Multi-level Contextual Type Theory

Mathieu Boespflug Brigitte Pientka
Mcgill University
Montreal, Canada
{mboes,bpientka}@cs.mcgill.ca

Contextual type theory distinguishes between bound variables and meta-variables to write potentially incomplete terms in the presence of binders. It has found good use as a framework for concise explanations of higher-order unification, characterize holes in proofs, and in developing a foundation for programming with higher-order abstract syntax, as embodied by the programming and reasoning environment Beluga. However, to reason about these applications, we need to introduce meta^2-variables to characterize the dependency on meta-variables and bound variables. In other words, we must go beyond a two-level system granting only bound variables and meta-variables.

In this paper we generalize contextual type theory to n levels for arbitrary n, so as to obtain a formal system offering bound variables, meta-variables and so on all the way to meta^n-variables. We obtain a uniform account by collapsing all these different kinds of variables into a single notion of variabe indexed by some level k. We give a decidable bi-directional type system which characterizes $\beta\eta$-normal forms together with a generalized substitution operation.

1 Introduction

A core problem when describing computations and proofs is the need to model unknown entities. The standard approach is to introduce meta-variables that one can use in place of concrete evidence that might not yet be available. Consider for example the development of the proof for $\forall x.\exists y.P(x,x) \wedge Q(x,x) \supset Q(y,x) \wedge P(x,y)$ in a proof assistant. Working from the goal formula, we first introduce a parameter a and subsequently introduce a meta-variable Y for y which may depend on a. We can describe the intermediate subgoal we must now solve as: $P(a,a) \wedge Q(a,a) \supset Q(Ya,a) \wedge P(Ya,a)$. At a later point in the proof, we may realize through (higher-order) unification that a is a good instantiation for Y giving us an trivially provable goal. The question we address in this paper is how to describe formally the incomplete proof state we are in prior to finding instantiations for Y. Clearly, the missing proof term we want to construct depends on the meta-variable Y and the parameter a bound in the context. We hence need to introduce meta^2-variables to describe it.

A similar situation arises in the Beluga programming and reasoning environment [16; 17]. Recursive programs in Beluga analyze and manipulate meta-objects of type $A[\Psi]$, i.e. objects which have type A in a bound variable context Ψ. For example, `[x:i] allI λy. andI (F x y) (F y x)` describes the derivation of the formula $\forall y.P(y,x) \wedge P(x,y)$ where `F` itself stands in lieu of a description of the derivation which ends in $P(y,x)$ in the context `[x:i,y:i]`. Note that the meta-variable `F` is bound: if we pattern match on the LF object, then `F` is introduced and bound in the branch or it is bound explicitely at the outside by an abstraction. We hence have two different kinds of bound variables in the LF object. In Beluga, we can write underscores anywhere in an LF object and let type-reconstruction find the correct instantiation. For example, to describe an incomplete derivation where we omit the second argument to `andI`, we may write `[x:i] allI λy. andI (F x y) _` . During type reconstruction, the underscore will be replaced by a meta^2-variable to express the fact that we may use the meta-variable `F` or the bound variables `x` and `y`.

To appear in EPTCS.

Contextual type theory [14] provides both bound variables and meta-variables, complete with a logical foundation for reasoning about them. Up to now it has been used to explain higher-order unification [18; 2], characterize concisely holes in proofs, and develop a foundation for programming with higher-order abstract syntax as found in the Beluga programming and reasoning environment [16; 17]. This paper generalizes and extends contextual type theory [14] to an arbitrary number of levels of variables. Bound variables are of level 0, meta-variables are of level 1, meta^2-variables are of level 2, and so on and so forth. This leaves us with a uniform treatment of contexts, variables and their associated substitution operations. Unlike earlier work sketched by Pfenning [15] for the simply typed case, we enforce that the context is ordered, i.e. if $n > m$, then variables of level n occur to the left of variables of level m. This will naturally enforce the correct dependency: variables of the higher level n cannot depend on the variables of lower level m. We give a bi-directional type system to characterize β-η-long normal forms and generalize the hereditary substitution operation to variables of arbitrary level. We prove the hereditary substitution to be terminating, prove that typing preserves the well-formedness of ordered contexts, and show bi-directional typing to be decidable for the multi-level system.

This work is one step of the way towards streamlining and simplifying the implementation of Beluga, where we currently distinguish between bound variables, meta-variables, and meta^2-variables. But more generally, this work can be used to formalize incomplete proofs that manipulate open proof objects containing meta-variables. This is important to scale tactic languages such as VeriML [20] where we manipulate meta-objects that may contain bound variables, or to reason about the tactics themselves. We envision down the line a multi-level Beluga, which would allow us to reason about and manipulate Beluga programs within Beluga itself. This will provide a uniform framework where the proofs, the development of proofs using tactics, and the reasoning about tactics all share a common basis and supporting implementation.

2 Language definition

2.1 Syntax

Contextual type theory was introduced by Nanevski et al [14] and extended the logical framework LF [13] with first-class meta-variables. Our work is a natural continuation of this work generalizing contextual types to multiple levels. Following Watkins et al [21], the syntax is limited to expressing terms in β-normal forms, which are sufficient for encoding the types and expressions of some logic or programming language as well as judgements and derivations pertaining to those types and expressions. We leave the development of a non-canonical version to future work. While the grammar below only enforces that objects are β-normal, the typing rules will also ensure objects are moreover in η-long form.

Sorts	s	$::=$	$\mathsf{type} \mid \mathsf{kind}$
Atomic Types/Kinds	P, Q	$::=$	$s \mid \mathbf{a} \mid P\ (\hat{\Gamma}.N)$
Normal Types/Kinds	A, B, K	$::=$	$P \mid \Pi x^n{:}A[\Psi^n].\, B$
Atomic Terms	R	$::=$	$x^n[\sigma] \mid \mathbf{c} \mid R\ (\hat{\Gamma}.N)$
Normal Terms	M, N	$::=$	$R \mid \lambda x^n.\, M$
Substitutions	σ, τ	$::=$	$\cdot \mid \sigma, \hat{\Gamma}^n.M \mid \sigma, {}_{\triangle}\, x^n$
Contexts	Ψ, Φ, Γ	$::=$	$\cdot \mid \Psi, x^n{:}A[\Phi^n]$
Signature	Σ	$::=$	$\cdot \mid \Sigma, \mathbf{a}{:}K \mid \Sigma, \mathbf{c}{:}A$

Normal objects may contain variables x^n which are bound by λ-abstraction or declared in a context Ψ. Variables are associated with a level n. The level n of a context Ψ is an upper bound on $\{k+1 | x^k \in$

$\mathsf{dom}(\Psi)\}$. In other words, we write Ψ^n when we know that all variables in Ψ are at levels strictly smaller than n. Unlike the superscript on variables, the superscript on contexts is purely a mnemonic convenience as this annotation can normally be inferred from information elsewhere wherever relevant, just as lambda-abstractions are not annotated with the domain type because that information is usually already available somewhere else.

