A NEW Calculus of Contexts

Murdoch J. Gabbay
Dept. of Computer Science
King's College London
Strand, London WC2R 2LS, UK
jamie@dcs.kcl.ac.uk

ABSTRACT

We study contexts (terms with holes) by proposing a 'λ-calculus with holes'. It is very expressive and can encode programming constructs apparently unrelated to contexts, including objects and algorithms in partial evaluation. We give proofs of confluence, preservation of strong normalisation, principal typing for an ML-style Hindley-Milner type system, and an applicative characterisation of contextual equivalence. We explore the limitations of the calculus including further applications, and discuss how they might be tackled.[1]

Categories and Subject Descriptors: F.4.1 [Mathematical Logic and Formal Languages]: lambda calculus and related systems

General Terms: Theory.

Keywords: Calculi of contexts, Functional programming, Binders, Lambda-Calculi, Nominal Techniques.

1. INTRODUCTION

This is a paper about contexts. We shall study them by proposing a "λ-calculus for contexts" which we call the **И calculus of contexts** (И is pronounced 'new'). In this calculus, contexts are terms (first-class values) and can be passed as arguments to functions, applied to other terms, and so on.

We prove • confluence (using results from [36]), • preservation of strong normalisation with respect to the untyped λ-calculus, • give a Hindley-Milner style type system [9] with subject reduction, and • an applicative characterisation of contextual equivalence (drawing on techniques in [10] and [29]). This *small library* of results is chosen to make the NEW context calculus a respectable implementation environment because respectively • the calculus has 'semantic content' (normal forms are unique if they exist), • is 'conservative' over a familar programming language, • can be implemented as an extension of a (toy core of) ML without destroying the type-system, and • programs can be optimised up to preserving extensional behaviour.

What are the technical design issues? A **context** is a term with a 'hole'. The canonical example in the untyped λ-calculus is $C[\text{-}] = \lambda x.\text{-}$. (We use '=' to denote **syntactic α-equality**.)

A term t may be placed in the hole -, e.g. $C[t] = \lambda x.t$ and if $t = x$ then $C[x] = \lambda x.x$. This cannot be modelled inside the calculus by $\lambda y.\lambda x.y$ because β-reduction avoids capture. Our idea is to treat the hole as a variable X which is **stronger** than x and y and for which substitution does not avoid capture of weaker variables. For example, $(\lambda X.\lambda x.X)x \rightsquigarrow \lambda x.x$.

But now α-equivalence is a problem; if $\lambda x.X = \lambda y.X$ then $(\lambda X.\lambda x.X)x \rightsquigarrow \lambda y.x$, giving non-confluent reductions. Dropping α-equivalence entirely is too drastic because some capture-avoidance, as in $(\lambda y.\lambda x.y)x$, should be legitimate.

Existing calculi of contexts include λm and $\lambda\mathcal{M}$ [31] by Sato et al.[2] which seems closest to ours because they have a hierarchy of variables (see below). Some other work [22, 15, 30] use a type system which may prove related to our freshness contexts (see below) though the connections are unclear. Finally we mention λc of Bognar's thesis, which also contains [7, Section 2] an essay on the issues involved in designing context calculi, as well as an extensive literature survey.

How does our calculus relate to the existing literature? The central point of *all* calculi of contexts we know of, including our own, is the interaction of α-equivalence with holes, and the resulting **problems with scope and abstraction**. Solutions include clever control of substitution and evaluation order, and types to prevent 'bad' α-conversions.

Technically, this paper contains *yet another* solution and we are not primarily interested in how that solution works though of course we do describe it in detail. Broadly speaking, its novelty is to imports ideas from previous work by the author and others, known under the umbrella term of 'Nominal techniques' (notably with Urban and Pitts on Nominal Unification [35], and with Fernández on Nominal Rewriting [21, 11]). These ideas were designed to manage α-equivalence; crudely put, we simply applied them to a new problem.

[1]We are very grateful to Pierre Lescanne, Laurence Tratt, Andrew Pitts, David Richter, Gérard Huet, Maribel Fernández, Antonino Salibra, and Gilles Dowek, for encouragement, help, conversations, and advice.

PPDP'05, July 11–13, 2005, Lisbon, Portugal.

[2]We take this opportunity to thank Sato for his hospitality in Kyoto, and for many interesting conversations.

Why do this and why make another context calculus?

First, it has been our personal belief for some time that contexts are significantly underrated.

Our inspiration came from specific technical issues which arise with existential variables in the theorem-proving environment Isabelle [27], and from technical work by Pitts [29] on (what we can slightly inaccurately call) applicative characterisations of contextual equivalence. The problem with existential variables is that the $\exists$ right sequent introduction rule $\dfrac{\Gamma \vdash P(t)}{\Gamma \vdash \exists y.\ P(y)}$ requires us to pick a specific fresh t to make the rest of the proof above $\Gamma \vdash P(t)$ work. This is silly (we want to build t interactively as we construct the rest of the proof). Therefore the designers of Isabelle introduced existential variables $?t$. These resemble 'holes' in λ-calculus contexts, but this is somehow disguised because they are Skolemised over the free variables they may contain.

With Pitts's work, we would claim that contexts — and not terms — are the true object of study. They occur in the definition of contextual equivalence which Pitts uses (originally due to Lockwood Morris), and again with Pitts's notion of 'continuation', which is actually a structured form of a Felleisen-style evaluation context [29, Page 15]. It always seemed to us naughty to have so many contexts around without saying what they are in some more abstract sense.

So our answer is 'here is a calculus of contexts'. Contexts and 'ordinary terms' are on an equal footing as terms in this calculus; indeed so are substitutions and many other things. Results about contexts and substitutions become results about suitable relations (such as β-equivalence or contextual equivalence) between the contexts represented as terms in the calculus.

This is not a denotational answer in quite the same sense as some other work (compare with Hamana [14] and possibly [28]) but for many applications a calculus is just what we want: for example Isabelle theories are encoded in Isabelle/Pure, which is a typed λ-calculus with a simple theory of equality.

But there is more. The NEW context calculus, with its elements of Nominal techniques and hierarchy of variables, gives us fine-grained control of substitution. It turns out that we can apply this to implement constructs which a priori have nothing to do with contexts. We shall explore notions of global state (for general references [2]), objects with forms of in-place update [1], and interesting interactions with partial evaluation [8]; in the conclusions we look at other potential applications such as staged programming (MetaML [26]) and logic and unification.

We are not aware of claims for such implementations made of other context calculi. Perhaps they could implement some or all of our examples (or not; our hierarchy of variables is shared only with [31] and makes a great contribution to expressivity). A different paper might try to make a fair comparison between the different calculi (probably by translating them into ours of course!) and in so doing gain some measure of their relative expressivities. However, we feel it is our *small library* of meta-properties listed above which distinguishes our work from other context calculi. We do not know, for example, of any other applicative characterisation of contextual equivalence.

Therefore, our motivation writing this paper is not *really* to produce a calculus of contexts even though that was our original motivation and this may be sufficient motivation for some readers. We aim further for an implementation environment with good meta-properties and fine-grained control of substitution (thanks to the use of Nominal techniques applied to other ideas already in the literature). We hope that this calculus or something resembling it will provide a convenient environment in which to implement and prove things about other systems which benefit from fine control of substitution including the non-capture-avoiding kind.

In summary: all computation can be coded in the untyped λ-calculus. Some questions then are: *(1)* can we optimise programs, *(2)* can we statically analyse them, and *(3)* can we prove properties of an implementation? If not, what extended calculus allows us to do so. This paper presents an extension of the untyped λ-calculus with constructs to manage capture in substitution, and our examples and meta-properties are chosen precisely to at least suggest that we should be able to answer *(1)* to *(3)* of some useful systems of independent interest.

Besides Nominal Techniques mentioned above, our calculus comes out of a very long tradition of calculi for studying programming. Our calculus has explicit substitutions, see [19] for a survey; our particular treatment of substitution is deliberately simple-minded but still subtle because of interactions with the rest of the language. We give a translation of possibly open terms of the untyped λ-calculus which preserves strong normalisation.

We do not directly import ideas from other calculi of contexts because our management of α-equivalence is based on Nominal techniques with are novel to this field; we do share the hierarchy of variables with one other work [31] and the separation of scope and binding in a λ-calculus appears in one other (unpublished) work we know of [33]. For our meta-properties, we have modelled our proofs on work which we cited above.

2. BASIC SYNTAX AND BASIC EXAMPLE REDUCTIONS

The **syntax of the context calculus** is easy to construct. We formally define reductions in the next section.

We suppose a countably infinite set of disjoint infinite **sets of variables** $\mathbb{A}_1$, $\mathbb{A}_2$, ..., where we write $a_i, b_i, c_i, n_i, \ldots \in \mathbb{A}_i$ for $i \geqslant 1$. We say that a_i **has level** i. The syntax is then given by:

$$s, t ::= a_i \mid tt \mid \lambda a_i.t \mid t[a_i \mapsto t] \mid \mathsf{И} a_i.t.$$

We call $s[a_i \mapsto t]$ an **explicit substitution**. We call $\lambda a_i.t$ an **abstraction**. We equate terms up to α-equivalence on a_i in $\mathsf{И} a_i.t$ (but *not* in $\lambda a_i.t$ or $t[a_i \mapsto t']$). We say '$\mathsf{И}$ is a **binder**' (and λ and explicit substitution are not).

For the moment we ignore $\mathsf{И}$ (and so α-equivalence) and concentrate on examples of the interaction of the hierarchy of variables with explicit substitutions.

We call a variable b_j **stronger** than another a_i when $j > i$, that is, when it has a strictly higher level. We say they have the same strength when they have the same level. For example, b_3 is stronger than a_1. There is no particular connection between variables of different levels with the same name, for example a_1 and a_2.

We now consider some example reductions. We write x, y, z for variables of level 1, and X, Y, Z for variables of level 2.

- **(Ordinary) β-reduction.**

  $$(\lambda x.x)y \rightsquigarrow x[x{\mapsto}y] \rightsquigarrow y.$$

  The explicit substitution implements β-reduction in a standard way. Here and later we colour subterms under an explicit substitution, to help visualise the brackets.

- **Context substitution.** In an explicit substitution $s[a_i{\mapsto}t]$ consider the case $s = \lambda x.X$ (a λ-abstraction of a level 2 variable by a level 1 variable), $a = X$, and $t = x$. The intended operational behaviour is

  $$(\lambda x.X)[X{\mapsto}x] \rightsquigarrow \lambda x.(X[X{\mapsto}x]) \rightsquigarrow \lambda x.x.$$

  The explicit substitution moves under the λx and acts on the X, without capture-avoidance. This kind of substitution (implemented here as a rewrite) is called **context substitution** or **grafting**.

  This example demonstrates that we cannot just equate terms up to α-equivalence, at least not blindly; if $\lambda x.X$ and $\lambda y.X$ were equal terms, context substitution would make no sense.

- **Explicit substitutions on variables.** Canonical reductions (and non-reductions) of terms of the form $a[b{\mapsto}t]$ where a and b are variables and t is any term are:

  $$x[X{\mapsto}t] \rightsquigarrow x \quad x[x'{\mapsto}t] \rightsquigarrow x \quad x[x{\mapsto}t] \rightsquigarrow t \quad X[x{\mapsto}t] \not\rightsquigarrow$$

  A substitution of a strong variable on a weak variable, or two different variables of the same strength, 'evaporates'. A substitution of a variable by itself is a substitution. A substitution of a weak variable on a stronger variable does not in general reduce; we say $[x{\mapsto}t]$ is **suspended** on X. (The precise rules $(\sigma\mathbf{a})$ and $(\sigma\#)$ are introduced below, but first we must introduce binding, and we give some more examples before that.)

