders modulo $\approx$-equivalence and in addition freshness problems) and also produce different solutions (capture-avoiding substitutions vs. possibly-capturing substitutions and freshness constraints). Neither the domains, nor the solutions, can be seen as variants of the other.

This comparison, of course, leaves open the question whether there is a suitable *encoding* of nominal unification problems into higher-order pattern unification problems and a suitable *decoding* that translates solutions produced by Miller's algorithm back to solutions of nominal unification problems. Miller's algorithm can then be employed to solve nominal unification problems. At first sight, there seems to be a simple encoding. Consider for example the nominal problem

$$a.X \approx ?\ b.b$$

for which nominal unification produces the solution $(\varnothing, [X := a])$. The literal encoding as the problem $\lambda a.X =_{\alpha\beta_0\eta} ?\lambda b.b$ is unsuitable, because there is no capture-avoiding substitution that solves the problem. However, X can be made dependent on a giving the problem

$$\lambda a.(X a) =_{\alpha\beta_0\eta} ?\ \lambda b.b$$

which can be solved by the capture-avoiding substitution $[X := \lambda c.c]$. If we further apply to $\lambda c.c$ the atom a, which was introduced by the encoding, then one can read back the original solution $[X := a]$ by applying some β-reductions.

There are however several problems with this encoding. First, it is not obvious how to encode uniformly suspensions combined with abstractions, such as

$$b.a.(a\,b){\cdot}X\ . \tag{17}$$

where either a or b need to be applied to X depending on the problem in which this term occurs. This hints at an encoding that makes variables dependent on *all* atoms occurring in a nominal unification problem, regardless of whether they actually occur in an individual equational problem. For example, if X occurs elsewhere within the scope of abstractions of c, d, e and f in a nominal unification problem in which (17) is one term, then X needs to be encoded as $(X\ a\,b\,c\,d\,e\,f)$, although (17) contains only a and b. Secondly, and more importantly, we cannot see how to encode our freshness constraints into higher-order patterns: the reason for this is that, as shown in the `type`-example, they do not necessarily come from abstractions, rather they can be chosen arbitrarily. In effect we do *not* expect that there is an encoding of nominal unification problems into higher-order pattern unification problems at all.

We conjecture that some of the problems mentioned above can indeed be overcome in an encoding of nominal unification problems into $\lambda\sigma$—a calculus with de-Bruijn indices and explicit substitutions presented in [7].The resulting $\lambda\sigma$ unification problems can be solved using the algorithm presented in that paper; however it is not clear how to *decode* the nominal unification solution from the solution of the encoded problem in $\lambda\sigma$. The encoding we have in mind is different from the 'pre-cooking' operation in [7] and is rather complicated: it depends on specific features of de-Bruijn indices and explicit substitutions. We shall sketch the encoding below.

Given a nominal unification problem, all its atoms are encoded as natural numbers, for example $a \to 1$, $b \to 2$ and so on. This mapping will be fixed in an *environment*, which we write as a sequence, for example

$$a.b.c.d.e.f \tag{18}$$

in which the position determines the de-Bruijn index of an atom. As hinted above, variables need to be 'dependent' on all atoms occurring in a problem—in $\lambda\sigma$ we use explicit substitutions for this. Given the variable X and the environment in (18), the encoding leads to

$$|X|_{a.b.c.d.e.f} \mapsto X[1.2.3.4.5.6.\uparrow^6]$$

where $[1.2.3.4.5.6.\uparrow^6]$ is an explicit substitution. One reason for using environments is that permutations can be encoded as permutations on environments, for example

$$|(b\,d){\cdot}X|_{a.b.c.d.e.f} \mapsto |X|_{a.d.c.b.e.f} \mapsto X[1.4.3.2.5.6.\uparrow^6]\ .$$

The other is that they help to define the encoding for abstractions. Because of the way abstractions are represented by de-Bruijn indices, the environment needs to be modified whenever an encoding deals with an abstraction. For example the abstraction $c.e.X$ is encoded as

$$|c.e.X|_{a.b.c.d.e.f} \mapsto \lambda|e.X|_{c.a.b.\bullet.d.e.f} \mapsto \lambda\lambda|X|_{e.c.a.b.\bullet.d.\bullet.f}$$

where the two $\bullet$ indicate, roughly speaking, that c and e are inaccessible under the λs.

As remarked above, the reason why we cannot see how to encode nominal unification problems into higher-order pattern problems has to do with our freshness constraints. In $\lambda\sigma$ there is no *direct* equivalent for freshness constraints, but François Pottier has pointed out to us that freshness is expressible implicitly in $\lambda\sigma$ and hence that an encoding of freshness problems is possible. Consider $b \,\#?\, X$, which can be solved if X is not instantiated with a term containing b freely. The same can be achieved in $\lambda\sigma$ with the problem

$$X =?\ Y[1.3.\uparrow^3]$$

where Y is a fresh variable (not occurring in the nominal unification problem). Note that the explicit substitution $[1.3.\uparrow^3]$ ensures that no matter what is substituted for Y the resulting term cannot mention the de-Bruijn index 2, which corresponds to b being fresh for X.

The main obstacle to relating nominal unification problems with unification problems in $\lambda\sigma$ is the decoding of solutions. Consider the nominal unification problem

$$X \approx?\ a.Y\ ,\ b.Z \approx?\ X$$

which has the solution $(b\#Y, [X := a.Y, Z := (a\,b)\bullet Y])$. Using the encoding sketched above (with the environment $a.b$), one obtains the following unification problem in $\lambda\sigma$

$$X[1.2.\uparrow^2] =? \lambda\, Y[1.3.\uparrow^3]\ ,\ \lambda\, Z[2.1.\uparrow^3] =? \lambda\, X[1.2.\uparrow^2]\ .$$

One solution for this problem is $[X := \lambda\, Y'[1.\uparrow^3], Z := Y'[\uparrow^1], Y := Y'[1.\uparrow^2]]$ in which Y' is a fresh variable. While in this case the solution seems to correspond to the solution of the nominal unification problem above, so far we have been unable to give a decoding that allows us to relate in *every* case a solution produced by unification in $\lambda\sigma$ to a solution produced by the nominal unification algorithm. As can be seen, the freshness constraint $b\#Y$ is completely embedded into the solution $\lambda\sigma$ produces. To make it explicit again seems to be non-trivial in more complicated cases and remains future work.

However, even if the encoding into and decoding from $\lambda\sigma$ can be made to work, there are at least two reasons which justify a direct algorithm:

- The calculations involved in using it to encode a nominal unification problem, find a solution in $\lambda\sigma$ and then decode it back to a 'nominal' solution are likely to be very complicated. Given the relative simplicity of the nominal unification algorithm, one would certainly hope not to have to use this roundabout method in practice.
- An encoding/decoding does not give any insight into the properties of the $\approx$-equivalence. In the paper we have shown that $\approx$ is indeed an equivalence relation and that it is respected by possibly-capturing substitutions. Both facts are non-obvious.[3]

The conclusion we draw from this is that solving nominal unification problems by the unification algorithm described in [7] might be possible, but this is *no* substitute for having the direct algorithm we presented.

[3] See `http://www.cl.cam.ac.uk/~cu200/Unification` for the Isabelle proof scripts.