A variable x^n has type $A[\Psi^n]$, i.e. it has type A in the context Ψ of variables at levels lower than n. To put it differently, x^n may refer to the (local) variables in Ψ^n and may also contain (global) variables of a higher level. If $n = 0$, we recover our ordinary bound variables of type A. The context will be dropped, because there is no context of level 0. Yet, A may refer to meta-variables or more generally, to variables at a higher level. Similarly, we can recover meta-variables which are of level 1 and have type $A[\Psi^1]$. The context Ψ^1 contains only variables of level 0, i.e. ordinary bound variables. Hence, locally meta-variables depend on ordinary bound variables, but they may also contain (global) variables of higher level (for example, meta^2-variables). A variable x^n of type $A[\Psi^n]$ stands for an object $\hat{\Psi}^n.M$ where $\hat{\Psi}$ lists the bound variables that may occur in M and again, the level n indicates that it may contain locally bound variables only up to level n. This is important information when the need arises to rename the locally bound variables occurring in M, to avoid captures for instance.

As we navigate under binders, it may be necessary to substitute the bound variable for another one or for a term. Such substitutions get "stuck" at the level of meta-variables because there is no term to substitute in until the meta-variable is instantiated. In the core syntax we impose the invariant that all meta-variables standing for a term in a context Ψ be associated with a (simultaneous) substitution σ such that the domain of σ matches that of Ψ. As such, given Ψ, it is not necessary to make the domain of substitutions explicit, as that information would be redundant. Intuitively, the i-th element in σ corresponds to the i-th assumption in Ψ. A postponed substitution σ is applied as soon as we know what x^n stands for and applying σ with domain Ψ to a term M is written $[\sigma]_\Psi M$.

A substitution maps (canonical) terms for variables. But as we push this substitution under binders, the size of the context grows and so must that of the substitution. A term of the form $(\lambda x.\, M)[\sigma]$ can be rewritten to $(\lambda x.\, M[\sigma'])$, where σ' extends σ by mapping the variable x to itself. However, recall that x by itself is not a meaningful term in our grammar — all variables are systematically paired with a simultaneous substitution. Without knowing the type for x, we cannot infer the appropriate identity substitution which would allow us to replace x by $x[\mathsf{id}]$. Even if $x[\mathsf{id}]$ was made a valid syntactic object in our grammar, it is not guaranteed that $x[\mathsf{id}]$ is a canonical form at higher type (canonical forms are η-long). Moreover, the property that a given term M inhabits a certain type is an extrinsic property of M and types should play no role when propagating substitutions. Since the type of a term and its free variables might not be known in general, it is thus not always possible to put x in $\beta\eta$-long normal form. We therefore allow extending substitutions with renamings of variables, such as in $\sigma' = \sigma, \triangle\, x$, which maps x to itself.

Applications resemble the way substitutions are built. Using the typing rules as a guide, notice that being able to apply some atomic term R to some other term must mean the type of R denotes a function whose type must be of the shape $\Pi x^n{:}A[\Psi^n].\, B$. The function R then, expects an argument of type $A[\Psi^n]$, which can therefore only be of the shape $\hat{\Psi}^n.M$. It would make little sense to simply write M here, since M may contain "free" variables from $\hat{\Psi}^n$. Recall that all occurrences of the variables x^n in M are associated with a postponed substitution σ which will provide instantiations for the variables in Ψ^n. To further bring home the connection to substitutions, consider β-reduction. Eliminating a redex $(\lambda x^n.\, N)\ (\hat{\Psi}^n.M)$ means substituting $\hat{\Psi}^n.M$ for x^n in N, i.e. $[\hat{\Psi}^n.M/x^n]N$.

2.2 Context operations

Before moving to the typing rules, we explain the two necessary context manipulating operations: merging and chopping. When checking the domain of a dependent function $\Pi x^n{:}A[\Phi^n].\, B$ in context Ψ, it will be necessary to drop some of the assumptions in Ψ and extend Ψ with Φ^n. To chop off all variables below level n from the context Ψ, we write $\Psi|_n$. To merge two contexts Ψ and Φ we write $\Psi \mathbin{+\!\!+} \Phi$.

As mentioned earlier, contexts must be sorted according to the level of assumptions $x^n{:}A[\Psi^n]$. One should conceptualize this ordered context as a stack of subcontexts, one for each level of variables. Let $\Psi(k)$ be the subcontext of Ψ^n with only assumptions of level k. Then, $\Psi^n = \Psi(n-1), \Psi(n-2), \ldots, \Psi(1)$. We opt here for a flattened presentation of this stack of contexts in order to simplify merging and chopping of stacks.

However, keeping the context sorted comes at a cost: inserting new assumptions $x^k{:}A[\Phi^k]$ must respect the invariant that contexts are always sorted. The flipside is that guaranteeing that merging two contexts respects well-formedness is much easier and chopping contexts is more efficient. With merging defined, insertion of a new assumption into a context is a special case, so we dispense with defining a separate operation. Merging and chopping contexts are defined inductively as follows:

$$
\begin{array}{lcll}
\text{Merging contexts:} \Psi \mathbin{+\!\!+} \Phi = \Gamma & & & \\
\cdot \mathbin{+\!\!+} \Phi & = & \Phi & \\
\Psi \mathbin{+\!\!+} \cdot & = & \Psi & \\
\Psi, x^n{:}A[\Gamma^n] \mathbin{+\!\!+} \Phi, y^k{:}B[\Gamma'^k] & = & (\Psi, x^n{:}A[\Gamma^n] \mathbin{+\!\!+} \Phi), y^k{:}B[\Gamma'^k] & \text{if } k \leq n \\
\Psi, x^n{:}A[\Gamma^n] \mathbin{+\!\!+} \Phi, y^k{:}B[\Gamma'^k] & = & (\Psi \mathbin{+\!\!+} \Phi, y^k{:}B[\Gamma'^k]), x^n{:}A[\Gamma^n] & \text{otherwise} \\
 & & & \\
\text{Chopping contexts:} \quad \Psi|_n = \Phi & & & \\
\cdot|_n & = & \cdot & \\
(\Psi, x^k{:}A[\Phi^k])|_n & = & \Psi|_n & \text{if } k < n \\
(\Psi, x^k{:}A[\Phi^k])|_n & = & \Psi, x^k{:}A[\Phi^k] & \text{otherwise}
\end{array}
$$

Merging of two independent contexts is akin to the merge step of the mergesort algorithm and therefore inherits many of its properties. In particular, the merge of two sorted independent contexts is again a sorted context. It is also stable, in the sense that the relative positions of any two assumptions in Ψ^n or in Φ^k is preserved in $\Psi^n \mathbin{+\!\!+} \Phi^k$.