  A canonical reduction illustrating the use of suspensions, is as follows:

  $$X[x{\mapsto}t][X{\mapsto}x] \rightsquigarrow X[X{\mapsto}x][x{\mapsto}t[X{\mapsto}x]] \rightsquigarrow x[x{\mapsto}t[X{\mapsto}x]] \rightsquigarrow t[X{\mapsto}x].$$

  So we can think of $b_j[a_i{\mapsto}t]$ for $i < j$ as a strong hole b_j with a substitution on a weaker a_i waiting for it to be filled.

- **Substitutions and λ-abstraction.** A λ-abstracted atom can never get substituted for, not even by a stronger substitution. For example $(\lambda x.x)[x{\mapsto}X] \rightsquigarrow \lambda x.x$.

- **Substitutions as terms.** Substitution $[a_i{\mapsto}t]$ is not a term and so cannot be made an argument of a function. Using suspensions we can express this: for example $\lambda X.X[x{\mapsto}y]$ encodes $[x{\mapsto}y]$ as a term.

  Reduction is not possible since X is stronger than x. Contrast this with $\lambda X.(X[X{\mapsto}y])$, which reduces in one step to $\lambda X.y$.

  Define $\mathbf{true} \equiv \lambda x, y.x$, $\mathbf{false} \equiv \lambda x, y.y$, and $\mathbf{Id} \equiv \lambda x.x$. Here is a program which takes a substitution, a truth value, and an argument, and applies the substitution or not according to the truth value: $f \equiv \lambda x, y, z.(yx\mathbf{Id})z$.

  Here is an example of reduction using f:

  $$f\ (\lambda X.X[x{\mapsto}y])\ \mathbf{true}\ x \rightsquigarrow^* y$$
  $$f\ (\lambda X.X[x{\mapsto}y])\ \mathbf{false}\ x \rightsquigarrow^* x.$$

To encode substitutions for stronger variables $[a_i{\mapsto}s]$ (we call these **stronger substitutions**), the general scheme is $\lambda a_{i+1}.(a_{i+1}[a_i{\mapsto}s])$. This is the first real use of level three variables, which are useful to encode the 'context substitution' $[X{\mapsto}s]$ as a term.

We hope this clarifies everything *except* for binding. We shall now give a formal definition of the syntax and operational semantics of the context calculus, all in a block, and then we return to the examples.

3. FORMAL DEFINITION OF THE NEW CONTEXT CALCULUS

We inherit the definitions, notation, and terminology of the beginning of the last section. Briefly, assume an infinity of disjoint sets of variables $a_i, b_i, c_i, n_i, \ldots \in \mathbb{A}_i$ for $i \geqslant 1$ and say that a_i has **level** i. The syntax is:

$$t ::= a_i \mid tt \mid \lambda a_i.t \mid t[a_i{\mapsto}t] \mid \text{И}a_i.t.$$

И is the only binder and we work up to α-equivalence between variables of the same level bound by И. So for instance $\text{И}a_1.a_1 = \text{И}b_1.b_1$ but $\text{И}a_1.a_1 \neq \text{И}a_2.a_2$. It may also be convenient to allow constants, e.g. for arithmetic or truth-values. They behave much like variables which we do not abstract or substitute for and we tend to ignore them. Note that $[a_i{\mapsto}t]$ is not a term.

Reduction rules are given on terms as follows:

(β)	$(\lambda a_i.s)u \rightsquigarrow s[a_i{\mapsto}u]$	
$(\sigma\mathbf{a})$	$a_i[a_i{\mapsto}u] \rightsquigarrow u$	$\forall j, c_j.\ j{<}i \wedge c_j\#a_i{\Rightarrow}c_j\#u$
$(\sigma\#)$	$s[a_i{\mapsto}u] \rightsquigarrow s$	$a_i\#s$
$(\sigma\mathbf{p})$	$(a_it_1 \ldots t_n)[b_j{\mapsto}u] \rightsquigarrow (a_i[b_j{\mapsto}u]) \ldots (t_n[b_j{\mapsto}u])$	
$(\sigma\sigma)$	$s[a_i{\mapsto}u][b_j{\mapsto}v] \rightsquigarrow s[b_j{\mapsto}v][a_i{\mapsto}u[b_j{\mapsto}v]]$	$j{>}i$
$(\sigma\lambda)$	$(\lambda a_i.s)[c_k{\mapsto}u] \rightsquigarrow \lambda a_i.(s[c_k{\mapsto}u])$	$a_i\#u, c_k\ k{\leqslant}i$
$(\sigma\lambda')$	$(\lambda a_i.s)[b_j{\mapsto}u] \rightsquigarrow \lambda a_i.(s[b_j{\mapsto}u])$	$j > i$
$(\sigma\mathbf{tr})$	$s[a_i{\mapsto}a_i] \rightsquigarrow s$	
$(\text{И}\mathbf{p})$	$(\text{И}n_j.s)t \rightsquigarrow \text{И}n_j.(st)$	$n_j{\notin}t$
$(\text{И}\lambda)$	$\lambda a_i.\text{И}n_j.s \rightsquigarrow \text{И}n_j.\lambda a_i.s$	$n_j{\neq}a_i$
$(\text{И}\sigma)$	$(\text{И}n_j.s)[a_i{\mapsto}u] \rightsquigarrow \text{И}n_j.(s[a_i{\mapsto}u])$	$n_j{\notin}u\ n_j{\neq}a_i$
$(\text{И}\notin)$	$\text{И}n_j.s \rightsquigarrow s$	$n_j{\notin}s$

$n_j \neq a_i$ means 'n_j **and** a_i **are distinct symbols**'. $n_j \notin t$ means 'n_j **does not occur in** t'. We use these conventions silently henceforth. The intuition of $a\#t$ is 'a cannot occur free in t'; defining and explaining its formal meaning is for the rest of this subsection. j in $\forall j$ varies over all levels and c_j in $\forall c_j$ varies over all variables of that level, so the side-condition on $(\sigma\mathbf{a})$ is 'for all variables c_j weaker than i, if $c_j\#a_i$ then $c_j\#u$ (see the end of §4.1 for further discussion). In $(\sigma\mathbf{p})$ and henceforth we write $((((mu)p)p)e)t$ as *muppet*, associating application to the left (as is standard).

Many of these rules are familiar from λ-calculi of explicit substitutions (see [19] for examples). $(\sigma\#)$ is reminiscent of Bloo's 'garbage collection' [6] and we refer the reader there for a further discussion. The rules involving И are scope extrusion rules, e.g. as in the π-calculus [24].

Call a pair of a variable and a term $a_i\#t$ a **freshness assertion**, we let ϕ vary over them. If $t = b_j$ is a variable and $i < j$ it a **primitive freshness assertion**. Call a possibly infinite set Γ of primitive freshness assertions a **freshness context**.

Define a notion of entailment between freshness contexts and freshness assertions by:

$$\frac{j > i}{b_j\#a_i} \quad \frac{a_i \neq b_i}{a_i\#b_i} \quad \frac{}{a_i\#\lambda a_i.u} \quad \frac{b_j\#u}{b_j\#\lambda a_i.u} \quad \frac{a_i\#s \quad a_i\#t}{a_i\#st}$$

$$\frac{a_i\#u}{a_i\#s[a_i{\mapsto}u]} \quad \frac{a_i\#s \quad b_j\#s}{b_j\#s[a_i{\mapsto}u]} \quad \frac{b_j\#s \quad b_j\#u}{b_j\#s[a_i{\mapsto}u]} \quad \frac{a_i\#s}{a_i\#\mathit{M}b_j.s}$$

We say $a\#s$ is **entailed** by Γ and write $\Gamma \vdash a\#s$, when $a\#s$ can be derived from Γ using these rules.

Read bottom-up these are rewrite rules on (sets of) freshness assertions. They reduce assertions to a normal form Sol which is a set of freshness contexts. For example $a_1\#a_1a_1$ reduces to $a_1\#a_1$, and $a_1\#b_1$ reduces to the empty set. $\Gamma \vdash a\#s$ holds when $Sol \subseteq \Gamma$.

Here are some examples/comments.

- $\Gamma \vdash b_j\#a_i$ always if $j > i$, or if $j = i$ but $b_j \neq a_i$. This expresses the intuition that a stronger hole can never 'occur in' a weaker one, and that two different holes of the same strength are different.

- $\Gamma \vdash a_i\#\lambda a_i.s$. We say that λ **abstracts** and this expresses a natural intuition.

- $\Gamma \vdash a_i\#s[a_i{\mapsto}t]$ if $\Gamma \vdash a_i\#t$. This expresses an intuition that substitution abstracts away all a_i in s.

- $\Gamma \vdash a_1\#b_2c_2$ is derivable from Γ if $\{a_1\#b_2, a_1\#c_2\} \subseteq \Gamma$. This expresses the intuition that 'if a_1 is not free in b_2 and a_1 is not free in c_2, then a_1 is not free in b_2 applied to c_2'. Here b_2 and c_2 are stronger variables which may for example 'become' a_1, in the sense that b_2c_2 may occur in a subterm of a larger term containing substitutions $[b_2{\mapsto}a_1]$.

A freshness context Γ induces a reduction relation $\Gamma \vdash s \rightsquigarrow t$, we may just write $s \rightsquigarrow t$, on the syntax of the calculus. It is given by reading the conditions $a_i\#b_j$ and $n_j\#t$ as $\Gamma \vdash a_i\#b_j$ and $\Gamma \vdash n_j\#t$.

Lemma 3.1 (Subject reduction for #). *If* $\Gamma \vdash a_i\#s$ *and* $\Gamma \vdash s \rightsquigarrow t$ *then* $\Gamma \vdash a_i\#t$.

Proof. By case-analysis on the rewrites defining $s \rightsquigarrow t$ *and* on the position in which the rewrite occurs in s. □

controls abstraction; what a variable can be replaced by, and what can be done under a λ and an explicit substitution. This is distinct from from scope (and literal binding in the syntax), which remains relatively simple and is controlled by M (M is the *only* binder, in the traditional sense of renaming variables). We discuss this further in the rest of the paper.

We note that $(\sigma\#)$ resembles Bloo and Rose's 'garbage collection rule' [6]. The deduction rules for # and explicit substitutions implement a notion of unabstracted variable which they reject [6, Remark 2.3] for reasons we seem to avoid because of the separation of binding and abstraction.

3.1 Freshness #, α-equivalence, and M

Why is all the apparatus of # necessary; why not use $\notin$?