The chopping operation allows us to drop all variable assumptions below a given index from a context. If $k \leq n$, then $\Psi^k|_n = \cdot$. Similar to the chopping and merging operation on contexts, we will need chopping and merging on the level of simultanous substitutions, written $\sigma|_{\Gamma^n}$ and $\sigma \mathbin{+\!\!+} \rho$ respectively. These operations will be defined in Section 2.5 on page 11.

2.3 Typing rules

We present in this section a bi-directional type system, capable of checking normal terms (resp. normal types) against a type (resp. sorts) and synthesizing types (resp. sorts) for atomic entities. The rules are given in Figure 1. When reading the rules bottom-up, assumptions are accumulated into the context Ψ at the left of the turnstile as we descend into multi-level objects, but it is sometimes necessary to restrict it using the previously defined operations. All typing judgments have access to a well-typed signature Σ where we store constants together with their types and kinds. However, signatures declare global constants and never change in the course of a typing derivation. Therefore the parameterization of the typing rules by the signature Σ for constants is kept implicit.

$\Psi \vdash M \Leftarrow A$ Normal term M checks against type A
$\Psi \vdash R \Rightarrow A$ Neutral term R synthesizes type A
$\Psi \vdash \sigma \Leftarrow \Phi^n$ Substitution σ has domain Φ^n and range Ψ.

The bi-directional rules can be understood as determining two mutually defined algorithms for inferring the type of an object and checking an object against a type. We always assume that Ψ and the subject (M, R, or σ) are given, and that the contexts Ψ contains only canonical types and is well-formed. For checking $M \Leftarrow A$ we also assume A is given and canonical, and similarly for checking $\sigma \Leftarrow \Phi$ we assume Φ is given and is well-formed. For synthesis $R \Rightarrow A$ we assume R is given and we generate a canonical A. Similarly, at the level of types and contexts we have

$\Psi \vdash A \Leftarrow s$ Type/Kind A is well-formed
$\Psi \vdash A \Rightarrow K$ Type A synthesizes kind K
$\Psi \vdash \Phi\ \mathsf{ctx}$ Context Φ is well-formed in the context Ψ

with corresponding assumptions on the constituents.

As in Pure Type Systems, a type is well formed if its type is a sort. Whereas signatures might contain term-level and type-level constant declarations, we only allow declarations of sort type in contexts since abstractions and dependent function types may only abstract over terms, not types.

Checking that types are well-kinded is bi-directional. To check that $\Pi x^n{:}A[\Phi^n].\, B$ is a well-formed type in the context Ψ, we check first that the context Φ^n is well-formed in Ψ. We note that the assumptions in Φ^n should only have access to assumptions greater than or equal to n and checking that Φ^n is well-formed in the context Ψ will amount to checking that Φ^n only depends on assumptions $\Psi|_n$. Next, we verify that A is well-kinded. Because of the dependency of types on terms, Φ^n scopes over the type A. Consider for instance

$$(\hat{\Gamma}.\mathsf{cons}\ n\ x\ xs) : (\mathsf{vec}\ n)[\Gamma]$$

where $\Gamma = n{:}\mathsf{nat}, x{:}\mathsf{bool}, xs{:}\mathsf{vec}\ n$. The type of this instantiation for a meta-variable depends on the variable n bound in Γ. A may refer to the variables in Φ^n, but also to variables m where $m \geq n$ from Ψ. Hence, we drop from Ψ all assumptions below n and merge the resulting context with Φ^n. Finally, we check that B is well-kinded in the context Ψ extended with the assumption $x^n{:}A[\Phi^n]$. We rely on the previously defined merging operation on contexts, to insert the assumption $x^n{:}A[\Phi^n]$ at the appropriate position in Ψ.

To check that atomic types are well-kinded, we synthesize their kind. For type constants, we simply look up their type in the signature Σ. The interesting case is the application rule. To synthesize the kind for $P\ (\hat{\Phi}^n.N)$, we first synthesize the kind for P as $\Pi x^n{:}A[\Phi^n].\, K$. Subsequently, we check that N has type A. We again must be careful regarding the context. First, some renaming may be necessary to bring the locally bound variables described in $\hat{\Phi}^n$ in sync with the context. Moreover, we observe that all variables below n which occur in N and A refer to binding sites in Φ^n. All variables equal or greater than n which occur in N and A refer to binding sites in $\Psi|_n$, i.e. the context Ψ where we drop all assumptions below n. Finally, we must be careful to substitute $\hat{\Phi}^n.N$ for x^n in K in the resulting kind we return. Because our grammar only recognizes β-normal objects as syntactically well-formed, we must rely on hereditary substitution to hereditarily eliminate any redices as we instantiate variables in the target of the substitution. We annotate here the substitution with the type of $A[\Phi^n]$. This is only strictly necessary to ensure that hereditary substitutions terminate. We postpone the definition and discussion on hereditary substitutions for now and will revisit it in Section 2.5.

In the lambda-abstraction rule, we check that $\lambda x^n.M$ has type $\Pi x^n{:}A[\Phi^n].\, B$ by inserting the new assumption $x^n{:}A[\Phi^n]$ at the appropriate position in Ψ and continuing to check that M has type B. Note that, without loss of generality, we implicitly assume here and everywhere else that $x^n \notin \Psi$. This can

Atomic Types/Kinds $\boxed{\Psi \vdash P \Rightarrow K}$

$$\frac{\Sigma(\mathbf{a}) = K}{\Psi \vdash \mathbf{a} \Rightarrow K} \qquad \frac{\Psi \vdash P \Rightarrow \Pi x^n{:}A[\Phi^n].\, K \quad \Psi|_n +\!\!+ \Phi^n \vdash N \Leftarrow A}{\Psi \vdash P\ (\hat{\Phi}^n.N) \Rightarrow [\hat{\Phi}^n.N/x^n]_{A[\Phi^n]}K}$$

Normal Types/Kinds $\boxed{\Psi \vdash A \Leftarrow s}$

$$\frac{}{\Psi \vdash \mathsf{type} \Leftarrow \mathsf{kind}} \qquad \frac{\Psi \vdash P \Rightarrow \mathsf{type}}{\Psi \vdash P \Leftarrow \mathsf{type}}$$

$$\frac{\Psi|_n +\!\!+ \Phi^n \vdash A \Leftarrow \mathsf{type} \quad \Psi \vdash \Phi^n\ \mathsf{ctx} \quad \Psi +\!\!+ x^n{:}A[\Phi^n] \vdash B \Leftarrow s}{\Psi \vdash \Pi x^n{:}A[\Phi^n].\, B \Leftarrow s}$$

Atomic Terms $\boxed{\Psi \vdash M \Rightarrow A}$

$$\frac{\Psi(x^n) = A[\Phi^n] \quad \Psi \vdash \sigma \Leftarrow \Phi^n}{\Psi \vdash x^n[\sigma] \Rightarrow [\sigma]_{\Phi^n}A} \qquad \frac{\Sigma(\mathbf{c}) = A}{\Psi \vdash \mathbf{c} \Rightarrow A} \qquad \frac{\Psi \vdash R \Rightarrow \Pi x^n{:}A[\Phi^n].\, B \quad \Psi|_n +\!\!+ \Phi^n \vdash N \Leftarrow A}{\Psi \vdash R\ (\hat{\Phi}^n.N) \Rightarrow [\hat{\Phi}^n.N/x^n]_{A[\Phi^n]}B}$$

Normal Terms $\boxed{\Psi \vdash M \Leftarrow A}$

$$\frac{\Psi \vdash R \Rightarrow P \quad P = Q}{\Psi \vdash R \Leftarrow Q} \qquad \frac{\Psi +\!\!+ x^n{:}A[\Phi^n] \vdash M \Leftarrow B}{\Psi \vdash \lambda x^n.\, M \Leftarrow \Pi x^n{:}A[\Phi^n].\, B}$$

Substitutions $\boxed{\Psi \vdash \sigma \Leftarrow \Phi^n}$

$$\frac{}{\Psi \vdash \cdot \Leftarrow \cdot} \qquad \frac{\Psi \vdash \sigma \Leftarrow \Phi^n \quad \Psi|_k +\!\!+ [\sigma]_{\Phi^n}(\Gamma^k) \vdash M \Leftarrow [\sigma']_{(\Phi^n|_k\, +\!\!+\, \Gamma^k)}A}{\Psi \vdash \sigma, \hat{\Gamma}^k.M \Leftarrow \Phi^n, x^k{:}A[\Gamma^k]}\ \text{where } \sigma' = \sigma|_k +\!\!+ \mathsf{id}(\hat{\Gamma}^k)$$

$$\frac{\Psi \vdash \sigma \Leftarrow \Phi^n \quad \Psi(y^k) = [\sigma]_{\Phi^n}(A[\Gamma^k])}{\Psi \vdash \sigma, {}_{\triangle}\, y^k \Leftarrow \Phi^n, x^k{:}A[\Gamma^k]}$$

Context well-formedness $\boxed{\Psi \vdash \Phi^n\ \mathsf{ctx}}$

$$\frac{}{\Psi \vdash \cdot\ \mathsf{ctx}} \qquad \frac{\Psi \vdash \Phi^n\ \mathsf{ctx} \quad \Psi|_n +\!\!+ \Phi^n|_k +\!\!+ \Gamma^k \vdash A \Leftarrow \mathsf{type} \quad \Psi|_n +\!\!+ \Phi^n|_k \vdash \Gamma^k\ \mathsf{ctx}}{\Psi \vdash \Phi^n, x^k{:}A[\Gamma^k]\ \mathsf{ctx}}\ k < n$$

Figure 1: Typing rules for LF with contextual variables and context variables

always be achieved by α-renaming. When we reach a normal object of atomic type, we synthesize a type Q and compare Q to the expected type P. Comparing two types reduces to checking structural equality between Q and P modulo renaming, since all types and terms are always in canonical form. The only minor complication arises when checking that two substitutions are equivalent. Because we may simply write x^n for a variable of type $A[\hat{\Psi}^n]$ or its expanded form, comparing two substitutions must take into account η-contraction.

To synthesize the type of a constant, we simply look up its type in the signature Σ. Term-level application $R\ (\hat{\Phi}^n.N)$ follows the same ideas as type-level applications. The most interesting rule is the one for variables. To synthesize the type of a variable x^n we retrieve its type $A[\Phi^n]$ from Ψ. Next, we check that the substitution σ which is associated with x^n maps variables from Φ^n to Ψ. Finally, we return the type of $x^n[\sigma]$ which is $[\sigma]_{\Phi^n} A$.

A substitution $\sigma, \hat{\Gamma}^k.M$ checks against domain $\Phi^n, x^k{:}A[\Gamma^k]$, if σ checks against Φ^n and in addition M is well-typed. As in the rules for applications, we must be a little careful about where variables in M are bound. M contains locally bound variables from $\hat{\Gamma}^k$ as well as global variables from $\Psi|_k$. We again restrict Ψ to only contain variables above k, since all variables below k are bound in Γ. Next, we inspect the type dependencies. We note that Γ^k is a well-formed context in Φ^n, although the typing rules will ensure Γ^k only accesses declarations from $\Phi^n|_k$. Similarly, when applying σ to Γ^k, we will ensure that σ will be appropriately restricted (see the definition in the appendix) to only substitute for variables of level k and higher. Therfore, $[\sigma]_{\Phi^n}(\Gamma^k)$ yields a well-formed context in $\Psi^n|_k$. On the other hand, A is well-typed in the context $\Phi^n|_k +\!\!+ \Gamma^k$, however σ has domain Φ^n. Simply applying σ to A would be incorrect; instead, we restrict σ to contain only the mappings for the variables in $\Phi^n|_k$ and map all the variables from Γ^k to themselves. This is written as $[\sigma|_k +\!\!+ \mathsf{id}(\hat{\Gamma}^k)]_{(\Phi^n|_k +\!\!+ \Gamma^k)}$.