The following two rewrites would be valid:

$$\begin{aligned}
(\lambda x.y)[y{\mapsto}X][X{\mapsto}x] &\overset{(\sigma\lambda)}{\rightsquigarrow} (\lambda x.y[y{\mapsto}X])[X{\mapsto}x] \\
&\overset{(\sigma\lambda')}{\rightsquigarrow} \lambda x.(y[y{\mapsto}X][X{\mapsto}x]) \\
&\overset{(\sigma\mathbf{a})}{\rightsquigarrow}{}^{*} \lambda x.x \\
(\lambda x.y)[y{\mapsto}X][X{\mapsto}x] &\overset{(\sigma\sigma)}{\rightsquigarrow} (\lambda x.y)[X{\mapsto}x][y{\mapsto}X[X{\mapsto}x]] \\
&\overset{(\sigma\mathbf{a})}{\rightsquigarrow} (\lambda x.y)[X{\mapsto}x][y{\mapsto}x] \\
&\overset{(\sigma\lambda')}{\rightsquigarrow} \lambda x.(y[X{\mapsto}x])[y{\mapsto}x] \not\rightsquigarrow
\end{aligned}$$

These rewrites are not confluent (and we want confluence). The *problem rewrite* is the first one, where the strong variable X (which contains only X in its raw syntax, and in particular does not contain x) comes under an abstraction by a weak variable x and is then substituted for by non-capture-avoiding context substitution.

We use freshness assumptions to manage abstraction. *Fix* some freshness context Γ and read the side-conditions $\phi \in \{n_j\#u,\ n_j\#t,\ a_i\#b_j,\ \ldots\}$, in the definition of reduction from the previous subsection, as meaning $\Gamma \vdash \phi$. Each freshness context induces a different reduction relation.

By way of example, we re-consider the example reductions above in the **universal** freshness context $\Theta = \{a_i\#b_j \mid i < j\}$, and the **empty** freshness context $\Phi = \varnothing$.

In the universal freshness context Θ the term above rewrites as follows:

$$\begin{aligned}
\Theta \vdash (\lambda x.y)[y{\mapsto}X][X{\mapsto}x] &\overset{(\sigma\lambda)\ x\#X}{\rightsquigarrow} (\lambda x.y[y{\mapsto}X])[X{\mapsto}x] \\
&\overset{(\sigma\lambda')}{\rightsquigarrow} \lambda x.(y[y{\mapsto}X][X{\mapsto}x]) \\
&\overset{(\sigma\mathbf{a})}{\rightsquigarrow} \lambda x.(X[X{\mapsto}x]) \not\rightsquigarrow \\
\Theta \vdash (\lambda x.y)[y{\mapsto}X][X{\mapsto}x] &\overset{(\sigma\sigma)}{\rightsquigarrow} (\lambda x.y)[X{\mapsto}x][y{\mapsto}X[X{\mapsto}x]] \\
&\overset{(\sigma\lambda')}{\rightsquigarrow} (\lambda x.(y[X{\mapsto}x]))[y{\mapsto}X[X{\mapsto}x]] \\
&\overset{(\sigma\#)\ X\#y}{\rightsquigarrow} (\lambda x.y)[y{\mapsto}X[X{\mapsto}x]] \\
&\overset{(\sigma\lambda)\ x\#X[X{\mapsto}x]}{\rightsquigarrow} \lambda x.(y[y{\mapsto}X[X{\mapsto}x]]) \\
&\overset{(\sigma\mathbf{a})}{\rightsquigarrow} \lambda x.X[X{\mapsto}x]] \not\rightsquigarrow
\end{aligned}$$

Here $X[X{\mapsto}x]$ cannot reduce (by $(\sigma\mathbf{a})$) because $\Theta \vdash x\#X$ and $\Theta \not\vdash x\#x$.

In the empty freshness context Φ,

$$\begin{aligned}
\Phi \not\vdash (\lambda x.y)[y{\mapsto}X][X{\mapsto}x] &\overset{(\sigma\lambda)}{\rightsquigarrow} (\lambda x.y[y{\mapsto}X])[X{\mapsto}x] \\
\Phi \vdash (\lambda x.y)[y{\mapsto}X][X{\mapsto}x] &\overset{(\sigma\sigma)}{\rightsquigarrow} (\lambda x.y)[X{\mapsto}x][y{\mapsto}X[X{\mapsto}x]] \\
&\overset{(\sigma\lambda')}{\rightsquigarrow} (\lambda x.(y[X{\mapsto}x]))[y{\mapsto}X[X{\mapsto}x]] \\
&\overset{(\sigma\#)\ X\#y}{\rightsquigarrow} (\lambda x.y)[y{\mapsto}X[X{\mapsto}x]] \\
&\overset{(\sigma\mathbf{a})}{\rightsquigarrow} (\lambda x.y)[y{\mapsto}x] \not\rightsquigarrow
\end{aligned}$$

The first reduction cannot take place because $\Phi \not\vdash y\#X$.

(We note in passing that we can emulate a constant with a variable a such that $b\#a$ always.)

Is there a *canonical* freshness context between the extremes of Θ and Φ? We need some notation: given a set of variables S write $\max(S)$ for the level of a strongest variable, if such a variable exists, and $\min(S)$ for the level of the weakest. For example, $\max\Phi = 0$ and $\max\{a_2, b_3\} = 3$. $\max\Theta$ is not well-defined. Given any set of variables I write $b_j\#I$ for the set $\{b_j\#a_i \mid a_i \in I\}$. Write $\Gamma \vdash b_j\#I$ for the assertion $\forall a_i \in S.\ \Gamma \vdash b_j\#a_i$.

A freshness context Γ **has sufficient freshnesses** when for all finite sets of variables S and all i, there exists $b_i \notin S$ with $\Gamma \vdash b_i\#S$, and for all $i \leqslant \min(S)$, there exists $a_i \notin S$ with $\Gamma \not\vdash a_i\#S$.

So Γ has two kinds of freshness: internally, for any finite set we can find a variable which Γ promises is fresh for them, but also externally we can find a variable fresh in the sense that Γ knows nothing about its relation to S (provided the variable is no stronger than any element of S, in which case its freshness may be deducible from the derivation rules, e.g. $a_2\#a_1$).

It is not hard to inductively construct a Γ with sufficient freshnesses inductively:

- Let $\Gamma_0 = \{\}$ and $U_0 = \{\}$.
- Given Γ_l and U_l, enumerate all subsets $S_1, \ldots, S_n$ of U_l. For each S_j and $i \leqslant \max(U_l) + 1$ choose distinct fresh b_i^j. For each S_j and $i \leqslant \min(S)$ choose distinct fresh a_i^j. Let $\Gamma_{i+1} = \Gamma_i \cup \{b_i^j \# S_j \mid i, j\}$ and let $U_{l+1} = U_l \cup \{a_i^j, b_i^j \mid i, j\}$.

Iterate and let $\Gamma = \bigcup_i \Gamma_i$ (actually, we must take the normal form with respect to the deduction rules for $\#$).

LEMMA 3.2. *If Γ has sufficient freshnesses then for any term t and any i there exists a_i such that $\Gamma \vdash a_i\#t$. Similarly for a finite set of terms. We may say a_i is **sufficiently fresh**, in the case that the other parameters are fixed or understood.*

PROOF. Let S be all variables mentioned in t and choose a_i such that $\Gamma \vdash a_i\#S$. It is not hard to prove by induction on t that $\Gamma \vdash a_i\#t$ is derivable. □

For the rest of this paper, **we silently fix a context** Γ with sufficient freshnesses. Thus when we write '$a\#s$' we mean '$\Gamma \vdash a\#s$'. This is just the usual assumption that 'we can always find a fresh variable'. The hierarchy of strengths of variables created problems which we solved with an *explicit* freshness context, so we make *explicit* assumptions about it; but the associated technical details work as we would expect so we tend to retain informal practice and terminology.

3.2 Intuitive 'facts' and silly reductions

We might expect that if $x \notin s$ then $x\#s$. This is not true; $x\#X$ is not true in general, only for those x and X such that $x\#X$ is in the context.

Conversely, it is not true that if $x\#s$ then $x \notin s$, a counterexample is $X[x{\mapsto}y]$. We leave it to the reader to check the derivation of $x\#(X[x{\mapsto}y])$ using the rules defining $\#$.

Why do rules $(\text{И}\star)$ for $\star \in \{p, \lambda, \sigma, \notin\}$ all use inequality $\notin$ and $\neq$, whereas rules $(\sigma\star)$ for $\star \in \{a, \#, \lambda\}$ use $\#$? Intuitively, И regulates α-equivalence, which has to do with *occurrences* and *scope*, and $\#$ regulates abstraction, which has to do with what can be substituted or moved where! For illustration we propose silly (=non-confluent) reductions based on (false) rules consistently using only $\#$ and only $\notin$ respectively:

$$(\text{И}x.X[x{\mapsto}y])[X{\mapsto}x] \overset{(\text{И}\sigma)\ x\#X[x\mapsto y]}{\rightsquigarrow} \text{И}x.(X[x{\mapsto}y][X{\mapsto}x]) \rightsquigarrow^* \text{И}x.x$$

This is silly, because we can α-rename the И-bound x to some x' and (as we can easily verify) reduce to $\text{И}x'.x$.

$$X[x{\mapsto}0][X{\mapsto}x] \overset{(\sigma\#)\ x\notin X}{\rightsquigarrow} X[X{\mapsto}x] \rightsquigarrow X.$$

This is silly, because we can also reduce with $(\sigma\sigma)$ to obtain 0.

A rule $s(\text{И}a.t) \rightsquigarrow \text{И}a.(st)$ (if $a \notin s$) would be silly:

$$(\lambda x.xx)\text{И}y.y \rightsquigarrow \text{И}y.(\lambda x.xx)y \rightsquigarrow^* \text{И}y.yy$$
$$(\lambda x.xx)\text{И}y.y \rightsquigarrow^* \text{И}y.y\text{И}y'.y' \rightsquigarrow \text{И}y, y'.yy'.$$

For this reason, И remains trapped in any position where it might be copied.

4. PROGRAMMING IN THE CALCULUS

Having defined and motivated the formal design of the calculus, we explore its expressivity, at first with respect to existing calculi in the literature but also with respect to some interesting applications.

4.1 Some simple reductions

We start by seeing how some familiar reductions work:

Consider variables x_1, y_1, X_2 (with levels 1, 1, and 2 respectively; we may state levels as subscripts for the first use of a variable, then drop them). We might expect $(\lambda x.s)x$ to reduce to s. This is so:

$$(\lambda x.s)x \overset{(\beta)}{\rightsquigarrow} s[x{\mapsto}x] \overset{(\sigma\mathbf{tr})}{\rightsquigarrow} s.$$

Note however that λ does not bind; does $(\text{И}\lambda x.s)x$ reduce to s? **Here and henceforth** we write $\text{И}\lambda x.s$ for $\text{И}x.\lambda x.s$. The answer is 'yes', but for more complex reasons. We consider the first part of its reduction:

$$(\text{И}\lambda x.s)x \overset{(\text{И}\mathbf{p})\ y\#x}{\rightsquigarrow} \text{И}y.((\lambda y.s[y/x])x) \overset{(\beta)}{\rightsquigarrow} \text{И}y.(s[y/x][y{\mapsto}x])$$

Write $[t/x]$ for **term substitution** avoiding capture by И-bound variables. We choose y such that $y\#x$ so we can extend the scope of И; here we assume the context has sufficient freshnesses.