Checking the extension $\sigma, \triangle\, x^k$ of a substitution by a variable involves looking up the declared type of x^k in Ψ and compare it to the expected type. Because the expected type $A[\Gamma^k]$ was well-typed in Φ^n, we must verify that $\Psi(x^k) = [\sigma]_{\Phi^n}(A[\Gamma^k])$. Note that $[\sigma]_{\Phi_n}(A[\Gamma^k]) = ([\sigma|_{\Gamma^k}]_{(\Phi_n|_k +\!\!+ \Gamma^k)}A)[[\sigma]_{\Phi_n}\Gamma^k]$.

Finally, we consider the rules that characterize well-formed contexts. In the typing rules discussed above, we are often given contexts Φ that are not closed but rather whose assumptions might depend on the ambient context Ψ. Since Ψ is already assumed well-formed, we keep it to the left of the turnstile and write $\Psi \vdash \Phi^n\ \mathsf{ctx}$ to mean Φ is a well-formed context at level n in context Ψ. An alternative would have been to have judgements of the form $\vdash \Gamma^n\ \mathsf{ctx}$ only and state that Φ^n in Ψ is well-formed as $\vdash \Psi|_n +\!\!+ \Phi^n$. A context $\Phi^n, x^k{:}A[\Gamma^k]$ is well-formed if Φ^n is well-formed and $A[\Phi^k]$ is well-typed. Again we must be careful about the dependency structure. The context Γ^k can refer to variables from Φ^n, but only at levels $k \leq n$. Moreover, any variable x^m where $n \leq m$ is declared in Φ.

2.4 Properties

We begin by proving some properties about contexts and context merging and chopping. We first show that we can always increase the upper bound of a context.

Lemma 1 (Cumulativity)**.** *If* $\Psi \vdash \Phi^n\ \mathsf{ctx}$ *and* $n < k$ *then* $\Psi \vdash \Phi^k\ \mathsf{ctx}$.

Next, we show that merging produces well-formed contexts, if both contexts are independent.

Lemma 2 (Closure under independent context merging)**.**
If $\cdot \vdash \Psi^n\ \mathsf{ctx}$ *and* $\cdot \vdash \Phi^k\ \mathsf{ctx}$ *then* $\cdot \vdash (\Psi^n +\!\!+ \Phi^k)^{\max(n,k)}\ \mathsf{ctx}$.

More importantly, if we extend a context Ψ^n with a context Φ^k where $\Psi^n \vdash \Phi^k\ \mathsf{ctx}$, the resulting context $\Psi^n +\!\!+ \Phi^k$ is well-formed. This lemma is crucial to ensure that we work with well-formed contexts during typing.

Lemma 3 (Well-formed context extension)**.**
If $\vdash \Psi^n$ ctx *and* $\Psi^n \vdash \Phi^k$ ctx *then* $\cdot \vdash (\Psi^n +\!\!+ \Phi^k)^{\max(n,k)}$ ctx.

Lemma 4 (Closure under chopping)**.** *If* $\cdot \vdash \Psi^k$ ctx *then* $\cdot \vdash (\Psi|_n)^k$ ctx.

Lemma 5 (Weakening, Identity)**.**

1. *If* $\Psi \vdash J$ *then* $\Psi +\!\!+ x^n{:}A[\Phi^n] \vdash J$.
2. *Let* $\Psi(n) = \overrightarrow{x^n{:}A[\Phi^n]}$ *denote the subcontext* $\Psi(n) \subseteq \Psi$ *of assumptions at level* n*. We have that* $\Psi \vdash \Psi(n)$.

Lemma 6 (Well-formedness of contexts at level k)**.** $\Psi \vdash \Phi^k$ ctx *iff* $\Psi|_k \vdash \Phi^k$ ctx.

2.5 Hereditary Substitution

Normal terms are not closed under vanilla substitution, a rather problematic matter of fact given that our syntax can only express normal forms. For example, when replacing naively x by $\lambda y.c\ y$ in the object $x\ z$, we would obtain $(\lambda y.c\ y)\ z$. It is essential therefore to iron out as we go any redices we might create as a result of substituting terms for variables. We hence follow [21] in defining a *hereditary* substitution, which does just that. That hereditary substitutions always terminate on well-typed normal terms is crucial to ensuring that our typing rules are decidable. In the above example, hereditary substitutions continue to substitute z for y in $c\ y$ to obtain $c\ z$ as a final result. This idea scales to our setting, but we must be careful to observe the scope of variables.

Hereditary substitution are defined structurally considering the term to which the substitution operation is applied and the type of the object which is being substituted. The type is only needed to construct evidence of termination. We define the hereditary substitution operations for types, normal object, neutral objects, substitutions, and contexts.

In the formal development it is simpler if we can stick to the structure of the example above and use only non-dependent types in hereditary substitutions. This suffices because we only need to know whether we have encountered a function which can be reduced further or whether we have reached an object of base type and reduction will terminate. We therefore first define type approximations α and an erasure operation $()^-$ that removes dependencies. Before applying any hereditary substitution $[\hat{\Phi}^n.N/x^n]_{A[\Phi^n]}(M)$ we first erase dependencies to obtain $\alpha[\phi] = (A[\Phi])^-$ and then carry out the hereditary substitution proper as $[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}B$. A similar convention applies to the other forms of hereditary substitutions.

$$\begin{array}{rrcl} \text{Type approximation} & \alpha, \beta & ::= & a \mid \alpha[\gamma] \Rightarrow \beta \\ \text{Context approximation} & \gamma, \psi & ::= & \cdot \mid \gamma, x^n{:}\alpha[\phi] \mid \gamma, x^n{:}\_ \end{array}$$

The last form of context approximation, $x^n{:}\_$ is needed when the approximate type of x^n is not available.[1] It does not arise directly from erasure.

Types and contexts are related to type and context approximations via an erasure operation $()^-$ which we overload to work on types and contexts.

$$\begin{array}{lcl} (a)^- & = & a \\ (P\ (\hat{\Psi}.N))^- & = & P^- \\ (\Pi x^n{::}A[\Psi^n].\, B)^- & = & (A)^-[(\Psi^n)^-] \Rightarrow (B)^- \\ (\cdot)^- & = & \cdot \\ (\Psi^n, x^k{:}A[\Phi^k])^- & = & (\Psi^n)^-, x^k{:}(A)^-[(\Phi^k)^-] \end{array}$$

[1] See the definition of $[\sigma]_{\psi^n}(\lambda y^n.\, M)$ in the electronic appendix.

Herediatary substitution is given by the following equations. We overload the substitution operation to work on normal terms, neutral terms, substitutions, and contexts.

$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(N) = N'$			Hereditary substitution into N
$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R) = R'$	or	$M' : \alpha'$	Hereditary substitution into R
$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\sigma) = \sigma'$			Hereditary substitution composition
$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\Psi) = \Psi'$			Hereditary substitution into Ψ

Applying a substitution to a neutral term, may yield either a neutral term or a normal term together with a type approximation. In the latter case, we can simply drop the type approximation, since it is only necessary for guaranteeing that any reductions triggered when applying the substitution to a neutral term will terminate.

Substitution into normal and neutral terms We present hereditary substitution for normal terms and neutral terms, substitutions and context. The definitions for types can be found in the appendix.

Substitution into normal terms

$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\lambda y^k.M)$	$=$	$\lambda y^k.M'$	where $[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(M) = M'$ if $k < n$
$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\lambda y^k.M)$	$=$	$\lambda y^k.M'$	where $[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(M) = M'$ if $y^k \notin \mathsf{FV}(\hat{\Phi}^n.N)$ and $k \geq n$
$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R)$	$=$	M	if $[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R) = M' : \alpha'$
$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R)$	$=$	R'	if $[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R) = R'$
$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(N)$		fails	otherwise

Substitution into neutral terms

$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\mathbf{c})$	$=$	$\mathbf{c}$	
$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(x^n[\sigma])$	$=$	$N' : \alpha$	where $[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\sigma) = \sigma'$ and $[\sigma']_\phi(N) = N'$
$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(y^k[\sigma])$	$=$	$y^k[\sigma']$	where $[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\sigma) = \sigma'$ if $y^k \neq x^n$
$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R\ (\hat{\Psi}^k.M))$	$=$	$R'\ \ (\hat{\Psi}^k.M')$	where $[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R) = R'$ and $[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(M) = M'$ if $k \leq n$
$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R\ (\hat{\Psi}^k.M))$	$=$	$R'\ \ (\hat{\Psi}^k.M)$	where $[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R) = R'$ if $k > n$
$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R\ (\hat{\Psi}^k.M))$	$=$	$N'' : \beta$	where $[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R) = \lambda y^k.\, N' : \gamma[\psi] \to \beta$ and $M' = [\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(M)$ and $N'' = [\hat{\Psi}^k.M'/y^k]_{\gamma[\psi]}(N')$ if $\gamma[\psi] \to \beta \leq \alpha[\phi]$ and $k \leq n$
$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R\ (\hat{\Psi}^k.M))$	$=$	$N'' : \beta$	where $[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R) = \lambda y^k.\, N' : \gamma[\psi] \to \beta$ and $N'' = [\hat{\Psi}^k.M/y^k]_{\gamma[\psi]}(N')$ if $\gamma[\psi] \to \beta \leq \alpha[\phi]$ and $k > n$
$[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R)$		fails	otherwise

Figure 2: Hereditary substitution on normal terms and neutral terms

We define $[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(M)$ and $[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(R)$ by nested induction, first on the structure of the type approximation $\alpha[\phi]$ and second on the structure of the objects N and R. In other words, we either go to a smaller type approximation (in which case the objects can become larger), or the type approximation remains the same and the objects become smaller. We write $\alpha \leq \beta$ and $\alpha < \beta$ if α occurs in β (as a proper subexpression in the latter case). Such occurrences can be inside a context approximation ψ in the function approximation $\beta_1[\psi] \Rightarrow \beta_2$, so we also write $\alpha < \psi$ if $\alpha \leq \beta$ for some $y^k{:}\beta[\gamma]$ in ψ, and we write $\alpha < \beta[\psi]$ if $\alpha \leq \beta$ or $\alpha < \psi$.

When defining substitutions, we must be careful to take into account where multi-level variables are bound. For example, when applying $[\hat{\Phi}^n.N/x^n]$ to a lambda-abstraction $\lambda x^k.M$, we must check for possible capture, if $k \geq n$. Recall that $\hat{\Phi}^n.N$ only binds variables up to level n locally and N can still refer to variables greater or equal to n which have from N's perspective a global status. Therefore, if $k \geq n$, we must ensure that y^k does not occur in the free variables of $\hat{\Phi}^n.N$, written as $\mathsf{FV}(\hat{\Phi}^n.N)$. If $k < n$, then y^k can in fact not appear in $\hat{\Phi}^n.N$ because all variables of level k are bound in $\hat{\Phi}^n$. Recall that all variables $x^n[\sigma]$ exist as closures, and hence all variables in $\hat{\Phi}^n$ will be substituted for using σ.

When considering the substitution operation on neutral terms, two cases are interesting, applying the substitution to a variable and to an application. When we apply $\hat{\Phi}^n.N/x^n$ to a variable $y^k[\sigma]$, we apply it to σ obtaining σ' as a result. If $y^k \neq x^n$, we simply return $y^k[\sigma']$. If $y^k = x^n$, i.e. $n = k$ and $y = x$, then we must continue to apply σ' to N obtaining N'. We then return $N' : \alpha$, because $y^k[\sigma]$ may have occurred in a functional position, and we must trigger a β-reduction step, if it is applied.

Propagating the hereditary substitution $[\hat{\Phi}^n.N/x^n]$ through an application $R\ (\hat{\Psi}^k.M)$ is split into multiple cases considering the level and the possible elimination of created redices. If $k \leq n$, we need to apply the substitution not only to R but also to $(\hat{\Phi}^k.M)$, because M may refer to x^n; otherwise, we only need to apply the substitution to R, because all occurrences of x^n in M are bound locally by $\hat{\Phi}^k$.

If applying the substitution to R produces a normal term $\lambda y^k.N'$, then we must continue to substitute and replace y^k with $(\hat{\Psi}^k.M)$ in N'. The approximate type annotations on the substitution guarantees that the approximate type of the lambda-abstraction is smaller than the approximate type of x^n, and hence this hereditary substitution will terminate.