Note that $s[y/x][y{\mapsto}x]$ does not in general reduce to $s[x/y]$. A counterexample is $s = X_2[y{\mapsto}x]$ ($X[y/x] = X$). y is weaker than X, so reductions stop. However:

LEMMA 4.1. *If the level of a is no less than that of the other variables mentioned in s then $s[a{\mapsto}t] \rightsquigarrow^* s[t/a]$ and as a corollary, $(\text{И}\lambda x.s)x \rightsquigarrow^* s$.*

PROOF. By induction on the reduction rules. □

We would expect $(\lambda x.s)t$ and $(\text{И}\lambda x.s)t$ to reduce to s if x does not occur in s. In fact, the first statement is not true! A counterexample is $(\lambda x.X)y \overset{(\beta)}{\rightsquigarrow} X[x{\mapsto}y]$. This does not reduce further unless $x\#X$. Of course, this is the point of the context calculus; X is stronger than x and a substitution may arrive from elsewhere in the term to instantiate it to another term, such as x, unless the context prohibits this. The second statement is true. The reductions are:

$$(\text{И}\lambda x.s)t \overset{(\text{И}\mathbf{p})\ x\#t}{\rightsquigarrow} \text{И}x.((\lambda x.s)t) \rightsquigarrow^* \text{И}x.s[x{\mapsto}t] \overset{(\sigma\#)\ x\#s}{\rightsquigarrow} \text{И}x.s \overset{(\text{И}\notin)\ x\notin s}{\rightsquigarrow} s.$$

The conditions on x annotating the reductions can be met by successive α-renamings, because the context has sufficient freshness.

We never use the following result in this paper because we always work with reductions of a fixed term, but it is important because it expresses something about the sense in which variables represent terms:

LEMMA 4.2. *Suppose s is a term mentioning a_i whose level (i) is no less than that of the other variables in s. Suppose u is a term such that for all variables c, if $\Gamma \vdash c\#a_i$ then $\Gamma \vdash c\#u$. Then if $s \rightsquigarrow t$ then $s[u/a_i] \rightsquigarrow t[u/a_i]$.*

PROOF. By exhaustively checking the reduction rules. □

With regards to $(\sigma\mathbf{a})$, the two canonical cases are: *(a)* $x[x{\mapsto}y] \overset{\sigma a}{\rightsquigarrow} y$ always; the side-condition is vacuous because x and y have the same strength. *(b)* $X[x{\mapsto}y]$ reduces if $x\#X$ and $y\#X$ or $\neg(x\#X)$ and $\neg(x\#X)$, but not if $x\#X$ and $\neg(y\#X)$. This rule is the crunch case, where name-capture (encoded by freshness conditions) can block a substitution actually occurring. Intuitively, a term t can only be substituted for a variable, X say, if it satisfies the intentional conditions on use of variables which the freshness context associates to X.

4.2 The untyped lambda-calculus

Terms of the untyped λ-calculus are given by

$$e ::= x \mid ee \mid \lambda x.e.$$

λ binds x in $\lambda x.e$. This is standard [3]. A translation into the context calculus is given by:

$$\llbracket x \rrbracket = x \qquad \llbracket ee' \rrbracket = \llbracket e \rrbracket \llbracket e' \rrbracket \qquad \llbracket \lambda x.e \rrbracket = \text{И}\lambda x.\llbracket e \rrbracket.$$

(Recall $\text{И}\lambda x.blah$ is shorthand for $\text{И}x.\lambda x.blah$.) We also assume some injection from the variables in the untyped λ-calculus to variables of level 1 in the context calculus. **We may silently assume these injections henceforth.**

Since λ does not bind in the context calculus we need an explicit И. Otherwise, the translation is as one would expect. We omit the operational semantics $e \to e'$ of the untyped λ-calculus; a **simulation** result is easy to prove:

LEMMA 4.3. *If $e \to e'$ then $\llbracket e \rrbracket \rightsquigarrow^* \llbracket e' \rrbracket$.*

Multiple reductions $\rightsquigarrow^*$, because the context calculus is an explicit substitution calculus. The proof is routine but subtle for reasons similar to the issues discussed in the last subsection. We use the fact that all variables in $\llbracket e \rrbracket$ have level 1, and the corollary of the previous subsection.

Another correctness result is:

LEMMA 4.4 (PRESERVATION OF STRONG NORMALISATION). *If e is any (possibly open) untyped λ-term then e has a normal form if and only if $\llbracket e \rrbracket$ does.*

PROOF. Confluence of the untyped λ-calculus means that we can choose a strict left-to-right $\to$-reduction strategy and be confident of arriving at the normal form if it exists. This particular rewrite strategy is easily simulated by the context calculus, one β-reduction corresponding to one instance of (β) followed by other rules (other reduction strategies can be less obvious because of the particular form of $(\sigma\mathbf{p})$). It is easy to verify that if e is a normal form (has no valid β-reducts) then $\llbracket e \rrbracket$ has no valid instances of (β). Finally, the reductions of the context calculus *without* (β) are strongly normalising. □

4.3 The $q\nu$-calculus

Baro and Maurel's $q\nu$-calculus [33] shares the context calculus's unusual feature of spliting λ into a binder ν and an abstractor q. (We are not aware of this idea anywhere else in the literature except for our work with Fernández on extensions of Nominal Rewriting [11].) This translates perfectly into the context calculus, with ν mapping to И and q to λ.

4.4 Records

Say a **record** is a term of the form $b_j[a^1_{i_1}{\mapsto}t_1]\cdots[a^n_{i_n}{\mapsto}t_n]$ where $j > i_k$ for $1 \leqslant k \leqslant n$. In words, a record is a set of substitutions suspended on a 'big hole' b_j.

Consider variables X_2, l_1, and p_1, and terms t_l, t_p (think of them as 'data'). For convenience drop the subscripts on variables, **a convention we shall use silently henceforth**.

So $R = X[l{\mapsto}t_l][p{\mapsto}t_p]$ is a simple record, we say it has two data elements t_l and t_p which are **stored at** l **and** p **on** X. Call the term $\text{И}\lambda a_3.a_3[X{\mapsto}l]$ **record lookup at** l and $\text{И}\lambda a_3.a_3[X{\mapsto}X[l{\mapsto}t'_l]]$ **record update at** l.

We observe the following reductions:

$$(\text{И}\lambda a_3.a_3[X{\mapsto}l])R \overset{(\beta)}{\rightsquigarrow} \text{И}a_3.a_3[X{\mapsto}l][a_3{\mapsto}R]$$
$$\overset{(\sigma\sigma),(\sigma\mathbf{a})}{\rightsquigarrow^*} \text{И}a_3.R[X{\mapsto}l] \overset{(\sigma\sigma)}{\rightsquigarrow^*} \text{И}a_3.l[l{\mapsto}t_l[X{\mapsto}l]][p{\mapsto}t_p[X{\mapsto}l]]$$
$$\overset{(\sigma\mathbf{a}),(\text{И}\#)}{\rightsquigarrow^*} t_l[X{\mapsto}l].$$

We leave it to the reader to verify similar reductions for record update.

So record lookup at l is a term which, when applied to a record, *almost* retrieves the data element — except for that troublesome $[X{\mapsto}l]$. We call this **name capture**.

If we suppose for the sake of argument that $X\#t_l$, then we can use $(\sigma\#)$ to drop the troublesome substitution. There are two ways of looking at this: either $X\#t_l$ is a desirable feature, or an undesirable restriction.

To avoid name capture it is not enough to insist that X not occur in t_l and t_p. For example suppose $t_l = O_4$. Then the substitution $[O{\mapsto}X]$ may arrive from the outside context, transforming t_l into X! Of course we can simply insist that $X\#t_l$ be provable, but there is also a more elegant way:

We can call a $R' = \text{И}\lambda X.X[l{\mapsto}t_l][p{\mapsto}t_p]$ (say X is **protected**). Then protected record lookup at l is encoded by $\text{И}\lambda a_3.a_3 l$ and record update by $\text{И}\lambda a_3.\text{И}\lambda X.a_3(X[l{\mapsto}t'_l])$. Then the variable X is bound by a И and will avoid capture (assuming, of course, that X does not actually occur in the syntax of t_l when R' is created).

More sophisticated programming with records is possible, for example **in-place update**. Suppose the calculus contains basic arithmetic and consider $X[l{\mapsto}1]$, a simple record 1 stored at l on X. We can apply the substitution $[X{\mapsto}X[l{\mapsto}l+1]$. The reader can verify the reduction

$$(X[l{\mapsto}1])\ [X{\mapsto}X[l{\mapsto}l+1]] \rightsquigarrow^* X[l{\mapsto}l+1][l{\mapsto}1]$$

— the l in $l+1$ is carried by the strong substitution for X under the weak substitution $[l{\mapsto}1]$. This does not reduce further as-is, but a later lookup for l will return 2. Thus $\text{И}\lambda a_3.a_3[X{\mapsto}X[l{\mapsto}l+1]]$ implements a command 'increment the value stored at l'. This term clearly relies on *not* using protected records, since X is free in it (later, we show how to obtain the same effect for protected records).

We call this 'in-place update' because to increment l we do not build a new record which is a copy of the old one only with an incremented value; we actually directly modify the original item of data. This is significant from an implementational standpoint. In full generality we can replace X by any term we wish.

These constructions and our notation should remind us of objects in the sense of 'object-oriented programming'. First we consider global state, then we return to records to generalise them to objects.

4.5 Global state (with calculus)

Consider an untyped λ-calculus with general references. Terms are generated by the grammar

$$e ::= x \mid !l \mid l := e \mid \text{И}l.e \mid \texttt{skip} \mid ee \mid \lambda x.e.$$

Here l is drawn from a set of **location variables** and x from a set of **program variables**. The intended semantics of $!l$ is 'the value which the global state associates to l', of $l := e$ is 'set l to e in the global state', and of $\text{И}l.e$ is 'allocate a fresh local location (with some unspecified or default value associated) and evaluate e in the extended state'. The rest is standard as for the untyped λ-calculus.

This calculus is roughly similar to others in the literature, for example $\mathcal{L}$ [2, Fig. 1 and Fig. 2] and to ReFS [29, Fig. 1 and Fig. 4]. The focus of study for $\mathcal{L}$ is a fully abstract game semantics; that of ReFS is how to prove contextual equivalences between programs. Both $\mathcal{L}$ and ReFS are typed. For brevity we have considered an untyped calculus.

[2] contains a construction of the Y combinator using general references in a typed system, thus obtaining the power of recursion. Therefore the absence of types does not make the calculus above obviously more powerful than $\mathcal{L}$, though it is more powerful than ReFS, which has recursion but integer-only references.

An encoding in the context calculus is given as follows: Fix a level 3 variable $\mathbf{W}$, a level 2 variable $\mathbf{S}$, and a level 1 variable $\mathbf{r}$. Continuing the discussion of records above, write $t.l$ for $t[\mathbf{S}{\mapsto}l]$. Write $t.l := s$ for $t[\mathbf{S}{\mapsto}\mathbf{S}[l{\mapsto}s]]$. (In particular, we write $\mathbf{W}.\mathbf{r}$ for $\mathbf{W}[\mathbf{S}{\mapsto}\mathbf{r}]$.)

$$\begin{gathered}
[\![x]\!] = \text{И}\lambda\mathbf{W}.\mathbf{W}.\mathbf{r} := x \quad [\![!l]\!] = \text{И}\lambda\mathbf{W}.\mathbf{W}.l \\
[\![l := e]\!] = \text{И}\lambda\mathbf{W}.(\mathbf{W}.l := e).\mathbf{r} := \top \quad [\![\texttt{skip}]\!] = \text{И}\lambda\mathbf{W}.\mathbf{W}.\mathbf{r} := \top \\
[\![\lambda x.e]\!] = \text{И}\lambda\mathbf{W}, x.[\![e]\!]\mathbf{W} \\
[\![ee']\!] = \text{И}\lambda\mathbf{W}.[\![e]\!]([\![e']\!]\mathbf{W}).\mathbf{r} := \top)(([\![e']\!]\mathbf{W}).\mathbf{r})
\end{gathered}$$

The definitions above faithfully simulate the reductions we would expect of a language with references, and in particular those from [2], though we have no space to give them.

The reader may have noticed that our interpretation of global state is a 'state transformer'; $[\![e]\!]$ is a function which transforms states into other states:

4.6 Global state (with state-transformer)

As observed by Moggi [25], monads can model state in a purely functional setting. Following [18], and continuing notation from the last subsection, we define macros:

$$\begin{aligned}
\texttt{newVar} &= \text{И}\lambda x, \mathbf{W}.\text{И}l.\mathbf{W}.l := x \\
\texttt{readVar} &= \text{И}\lambda l, \mathbf{W}.\mathbf{W}.\mathbf{r} := (\mathbf{W}.l) \\
\texttt{writeVar} &= \text{И}\lambda l, x, \mathbf{W}.(\mathbf{W}.l := x).\mathbf{r} := \top \\
\texttt{runST} &= \lambda P.\text{И}\mathbf{W}.(P\mathbf{W}).\mathbf{r}
\end{aligned}$$

For detailed explanations see [18, §2.2, 2.4] and [17, 16]. Intuitively, $\texttt{newVar}x$ takes a state $\mathbf{W}$ (represented by a big hole!), allocates a new reference l in it, assigns l to x, and returns the new state. $\texttt{readVar}l$ takes a state $\mathbf{W}$ and extracts the value assigned to l to the distinguished result record $\mathbf{r}$. `writeVar` takes a reference l and a value x, and given a state $\mathbf{W}$ returns $\mathbf{W}$ with l allocated to x. If x evaluates to some term t containing $\texttt{readVar}(l, \mathbf{W})$ then l will be linked to the old value of l, before `writeVar` (this is not worthy of comment in [18] because the types make it quite clear how states are passed around). Finally, `runST` takes a program P, applies it to a fresh empty state, and returns the result.

4.7 Objects

We propose terms for *method invocation* and *method update* which behave like constructs of the same name in the **imp-ς** calculus of Abadi and Cardelli [1]. For brevity we must assume the reader has some familiarity with it.

- Call $\text{И}\lambda O_3, l_1.O_3O_3l_1$ **method invocation**. Write $O.l$ for it applied to terms O and l.
- Call $\text{И}\lambda O_3, P_3.\text{И}\lambda S_3, Y_2.\big(O_3S_3(Y_2[l_1{\mapsto}(P_3S_3)])\big)$ **method update (at l)** and write $O.l{\Leftarrow}P$ for it applied to terms O and P.

To see how these definitions relate to daily life and in particular to **imp-ς** say a term t is an (Abadi-Cardelli style) **object** when it has the form

$$\text{И}\lambda S_3, X_2.(X_2[l_1^1{\mapsto}s_1]\cdots[l_n^1{\mapsto}s_n]).$$

We have typeset the levels of the variables l_i as superscripts and not subscripts. S may occur in the syntax of the s_i and plays the rôle of the *self parameter* in [1]. We call the terms s_i the **data**.

Consider $t = \text{И}\lambda S, X.X[p{\mapsto}S][q{\mapsto}0]$. Then $t.p =$

$$\begin{aligned}
(\text{И}\lambda O, l.(OO)l)tp &\rightsquigarrow^* ttp \\
&\rightsquigarrow^* \text{И}S, X.((X[p{\mapsto}S][q{\mapsto}0])[S{\mapsto}t][X{\mapsto}p]) \\
&\rightsquigarrow^* \text{И}S, X.(X[p{\mapsto}t][q{\mapsto}0][X{\mapsto}p]) \rightsquigarrow^* t
\end{aligned}$$

Similarly, $t.q \rightsquigarrow^* 0$. For the same t, consider

$$\begin{aligned}
(t.p){\Leftarrow}(\text{И}\lambda S.\langle p, p\rangle) &\rightsquigarrow^* \text{И}\lambda S, Y.tS(Y[p{\mapsto}\langle p, p\rangle]) \\
&\rightsquigarrow^* \text{И}\lambda S, Y.(Y[p{\mapsto}\langle p, p\rangle][p{\mapsto}S][q{\mapsto}0])
\end{aligned}$$

We can verify that $((t.p){\Leftarrow}(\text{И}\lambda S.\langle p, p\rangle)).p \rightsquigarrow^* \langle t, t\rangle$.

Similarly, $((t.q){\Leftarrow}(\text{И}\lambda S.q + 1)).q \rightsquigarrow^* 1$.

Recalling the discussion of records above, we have a notion of protected record with a 'self' parameter with implementations of method invocation and update. Note a technical feature of our presentation is that S the self parameter is abstracted at top level, and not in every data item, for example $\text{И}\lambda X.X[p{\mapsto}\text{И}\lambda S.s_p]$, which would be more similar to **imp-ς**. Both ways are possible in our calculus and our choice seemed the simpler.

Note that **imp-ε** is a purpose-built abstract machine with a strong notion of global state whereas the context calculus is a purely functional rewrite system (so: no state and no control of evaluation order). So we have not 'implemented **imp-ε**'. We have shown that some of the behaviour it was designed to study exists in the context calculus. And this in a strong sense that for example a term 'method invocation' exists which when applied to a record s and a location l obtains in a suitable sense the data stored at l in s. There is no need to use a family of macros encoding 'method invocation for s at l' for each s and l.

We could continue with dynamic rebinding and mustering [5, 10] or modules and linking [20]. But now for something completely different:

4.8 Partial evaluation

Partial evaluation seeks strategies to optimally specialise code when some of the parameters are known. If s and t are programs (closed terms) then partial evaluation can be viewed as the search for algorithms to efficiently (in a suitable sense) compute an optimal (in a suitable sense) term

u which is equivalent (in a suitable sense) to st (s applied to t).

See [8] for a short but efficient survey of this huge field. The context calculus may have something to contribute; we use an example adapted from [8, §6.1]. Write

$$\texttt{if} = \lambda a, b, c.abc \quad \texttt{true} = \lambda ab.a \quad \texttt{false} = \lambda ab.b$$
$$\texttt{not} = \lambda a.\texttt{if}\ a\ \texttt{false}\ \texttt{true}.$$

in the untyped λ-calculus. It is easy to verify β-equivalences showing these terms model truth values in a suitable sense. Now suppose we want to calculate

$$s = \lambda f, a.\texttt{if}\ a\ (f\ a)\ a \qquad \text{specialised to} \qquad s\ \texttt{not}.$$

A naïve method β-reduces $s\texttt{not}$ and obtains a term β-equivalent to $\lambda a.\texttt{if}\ a\ (\texttt{not}\ a)\ a$ — a more intelligent method with access to a type system and which performs static analysis of dynamic values, may recognise that the program will always return `false`.

Now choose level 1 variables a, b and level 2 variables and B, C and define context calculus terms

$$\texttt{true} = \lambda ab.a \quad \texttt{false} = \lambda ab.b$$
$$\texttt{if} = \lambda a, B, C.\, a(B[a{\mapsto}\texttt{true}])(C[a{\mapsto}\texttt{false}])$$
$$\texttt{not} = \lambda a.\texttt{if}\ a\ \texttt{false}\ \texttt{true}.$$

Here B and C are variables which just happen to be stronger than a. This enables to directly encode in the term knowledge we have about execution flow, e.g. (very informally) that if we get to B, a must equal `true`.

If we want to partially evaluate

$$s = \lambda f, a.\texttt{if}\ a\ (f\ a)\ a \qquad \text{specialised to} \qquad s\ \texttt{not}.$$

we can naïvely reduce to obtain

$$\begin{aligned} s\ \texttt{not} &\rightsquigarrow^* \lambda a.a\ ((\texttt{not}B)[a{\mapsto}\texttt{true}][B{\mapsto}a])\ (C[a{\mapsto}\texttt{false}][C{\mapsto}a]) \\ &\rightsquigarrow^* \lambda a.a\ ((\texttt{not}a)[a{\mapsto}\texttt{true}])\ (a[a{\mapsto}\texttt{false}]) \\ &\rightsquigarrow^* \lambda a.(a\ \texttt{false}\ \texttt{false}). \end{aligned}$$

This is more efficient and with types a must be `true` or `false` so the term can be specialised to `false` by standard techniques. So with strong variables and explicit substitution we 'suspend information' about a on B and C.

5. CONFLUENCE

We use an attractive result in Van Oostrom's thesis [36, Thm 2.3.5], but we need some notation to state it.

Take X a set and $\rightsquigarrow \subseteq X \times X$ a **reduction relation** on it. In connection with X we shall *always* write $r = (x \rightsquigarrow y)$ and $s = (x \rightsquigarrow z)$ for pairs in $\rightsquigarrow$. Take $\preccurlyeq$ a well-founded partial order on reductions $r \preccurlyeq s$. We write $r \prec s$ when $r \preccurlyeq s$ and $s \not\preccurlyeq r$, and $r \approx s$ when $r \preccurlyeq s$ and $s \preccurlyeq r$.

Call a sequence $c_1 = (x_1 \rightsquigarrow x_2)$, $c_2 = (x_2 \rightsquigarrow x_3)$, $\ldots$, $c_{n-1} = (x_{n-1} \rightsquigarrow x_n)$, a **chain** and write it namelessly just as $x \rightsquigarrow^* y$ (n may equal 0). In the case that the chain has at length at most one (so $n \leqslant 1$) we write $x \rightsquigarrow^+ y$.

When $c_i \prec s$ for all elements of the chain, we write $(x \rightsquigarrow^* y) \prec s$. To assert the existence of such a chain, we write $x \overset{\prec s}{\rightsquigarrow^*} y$. Similarly for $\preccurlyeq$ and $\approx$. When there is some pair r and s of reductions such that at least one of $c_i \prec r$ or $c_i \prec s$ always holds, we write $(x \rightsquigarrow^* y) \prec \{r, s\}$, and to assert the existence of such a chain we write $(x \overset{\prec\{r,s\}}{\rightsquigarrow^*} y)$.

Given x and z, to assert the existence of two chains $x \overset{\phi}{\rightsquigarrow^*} y$ and $y \overset{\psi}{\rightsquigarrow^*} z$ (satisfying conditions ϕ and ψ as discussed above) for some y, we write namelessly $x \overset{\phi}{\rightsquigarrow^*} \overset{\psi}{\rightsquigarrow^*} z$.

Call a pair $r = (x \rightsquigarrow y)$ and $s = (x \rightsquigarrow z)$ a **local divergence**, and call a pair of chains $y \rightsquigarrow^* u$ and $z \rightsquigarrow^* u$ a **convergence**

THEOREM 5.1 (VAN OOSTROM). *Suppose for every pair of local divergences $r = (x \rightsquigarrow y)$ and $s = (x \rightsquigarrow z)$ there exists a convergence of the form $y \overset{\prec r}{\rightsquigarrow^*} \overset{\approx s}{\rightsquigarrow^+} \overset{\prec\{r,s\}}{\rightsquigarrow^*} u$ and $z \overset{\prec s}{\rightsquigarrow^*} \overset{\approx r}{\rightsquigarrow^+} \overset{\prec\{r,s\}}{\rightsquigarrow^*} u$. Then $\rightsquigarrow$ is confluent.*

The partial order we impose on reductions is as follows: Instances of...