Single substitution into simultaneous substitutions Applying $\hat{\Phi}^n.N/x^n$ to a substitution σ is done recursively and is straightforward, when we keep in mind the fact that all variables below k are bound within M when we encounter $\hat{\Psi}^k.M$. If $k \geq n$, then applying the substitution $\hat{\Phi}^n.N/x^n$ to $\hat{\Psi}^k.M$ will leave it unchanged. Only if $k \leq n$, we push the substitution $\hat{\Phi}^n.N/x^n$ through M. When we encounter $\sigma', \triangle\, y^k$ where $y^k \neq x^n$, we simply apply the substitution to σ'. If we encounter $\sigma', \triangle\, x^n$, then we must replace x^n by $\hat{\Phi}^n.N$.

$$
\begin{array}{lcll}
[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\cdot) & = & \cdot & \\
[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\sigma, \hat{\Psi}^k.M) & = & \sigma', \hat{\Psi}^k.M' & \text{where } [\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(M) = M' \\
 & & & \text{and } [\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\sigma) = \sigma' \text{ if } k \leq n \\
[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\sigma, \hat{\Psi}^k.M) & = & \sigma', \hat{\Psi}^k.M & \text{where } [\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\sigma) = \sigma' \text{ if } k > n \\
[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\sigma, \triangle\, x^n) & = & \sigma', \hat{\Phi}^n.N & \text{where } [\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\sigma) = \sigma' \\
[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\sigma, \triangle\, y^k) & = & \sigma', \triangle\, y^k & \text{where } [\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\sigma) = \sigma' \text{ if } y^k \neq x^n
\end{array}
$$

Substitution into contexts Applying $\hat{\Phi}^n.N/x^n$ to a context proceeds recursively on the structure of the context until we encounter a declaration x^k where $k > n$. Because our contexts are ordered, we know that the remaining context cannot contain an occurrence of x^n.

$$
\begin{array}{lcll}
[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\cdot) & = & \cdot & \\
[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\Psi, y^k{:}A[\Gamma^k]) & = & \Psi', y^k{:}A'[\Gamma'^k] & \text{where } [\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\Psi) = \Psi' \\
 & & & \text{and } [\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(A[\Gamma^k]) = A'[\Gamma'^k] \text{ if } k \leq n \\
[\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(\Psi, y^k{:}A[\Gamma^k]) & = & \Psi, y^k{:}A[\Gamma^k] & \text{if } k > n
\end{array}
$$

Simultaneous substitutions Similar to the single substitution operation, the simultanous substitution operation is indexed with the domain of σ described by the approximate context ϕ. To define simultaneous substitutions it is worth recalling how they arise: A simultaneous substitution σ is associated with a variable x^n which has type $A[\Phi^n]$. As a consequence, σ provides instantiations for variables declared in Φ^n, i.e. variables up to level n. Any variable at level n or above is bound in the ambient context and σ will not provide any substitutions for such variables. It thus makes sense to write any upper bound on a simultaneous substitution explicitly as in σ^n, just as for contexts, but we generally choose to omit it. Defining the simultaneous substitution we must be careful to ensure that the usual substitution properties for simultaneous substitution holds — if $\Phi \vdash \sigma \Leftarrow \Psi^n$ and $\Gamma|_n +\!\!+ \Psi^n \vdash J$ then $\Gamma|_n +\!\!+ \Phi \vdash [\sigma]_\psi J$. $\Gamma|_n$ is the ambient context which remains untouched by the simultanous substitution σ.

The typing rules act again as a guide in our definitions. For lambda-abstractions for instance, pushing a simultanous substition σ^n through $\lambda y^k.M$ we need to distinguish cases based on the level: if $k < n$, then we must extend the substitution σ with the identity mapping for y^k. From the typing rule for lambda-abstractions, we see that contexts are ordered and we insert the new declaration y^k at its correct position using context merging. Similarly, we need to define substitution merging operation which will extend σ and insert y^k at its correct position. If $k \geq n$, then σ does not need to be extended as we push σ^n through the λ-abstraction because the substituion σ^n has no effect on variables above level n.

For applications for instance, applying a simultaneous substitution σ where $\Gamma \vdash \sigma \Leftarrow \Psi^n$ ctx should take R $(\hat{\Phi}^m.N)$ from the context Ψ to the context Γ. This is justified in a straightforward manner by appealing to the induction hypothesis on the premises of the typing rule. However, we cannot directly appeal to the induction hypothesis in the second premise since the assumptions do not match the domain of σ. If we want the substitution property to hold, we must define substitution chopping operation similar the context chopping operation together with an identity substitution for mapping the variables in Φ^n to themselves. We therefore define chopping as $(\sigma/\psi)|_n$ inductively on the structure of σ and its domain ψ. Both for chopping off parts of a substitution and merging substitutions, we will resurrect the domain of the substitution to have access to the level of each variable for which the substitution provides a mapping. For convenience, we omit writing out the resurrected contexts during merging and chopping when they are understood. We also write F for $\triangle\, x^n$ and $\hat{\Psi}^n.N$.

Merging substitutions: $\sigma/\psi +\!\!+ \tau/\phi = \rho$

$$
\begin{array}{lcll}
\cdot/\cdot +\!\!+ \tau/\phi & = & \tau & \\
\sigma/\psi +\!\!+ \cdot/\cdot & = & \sigma & \\
(\sigma/\psi,\ F/x^n) +\!\!+ (\tau/\phi,\ F'/y^k) & = & ((\sigma/\psi,\ F/x^n) +\!\!+ \tau/\phi), F' & \text{if } k \leq n \\
(\sigma/\psi,\ F/x^n) +\!\!+ (\tau/\phi,\ F'/y^k) & = & (\sigma/\psi +\!\!+ (\tau/\phi,\ F'/y^k)), F & \text{otherwise}
\end{array}
$$

Chopping substitutions: $(\sigma/\psi)|_n = \tau$

$$
\begin{array}{lcll}
\cdot|_n & = & \cdot & \\
(\sigma/\psi,\ \ F/y^k)|_n & = & (\sigma/\psi)|_n & \text{if } k < n \\
(\sigma/\psi,\ \ F/y^k)|_n & = & \sigma, F & \text{if } k \geq n
\end{array}
$$

The identity substitution, written as $\mathsf{id}(\hat{\Phi})$ is the simple unrolling of $\hat{\Phi}$ into a substitution: $\mathsf{id}(\cdot) = \cdot$ and $\mathsf{id}(\hat{\Phi}, x^l) = \mathsf{id}(\hat{\Phi}), {}_{\triangle}\, x^l$. Due to lack of space we omit the definition of simultanous substitution here (see appendix for the full definition).