- $(\sigma\sigma)$ are ranked according to the size of s:

$$s[a{\mapsto}u][b{\mapsto}v] \overset{(\sigma\sigma)}{\rightsquigarrow^*} s[b{\mapsto}v][a{\mapsto}u[b{\mapsto}v]] \prec \langle s, s\rangle[a{\mapsto}u][b{\mapsto}v] \overset{(\sigma\sigma)}{\rightsquigarrow^*} \langle s, s\rangle[b{\mapsto}v][a{\mapsto}u[b{\mapsto}v]].$$

- $(\sigma\lambda)$ and $(\sigma\lambda')$ are ranked according to the size of the context in which they occur:

$$(\lambda a.(s[c{\mapsto}u]))[d{\mapsto}v] \overset{(\sigma\lambda)}{\rightsquigarrow^*} \lambda a.(s[c{\mapsto}u][d{\mapsto}v]) \prec (\lambda a.s)[c{\mapsto}u][d{\mapsto}v] \overset{(\sigma\lambda)}{\rightsquigarrow^*} (\lambda a.(s[c{\mapsto}u]))[d{\mapsto}v]$$

- $(\sigma\mathbf{p})$ are ranked according to the size of $a_i t_1 \ldots t_n$.

- (β), $(\sigma\mathbf{a})$, $(\sigma\#)$, and $(\sigma\mathbf{tr})$, are ranked together inversely according to the size of the context in which they occur: $(s[b{\mapsto}a[a{\mapsto}u]])t \overset{(\sigma\mathbf{a})}{\rightsquigarrow^*} (s[b{\mapsto}u])t \prec (st)[b{\mapsto}a[a{\mapsto}u]] \overset{(\sigma\mathbf{a})}{\rightsquigarrow^*} (st)[b{\mapsto}u]$

- $(И\mathbf{p})$, $(И\mathbf{a})$, $(И\sigma)$, and $(И\notin)$, are ranked according to the size of the context in which they occur. For example, $(И a.s)t \overset{(И\mathbf{p})}{\rightsquigarrow^*} И a.(st) \prec u[b{\mapsto}(И a.s)t] \overset{(И\mathbf{p})}{\rightsquigarrow^*} u[b{\mapsto}И a.(st)]$

- $\preccurlyeq$ is reflexive; $s \preccurlyeq s$. Everything else is incomparable: $a[a{\mapsto}(\lambda b.s)t] \overset{(\sigma\mathbf{a})}{\rightsquigarrow^*} (\lambda b.s)t \not\preccurlyeq a[a{\mapsto}(\lambda b.s)t] \overset{(\beta)}{\rightsquigarrow^*} a[a{\mapsto}s[b{\mapsto}t]])$

The proof is now by the theorem above, verifying that every divergence has a suitable convergence. When closing a divergence involving scope-extrusion for $И$, $(И a.s)t \rightsquigarrow И a.(st)$ say, the side-condition $a \notin t$ does *not* ensures $a\#t$. However we may α-rename to some a' such that $a'\#t$; we assumed the context has enough freshnesses so this can be done.

Let $\leftrightarrow$ be the transitive symmetric reflexive closure of $\rightsquigarrow$.

COROLLARY 5.2. *$\leftrightarrow$ does not relate all terms to all other terms (we say the calculus is **consistent**).*

PROOF. If s and t are two terms and $s \leftrightarrow t$ then by confluence there exists some u with $s \rightsquigarrow u$ and $t \rightsquigarrow u$. To prove consistency, it suffices to exhibit two different normal forms; for example $\lambda a_1.a_1$ and $\lambda a_2.a_2$ (or, if the reader prefers, $\lambda a_1, b_1.a_1$ and $\lambda a_1, b_1.b_1$). □

6. HINDLEY-MILNER TYPES

Some basic motivation: a type system is a logic (often a decidable logic) on terms, which allows us to reason on terms without having to evaluate them. For example, in ML if a term types has type integer, by Subject Reduction the term will always have type integer no matter how we evaluate it, and if it reduces to a normal form that normal form will *be* an integer. So there is no 'one' type system; it depends what properties we are interested in.

Hindley-Milner typing [9] is a simple and successful polymorphic type system which underlies ML. As such it is both a 'working (functional) programmer's' tool and a starting point for more complex schemes. If the context calculus interacts well with it, then it is that bit less difficult to implement, e.g. as an extension of ML.

Fix infinitely many **type variables** $\alpha, \beta \in \mathrm{TyVar}$. **Types** and **type schemes** are defined by:

$$\tau ::= \alpha \mid (\tau, \tau) \mid \tau \to \tau \qquad \sigma ::= \tau \mid \forall\alpha.\ \sigma.$$

Let a **type substitution**, we generally write S or T, be a function from type variables α to types τ such that $S\alpha = \alpha$ for all but finitely many α. Type substitutions act on types in the standard way. Write $\tau \preceq \sigma$ when $\sigma = \forall\alpha_1.\ \cdots \forall\alpha_n.\ \tau'$ (we shall just write $\sigma = \forall\overline{\alpha}.\ \tau$ here and $\tau = S\tau'$. Also write $tyv\tau$ for the type variables appearing in τ. Write Γ, Δ for (finite) **type contexts** in the standard sense.

Then typing rules are as follows:

$$\frac{x : \sigma \in \Gamma \quad \tau \preceq \sigma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, a_i : \tau \vdash s : \tau'}{\Gamma \vdash \lambda a_i.s : \tau \to \tau'}$$

$$\frac{\Gamma \vdash s' : \tau' \quad \Gamma, a_i : \forall\overline{\alpha}.\ \tau' \vdash s : \tau \quad \overline{\alpha} = tyv\tau'\backslash tyv\Gamma}{\Gamma \vdash s[a_i{\mapsto}s'] : \tau}$$

$$\frac{\Gamma, n_j : \alpha \vdash s : \tau \quad n_j, \alpha \notin \Gamma}{\Gamma \vdash \text{И}n_j.s : \tau} \qquad \frac{\Gamma \vdash s : \tau \to \tau' \quad \Gamma \vdash t : \tau}{\Gamma \vdash st : \tau'}$$

LEMMA 6.1 (WEAKENING FOR TYPE JUDGEMENTS). *If* $\Gamma \vdash s : \tau$ *then* $\Gamma, \Delta \vdash s : \tau$.

PROOF. By induction on the derivation. □

THEOREM 6.2 (SOUNDNESS). *If* $\Gamma \vdash s : \tau$ *and* $s \rightsquigarrow s'$ *then* $\Gamma \vdash s' : \tau$.

PROOF. We exhaustively check all rules for $\rightsquigarrow$ (we consider just two cases).

Suppose $\Gamma \vdash (\lambda n_j.t)[a_i{\mapsto}u] : \tau \to \tau'$ is derivable. By following the derivation rules we see that

$$\Gamma \vdash u : \tau'' \quad \text{and} \quad \Gamma, a_i : \forall\overline{\alpha}.\ \tau'', n_j : \tau \vdash t : \tau'$$

must be derivable, where $\alpha = tyv\tau''\backslash tyv\Gamma$. Without loss of generality we rename elements of α to be disjoint from $tyv\tau$. It is now not hard to derive $\Gamma \vdash \lambda n_j.(t[a_i{\mapsto}u]) : \tau \to \tau'$, using weakening for type judgements above to weaken $\Gamma \vdash u : \tau''$ to $\Gamma, n_j : \tau \vdash u : \tau''$.

Suppose $(\text{И}n_j.s)t \rightsquigarrow \text{И}n_j.(st)$ and $\Gamma \vdash (\text{И}n_j.s)t$ is derivable. Then for some τ', $\Gamma, n_j : \alpha \vdash s : \tau \to \tau'$ and $\Gamma \vdash t : \tau$, where $\alpha \notin \Gamma$. It is now easy to derive $\Gamma \vdash \text{И}n_j.(st) : \tau'$. □

We use the following technical lemma in the theorem which follows:

LEMMA 6.3. *If* $\Gamma \vdash s : \tau$ *then* $S\Gamma \vdash s : S\tau$.

PROOF. By induction on the derivation of $\Gamma \vdash s : \tau$. □

THEOREM 6.4. *For all* Γ *and* s *there exists a pair* (S, τ) *such that* $S'\Gamma \vdash s : \tau'$ *if and only if* $(S', \tau') \preceq (S, \sigma)$.

PROOF. (S, τ) is calculated by the following algorithm — the rules are read bottom-up, and we write $\Gamma \vdash_? s$ for the pair (S, τ) which is being calculated:

$$\frac{(\Gamma, a_i : \alpha \vdash_? s) = (S, \tau)}{(\Gamma \vdash_? \lambda a_i.s) = (S, S\alpha \to \tau)}$$

$$\frac{\begin{array}{ll}(\Gamma \vdash_? s) = (S, \tau) & S''{=}mgu(S'\tau, \tau' \to \alpha) \\ (S\Gamma \vdash_? s') = (S', \tau') & \alpha{\notin}S, S', \Gamma, s, s'\end{array}}{(\Gamma \vdash_? ss') = (S''S'S, S''\alpha)}$$

$$\frac{\begin{array}{l}(\Gamma \vdash_? s') = (S', \tau') \quad \overline{\alpha}{=}tyv\tau'\backslash tyv(S', \Gamma) \\ (S'\Gamma, a_i : \forall\overline{\alpha}.\ \tau' \vdash_? s) = (S, \tau)\end{array}}{(\Gamma \vdash_? s[a_i{\mapsto}s']) = (SS', \tau)}$$

$$\frac{(\Gamma, a_i : \alpha \vdash_? s) = (S, \tau) \quad \alpha \notin \Gamma}{(\Gamma \vdash_? \text{И}a_i.s) = (S, \tau)} \qquad \frac{a_i : \forall\overline{\alpha}.\ \tau \in \Gamma}{\Gamma \vdash a_i : \tau}$$

We prove by induction on the syntax of s that for all Γ, if $(\Gamma \vdash_? s) = (S, \tau)$ and $\overline{\alpha} = tyv\tau\backslash tyv(S\Gamma)$ then $(S, \forall\overline{\alpha}.\ \tau)$ is a principal solution to $(\Gamma \vdash_? t)$.

The case $t = a_i$ ... is simple.

Suppose $t = ss'$ ... is complex but standard straight out of [9] .

Suppose $t = \text{И}a_i.s$ Let (S, τ) and α be as in the inference rule, so that $S\Gamma, a_i : \alpha \vdash s : \tau$. Suppose $T\Gamma \vdash \text{И}a_i.s : \mu$. Following the typing rules, $T\Gamma, a_i : \alpha \vdash s : \mu$. By the universal property of (S, τ), $T = US$ and $\mu = U\tau$ for some U. The case $t = s[a_i{\mapsto}s']$ is also simple. □

In conclusion, the type system is almost completely standard Hindley-Milner polymorphic types. The typing rule for explicit substitutions is identical to that of ML `let`-statements, so we are justified in viewing the explicit substitution as a `let`, and viewing the context calculus as an extension of a core of ML with a hierarchy of context variables (and a separation of abstraction and binding).

7. APPLICATIVE CHARACTERISATION OF CONTEXTUAL EQUIVALENCE

It is interesting to ask what an appropriate theory of contextual equivalence between programs is, since our system has programs which are contexts.

7.1 Programs, contexts, evaluations, and equivalences

It is convenient to introduce a constant $\top$. $a\#\top$ always and $\top$ engages in no reductions (e.g. $\top s$ is a normal form for all s).

Call a term s with no free variables a **program**. $\lambda x.x$ is not a program, and $\text{И}\lambda x.x$ and $\text{И}x.x$ are. (Recall: we write $\text{И}\lambda x.s$ for $\text{И}x.\lambda x.x$.)

If s is a program write $s{\searrow}$ when $s \rightsquigarrow^* \top$ and say s **evaluates**.

Say C is a **context** when *(a)* C is a program and *(b)* $C = \text{И}\lambda d_l.D$ where d_l is stronger than all other variables in D. We may abuse notation and also call D a **context**.

For example $\text{И}\lambda X_2.X_2$, $\text{И}f_1.\text{И}\lambda X_2.\lambda f_1.(f_1X_2)X_2$, and $\text{И}\lambda X_2.\top$ are contexts. $\lambda X.a_1$ and $\lambda X_2.X_2$ are not a contexts since they are not closed. $\text{И}\lambda X_2.\text{И}\lambda Y_2.Y_2X_2$ is not a

context because Y_2 has level 2 and so does X_2, so X_2 is not strictly stronger than Y_2.

An **equivalence relation** is a transitive symmetric reflexive relation. Call an equivalence relation $\mathcal{X}$ on programs **contextual** (or **a congruence**) when

$$\forall s,t.\ s\ \mathcal{X}\ t \Rightarrow \forall C.\ Cs\ \mathcal{X}\ Ct \qquad \forall s,t.\ s\ \mathcal{X}\ t \wedge s{\searrow} \Rightarrow t{\searrow}$$

Here C varies over contexts. Write $=_{ctx}$ for the greatest contextual equivalence, which (abusing notation) we call **contextual equivalence**. We discuss this definition below.

Call an equivalence relation $\mathcal{P}$ on programs **applicative** when

$$\forall s,t.\ s\ \mathcal{P}\ t \Rightarrow \forall u.\ su\ \mathcal{P}\ tu \qquad \forall s,t.\ s\ \mathcal{P}\ t \wedge s{\searrow} \Rightarrow t{\searrow}.$$

Here u varies over programs. Write $=_{ap}$ for the greatest applicative equivalence and abuse notation calling it **applicative equivalence**.

The main result of this section is

THEOREM 7.1. *$=_{ctx}$ and $=_{ap}$ are equal.*

. . . but we need some technical machinery to prove it:

LEMMA 7.2 (TECHNICAL LEMMA). *Write $s \leftrightarrow t$ when s and t are related by the transitive symmetric closure of $\rightsquigarrow$. Then $\leftrightarrow \subseteq =_{ap}$ and $\leftrightarrow \subseteq =_{ctx}$.*

Also, $\top s_1 \dots s_n =_{ap} \top t_1 \dots t_n$ always, for $n > 0$.

PROOF. Suppose $s \leftrightarrow t$. By confluence, $su{\searrow}$ if and only if $tu{\searrow}$, and similarly for Cs and Ct. The second part follows from the observation that $\top s_1 \dots s_n{\searrow}$ is impossible unless $n = 0$. □

Note that when we define $=_{ctx}$ we consider C *applied* to s. From the first part of the technical lemma above and from the definition of context, this is clearly equivalent to a definition in more traditional form $D[d_l \mapsto s]$. In the proof below we tend to use traditional notation of $D[d_l \mapsto s]$ rather than $(\lambda d_l.D)s$.

7.2 Proof that $=_{ctx}$ equals $=_{ap}$

PROOF ($=_{ctx}$ EQUALS $=_{ap}$). It is easy to show that $=_{ctx}$ implies $=_{ap}$.

Conversely suppose $s =_{ap} t$. We work by induction on the tuple (oc_D, nf_D, si_D) where

- oc_D is the number of occurrences of d_l in the normal form of D and ω otherwise (=the first uncountable ordinal; if D has no normal form there will be nothing to prove),
- nf_D is the least number of $\rightsquigarrow$-reductions to reduce D to its normal form and ω otherwise, and
- si_D is the size of D,

proving that

$$\{(\text{И}as.D[d_l \mapsto s], \text{И}as.D[d_l \mapsto t]) \mid D, d_l, as\}$$

is a contextual relation, where as varies over possibly empty lists of variables no stronger than d_l. We work by cases on the form of D.

In the the cases below we may implicitly use the first part of the technical lemma above, along with confluence and the inductive hypothesis, to suppose that D is in $\rightsquigarrow$-normal form. We also silently strip leading Иs, writing for example d_l for $\text{И}as.d_l$.

Suppose $D = d_l$. Then $d_l[d_l \mapsto s] \leftrightarrow s =_{ap} t \leftrightarrow d_l[d_l \mapsto t]$. We use the first part of the technical lemma above and the fact that $=_{ap}$ is by construction an equivalence.

Suppose $D = (\lambda n_j.D')$. Then $D[d_l \mapsto s]$ and $D[d_l \mapsto t]$ cannot evaluate to $\top$ so there is nothing to prove.

Suppose $D = D'D''$, and suppose d_l occurs in D' and D''. We reason as follows:

$$\begin{array}{lcl} (D'D'')[d_l \mapsto s] & \leftrightarrow & (D'[d_l \mapsto s])(D''[d_l \mapsto s]) \\ & =_{ap} & (D'[d_l \mapsto t])(D''[d_l \mapsto s]) \\ & \leftrightarrow & (\lambda x_1.x_1 D'')[d_l \mapsto s](D'[d_l \mapsto t]) \\ & =_{ap} & (\lambda x_1.x_1 D'')[d_l \mapsto t](D'[d_l \mapsto t]) \\ & \leftrightarrow & (D'D'')[d_l \mapsto t] \end{array}$$

Here x_1 is chosen fresh. We use the inductive hypothesis for D' and $\lambda x_1.(x_1 D'')$, both of which have fewer occurrences of d_l then $D'D''$. Then $D[d_l \mapsto s] =_{ap} D[d_l \mapsto t]$ follows using the technical lemma and the fact that $=_{ap}$ is by construction an equivalence and also closed under application on the right.

Suppose $D = D'D''$, and suppose d_l occurs in D' but not D''. We reason as follows:

$$\begin{array}{lcl} (D'D'')[d_l \mapsto s] & \leftrightarrow & (D'[d_l \mapsto s])D'' \\ & =_{ap} & (D'[d_l \mapsto t])D'' \\ & \leftrightarrow & (D'D'')[d_l \mapsto t] \end{array}$$

Suppose $D = D'D''$, and suppose d_l does not occur in D' and may or may not occur in D''. Then D' is closed. If D' is a λ-abstraction and $D'D''$ reduces with (β) so we use the inductive hypothesis. Otherwise, $D'D''$ cannot ever evaluate and there is nothing to prove.

Suppose $D = D'[n_j \mapsto D'']$. If this is not a normal form, we reduce and use the inductive hypothesis. If this is a normal form we reason by cases. If d_l occurs in D' and D'', or in D' and not in D'', we can proceed as we did for applications above.

Suppose d_l does not occur in D' and does occur in D''. We have supposed this is a normal form, so (checking cases) it must be that:

- $D' = \lambda e.E$ for some e and E, where $e\#D''$ does not hold but $e\#n_j$ does (so $(\sigma\lambda)$ does not apply). Then $D'[n_j \mapsto D''][d_l \mapsto u]$ cannot possibly evaluate to $\top$ for any u.
- $D' = \text{И}e.E$, where $e\#D''$ does not hold but $e\#n_j$ does. Since И binds, e can always be renamed to some atom such that $e\#D''$ does hold. Therefore, this is not a normal form.
- $D' = n_j$ and for some e, $e\#D''$ does not hold but $e\#n_j$ does.

  d_l is stronger than n_j and because n_j is assumed bound by a top-level И which we silently omitted, we can assume that $n_j\#d_l$ does not hold. Then for $u \in \{s,t\}$ we can reduce as follows:

$$\begin{array}{lcl} n_j[n_j \mapsto D''][d_l \mapsto u] & \overset{(\sigma\sigma)}{\rightsquigarrow} & n_j[d_l \mapsto u][n_j \mapsto D''[d_l \mapsto u]] \\ & \overset{(\sigma \mathbf{b})}{\rightsquigarrow} & n_j[n_j \mapsto D''[d_l \mapsto u]]. \end{array}$$

Recall that s and t are both closed so $\forall a.\ a\#s$ and $\forall a.\ a\#t$. By the technical lemma below, $e\#D''[d_l \mapsto s]$ if and only if $e\#D''[d_l \mapsto t]$. Write u for s and/or t. If $e\#D''[d_l \mapsto u]$ do not hold then $n_j[n_j \mapsto D''[d_l \mapsto u]]$ do not reduce for $u \in \{s,t\}$ and there is nothing to prove. If $e\#D''[d_l \mapsto u]$ do hold then

$n_j[n_j{\mapsto}D''[d_l{\mapsto}u]] \rightsquigarrow D''[d_l{\mapsto}u]$. D'' is in normal form, has the same number of occurrences of d_l as $n_j[n_j{\mapsto}D'']$, and is strictly smaller. So we use the inductive hypothesis. □

Note how the most fiddly case is $n_j[n_j{\mapsto}D'']$, which might look 'simplest'. But of course, this is the crunch case, where calculation actually gets done.

LEMMA 7.3 (TECHNICAL LEMMA). *Suppose d is a variable and s and t are terms. Suppose $a\#s$ if and only if $a\#t$ for all variables (a variable is fresh for s if and only if it is fresh for t). Then $a\#E[e{\mapsto}s]$ if and only if $a\#E[e{\mapsto}t]$, for all terms E and variables e.*

PROOF. By routine induction on the syntax of E. □

It may be that this result can be 'lifted' to calculi translated into the context calculus. We only need check that at points in the proof above where we convert contexts into applicative contexts, every manipulation keeps us within the image of the translation. We have in mind results similar to those by Pitts [29] characterising contextual equivalence for languages with global state, or perhaps analogous results for MetaML (see the Conclusions). We must defer this for a later paper.

8. CONCLUSIONS AND FUTURE WORK

Our NEW calculus of contexts is a little different, because of its freshness contexts and separation of abstraction (λ) and binding ($\mathsf{И}$) — but it is not so complicated and has standard (and nontrivial) meta-properties motivated in the introduction. We have programmed interesting things in it including of course contexts, but also objects, general references, and we have made some observations about tracing values variables must have as a construction encoded directly in terms, which we applied to partial evaluation. We do not believe that this exhausts the possibilities.

We now explore what we *cannot* do in the calculus, since it is a prototype and there is plenty of ways to extend it.

The $\mathsf{И}$ context calculus can express 's with x substituted for t' as $s[x{\mapsto}t]$; this is a primitive of the calculus. We can abstract over s and t to obtain $\mathsf{И}\lambda P, X.X[x{\mapsto}P]$ (X stronger than x, and P stronger than X); call this term **substitution for** x, because $(\mathsf{И}\lambda P, X.X[x{\mapsto}P])tx \rightsquigarrow^* s[x{\mapsto}t]$. However, we cannot abstract over x;

$$
\begin{aligned}
&(\mathsf{И}\lambda P, X, x.X[x{\mapsto}P])tsy \\
&\quad \overset{x,P,X\notin t,s,y}{\rightsquigarrow^*} \mathsf{И}x, P, X.X[x{\mapsto}P][P{\mapsto}t][X{\mapsto}s][x{\mapsto}y] \\
&\qquad \rightsquigarrow \mathsf{И}x, P, X.X[x{\mapsto}t][X{\mapsto}s][x{\mapsto}y] \\
&\qquad\quad \rightsquigarrow^* \mathsf{И}x.s[x{\mapsto}t][x{\mapsto}y] \\
&\qquad\qquad \overset{(\sigma\#)}{\rightsquigarrow} \mathsf{И}x.s[x{\mapsto}t] \overset{(\sigma\#),(\mathsf{И}\notin)}{\rightsquigarrow} s.
\end{aligned}
$$

This is not the intended operational behaviour. This is no surprise: variables can be substituted for at any time, so it should not possible to pass a variable as a first-class value. However we believe things are not quite so clear-cut now because of the context Γ, into which we already put assertions such as $a\#X$ (X is a hole that should not contain a). We may be able to also place assertions such as **var**a, meaning 'a is a variable'. If that seems far-fetched, the reader can look at [12] where the author carries out a similar exercise to the one described in the context of first-order logic.

The $\mathsf{И}$ context calculus cannot express 'substitute all variables of level 1 for *blah* in s', which we might write as $s[\star{\mapsto}blah]$. This idea appears in work by Dami [10] on *dynamic binding*. We do believe that our context calculus could have something to say about dynamic binding and linking, but we leave that for future work.

Now we recall the problem of implementing proof-search in first-order logic. The sequent right-introduction rules for $\forall$ and $\exists$ are [4]:

$$\frac{\Gamma \vdash P[y{\mapsto}x]}{\Gamma \vdash \forall y.\ P} \qquad \frac{\Gamma \vdash P[y{\mapsto}t]}{\Gamma \vdash \exists y.\ P}$$

Here $x \notin \Gamma, P$ and t is any term. This can be a problem, for example for the design of a theorem-prover such as Isabelle [27], since it may become apparent only further up in the proof-tree (if at all) which t is appropriate. Isabelle uses, instead of t, an **existential variable** which behaves much like a level 2 variable only function-application is used to control scope.

We can certainly model existential variables using variables of level 2; for example $\exists ?t.\ P(?t)$ is modelled by a term of the form $\exists \lambda X.PX$ where $\exists$ is some constructor. Substitutions on X can be discovered deep in a proof, e.g. when we arrive at a leaf $\Gamma \vdash X = 1$, and the substitution can be passed upwards and applied directly to the proof-state as a whole to specialise X. However, there is no way to write a unification algorithm in the $\mathsf{И}$ context calculus precisely because variables can be instantiated randomly by explicit substitutions from outside.

It is of significant long-term interest to present proof-search and programming (if the reader prefers; unification and tactics) in a unified framework, so we should definitely consider whether anything can be done here. We can introduce judgements of the form **var**a into the context, and constructors into the calculus which detect when a variable is 'known' to be a variable in that sense. We can impose types and build the terms of the logic in an inductive datatype. A more-or-less strict evaluation order would avoid issues with confluence. We could try to solve only higher-order patterns [23], or restrict substitutions to be fresh renamings (substitutions of atoms for fresh atoms, similar in power to FM swapping [13]). Nominal Unification [35] already implements an algorithm which can be viewed as a special case of unifiers and the author's a-logic [12] introduces a logical judgement 'this *is* a variable' into first-order logic; so there is hope.

(We should mention that even in purely propositional logic, the Cut rule can be programmed in a similar way using a meta-variable for the formula introduced by the cut. So the issues just discussed arise in many logics and on many levels.)

We mention **staged computation**, as in for example MetaML [26, 28], Template Haskell [32], and Converge [34]. These languages offer a program enough control of its own execution that it can suspend its own execution, compose suspended programs into larger (suspended) programs, pass suspended programs as arguments to functions, and evaluate them. This raises issues similar to those surrounding contexts.

The context calculus cannot model staged computation, simply because it is a pure rewrite system with no control of evaluation order. *However*, it is possible to design a *programming language* based on the $\mathsf{И}$ context calculus which

can model staged computation. We have explored the idea in detail but have no space to do more than indicate the idea: in the term $s[a{\mapsto}t]$, restrict evaluations in t to those involving variables that are at least as strong as a. So for example $a_3[a_1{\mapsto}(\lambda a_2.1)0] \rightsquigarrow a_3[a_1{\mapsto}1]$, but $a_3[a_2{\mapsto}(\lambda a_1.1)0]$ does not reduce. $a_3[a_2{\mapsto}(\lambda a_2.\lambda a_1.a_2)11 \rightsquigarrow^* a_3[a_2{\mapsto}\lambda a_1.1]$, because a_2 is strong enough to reduce under a substitution by a_2, but a_1 is not. This to give enough control of execution flow to encode the *brackets*, *escape*, and *run* of MetaML (as well as other less exotic constructs, such as call-by-name and call-by-value versions of function application).

This programming language is as expressive as the M context calculus; we just have control of execution flow. We expect it has the same good meta-properties as the smaller and simpler calculus of this paper, but proving that is for future work. We can hope the language might provide a semantics and common intermediate language for comparing additional features related to control of execution flow.

Other obvious extensions of the calculus include enriching the structure and treatment of levels. Finally, it is interesting to ask what kind of abstract machine is suitable for executing the M context calculus (or the staged version discussed above). We have some promising results in this direction but they too must remain for another paper.

9. REFERENCES

[1] Martín Abadi and Luca Cardelli, *An imperative object calculus*, TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development, LNCS, vol. 915, 1995, pp. 471–485.

[2] S. Abramsky, K. Honda, and G. McCusker, *A fully abstract game semantics for general references*, Proc. 13th IEEE Symp. Logic in Comp. Sci., IEEE Computer Society Press, 1998, pp. 334–344.

[3] H. P. Barendregt, *The lambda calculus: its syntax and semantics (revised ed.)*, Studies in Logic and the Foundations of Mathematics, vol. 103, North-Holland, 1984.

[4] John Bell and Moshé Machover, *A course in mathematical logic*, North-Holland, 1977.

[5] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough, *Dynamic rebinding for marshalling and update, with destruct-time lambda*, Proceedings of ICFP 2003: the 8th ACM SIGPLAN International Conference on Functional Programming (Uppsala), August 2003, pp. 99–110.

[6] Roel Bloo and Kristoffer Høgsbro Rose, *Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection*, CSN-95: Computer Science in the Netherlands, 1995.

[7] Mirna Bognar, *Contexts in lambda calculus*, Ph.D. thesis, Vrije Universiteit Amsterdam, 2002.

[8] Charles Consel and Olivier Danvy, *Partial evaluation: Principles and perspectives*, Journées Francophones des Langages Applicatifs, February 1993, pp. 493–501.

[9] Luis Damas and Robin Milner, *Principal type-schemes for functional programs*, POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, 1982, pp. 207–212.

[10] Laurent Dami, *A lambda-calculus for dynamic binding*, Theoretical Comp. Sc. **192(2)** (1998), 201–231.

[11] Maribel Fernández and Murdoch J. Gabbay, *Extensions of nominal rewriting*, PPDP, 2005.

[12] Murdoch J. Gabbay, *a-logic*, Submitted, 2005.

[13] Murdoch J. Gabbay and A. M. Pitts, *A new approach to abstract syntax with variable binding*, Formal Aspects of Computing **13** (2001), 341–363.

[14] M. Hamana, *Free sigma-monoids: A higher-order syntax with metavariables*, The Second Asian Symposium on Programming Languages and Systems (APLAS 2004), Lecture Notes in Computer Science, vol. 3202, 2004, pp. 348–363.

[15] Masatomo Hashimoto and Atsushi Ohori, *A typed context calculus*, Theor. Comput. Sci. **266** (2001), no. 1-2, 249–272.

[16] Peyton Jones, *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell*, Engineering theories of software construction, Marktoberdorf Summer School, NATO ASI, 105 Press, 2001, pp. 47–96.

[17] Simon L. Peyton Jones and Philip Wadler, *Imperative functional programming*, Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, 1993, pp. 71–84.

[18] John Launchbury and Simon L. Peyton Jones, *Lazy functional state threads*, PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, ACM Press, 1994, pp. 24–35.

[19] Pierre Lescanne, *From lambda-sigma to lambda-upsilon a journey through calculi of explicit substitutions*, POPL, 1994, pp. 60–69.

[20] Elena Machkasova and Franklyn A. Turbak, *A calculus for link-time compilation*, Lecture Notes in Computer Science **1782** (2000), 260–274.

[21] Ian Mackie Maribel Fernández, Murdoch J. Gabbay, *Nominal rewriting*, Submitted, January 2004.

[22] Takafumi Sakurai Masahiko Sato and Rod Burstall, *Explicit environments*, Fundamenta Informaticae **45:1-2** (2001), 79–115.

[23] Dale Miller, *A logic programming language with lambda-abstraction, function variables, and simple unification*, Extensions of Logic Programming (1991), no. 475, 253–281.

[24] Robin Milner, Joachim Parrow, and David Walker, *A calculus of mobile processes, II*, Information and Computation **100** (1992), no. 1, 41–77.

[25] Eugenio Moggi, *Notions of computation and monads*, Inf. Comput. **93** (1991), no. 1, 55–92.

[26] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard, *An idealized metaml: Simpler, and more expressive*, ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems, LNCS, vol. 1576, 1999, pp. 193–207.

[27] Larry Paulson, *The Isabelle reference manual*, Cambridge University Computer Laboratory, February 2001.

[28] A. M. Pitts and T. Sheard, *On the denotational semantics of staged execution of open code*, Submitted, February 2004.

[29] A. M. Pitts and I. D. B. Stark, *Operational reasoning for functions with local state*, Higher Order Operational Techniques in Semantics (A. D. Gordon and A. M. Pitts, eds.), Publications of the Newton Institute, Cambridge University Press, 1998, pp. 227–273.

[30] Masahiko Sato, Takafumi Sakurai, and Yukiyoshi Kameyama, *A simply typed context calculus with first-class environments*, Journal of Functional and Logic Programming **2002** (2002), no. 4, 359 – 374.

[31] Masahiko Sato, Takafumi Sakurai, Yukiyoshi Kameyama, and Atsushi Igarashi, *Calculi of meta-variables*, Computer Science Logic and 8th Kurt Gödel Colloquium (CSL'03 & KGC), Vienna, Austria. Proccedings (M. Baaz, ed.), Lecture Notes in Computer Science, vol. 2803, 2003, pp. 484–497.

[32] Tim Sheard and Simon Peyton Jones, *Template metaprogramming for Haskell*, ACM SIGPLAN Haskell Workshop 02 (Manuel M. T. Chakravarty, ed.), ACM Press, October 2002, pp. 1–16.

[33] Francois Maurel Sylvain Baro, *The qnu and qnuk calculi : name capture and control*, Tech. report, Université Paris VII, March 2003, Extended Abstract, Prépublication PPS//03/11//n16.

[34] Laurence Tratt, *Compile-time meta-programming in converge*, Tech. Report TR-04-11, Department of Computer Science, King's College London, December 2002.

[35] C. Urban, A. M. Pitts, and Murdoch Gabbay, *Nominal unification*, Theoretical Computer Science **323** (2004), 473–497.

[36] Vincent van Oostrom, *Confluence for abstract and higher-order rewriting*, Ph.D. thesis, Vrije Universiteit, Amsterdam, March 29 1994.