Properties of substitutions If the original term is not well-typed, a hereditary substitution, though terminating, cannot always return a meaningful term. In that case, we simply fail to return a result. Later we show that on well-typed terms, hereditary substitution always returns well-typed terms.

Applying the substitution to an object will terminate because either we apply the substitution to a sub-expression or the objects we substitute are smaller. The following substitution property holds for types, terms, substitutions and contexts.

Lemma 7 (Termination)**.**

1. *If* $[\hat{\Psi}^n.N/x^n]_{\alpha[\psi]}(R) = M' : \beta$ *then* $\beta \leq \alpha[\psi]$.
2. $[\hat{\Psi}^n.N/x^n]_{\alpha[\psi]}(\_)$ *terminates, either by returning a result or failing after a finite number of steps.*

Lemma 8 (Identity extension)**.**

1. *If* $\Psi \vdash \sigma \Leftarrow \Phi$ *then* $\Psi +\!\!+ x^k{:}[\sigma]_\Phi(A[\Gamma^k]) \vdash \rho \Leftarrow \Phi +\!\!+ x^k{:}A[\Gamma^k]$ *where* $\rho = \sigma/\phi +\!\!+ {}_{\triangle}\, x^k/x^k$.
2. *If* $\Psi \vdash \sigma \Leftarrow \Phi$ *then* $\Psi +\!\!+ [\sigma]_\phi\Gamma \vdash \rho \Leftarrow \Phi +\!\!+ \Gamma$ *where* $\rho = \sigma/\phi +\!\!+ \mathsf{id}(\gamma)$.

Lemma 9 (Substitution property)**.**

1. *If* $\Delta|_n +\!\!+ \Psi^n \vdash \sigma \Leftarrow \Phi^n$ *and* $\Delta|_n +\!\!+ \Phi^n \vdash J$ *then* $\Delta|_n +\!\!+ \Psi^n \vdash [\sigma]_\phi(J)$.
2. *If* $\Psi|_n +\!\!+ \Phi^n \vdash N \Leftarrow A$ *and* $\Psi, x^n{:}A[\Phi^n] \vdash J$ *then* $\Psi \vdash [\hat{\Phi}^n.N/x^n]_{\alpha[\phi]}(J)$.

Note that in the case of type synthesis of terms, the conclusion of the statements depend on the result of the substitution, e.g.:

(a) $\Delta|_n +\!\!+ \Psi^n \vdash R' \Rightarrow [\sigma]_\phi C$ *if* $[\sigma]_\phi R = R'$.

(b) $\Delta|_n +\!\!+ \Psi^n \vdash M \Leftarrow [\sigma]_\phi C$ *if* $[\sigma]_\phi R = M : \alpha$ *where* $\alpha = ([\sigma]_\phi C)^-$.

The typing judgments are syntax-directed and therefore clearly decidable. Hereditary substitution always terminates, giving us a decision procedure for dependent typing.

Theorem 10 (Decidability of Type Checking)**.**
All judgments in the dependent contextual modal type theory are decidable.

3 Related Work

Multi-level logics of contexts Contextual reasoning has been extensively studied for various applications in AI. For example, Giunchiglia *et al* have explored contextual reasoning [9] and have investigated a multi-language hierarchical logic where we have an infinite level of multiple distinct languages.

In [10] they introduce a class of multi-language systems which use a hierarchy of first-order languages, each language containing names of the language below. Any two adjacent languages in the hierarchy are linked only by two bridge rules. The hierarchy is understood as an alternative to extending modal logics with new modalities. Their goal is to provide a foundation to the implementation of "intelligent" reasoning systems. As the authors observe, we may use a different system to reason about a object logic (which may rely on induction) vs reasoning within a given object logic. Indexes encode information of the "locality of the reasoning", where the reasoning take place. This is similar in spirit to our use of level annotations on variables.

Multi-level meta-variables We motivated the multi-level system in the introduction with the need to model the dependency of holes on bound variables as well as meta-variables. This naturally leads to a multi-level system. This idea has played an important role in Sato et al [19] where the authors develop a multi-level calculus for meta-variables. As in our work, variables carry an index to indicate whether they are a bound variable, meta-variable, or a meta^2-variable, etc. The main difference compared to our work is that the authors define a "textual" substitution which allows capture. This is unlike our capture-avoiding substitution operation. There are two main obstacles with textual substitutions. First, we will lose confluence. The second problem is that some reductions may get stuck. To address these problems the authors suggest to define reductions in such a way that it takes into account the different levels and keep track of arities of functions. This leads to a carefully engineered system which is confluent and strongly normalizing, although not very intuitive. We believe our framework is simpler.

Gabbay and Lengrand [6; 7] propose a multi-level calculus for meta-variables called the Lambda Context Calculus where variables are modeled via nominals. They also define two different kinds of substitutions: one, a capture-allowing substitution, i.e. a meta-variable of level n is allowed to capture names below n and two, a captrue-avoiding substitution for variables greater than n. This inherently leads to difficulaties regarding confluence. Our work has one uniform capture-avoiding substitution operation leading to a more elegant calculus.

Finally, we mention the work by Geuvers and Jogjov (see for example [8]). Open terms are represented via a kind of meta-level Skolem function. However, in general reduction and instantiation of meta-variables (or holes) do not commute. This problem also arises in Bognar and de Vrijer [3]. Our work resolves many of these aforementioned problems, since reduction and instantiation naturally commute and require no special treatment.

Functional multi-level staged computation The division of programs into two stages has been studied intensively in partial evaluation and staged functional computation. Davies and Pfenning [4] proposed the use of the modal necessity operator to provide a type-theoretic foundation for staged computation and more specifically, run-time code generation. Abstractly, values of type $\Box A$ stand for an (unevaluated) source expression. To avoid generating closures, contextual types may be used to generate "open" (unevaluated) source expressions [14]. An open source expression would then have type $A[\Psi]$ to describe code of type A in a context Ψ.

However, this two-level framework does not allow us to specify multi-level transition points (e.g. dynamic until stage n). For example, Glück and Jørgensen [12; 11] propose a multi-level specialization to allow an accurate and fast multi-level binding-time analysis. This means that a given program can be optimized with respect to some inputs at an earlier stage, and others at later stages. Glück and Jørgensen generalized two-level program generators into multi-level ones, called multi-level generating extensions. By this generalization, a generated code fragment can be used for different levels. Subsequently, Yuse and Igarashi [22] proposed a logical foundation for multi-level generating extensions based on linear time temporal logics. We believe that our framework of multi-level contextual types may be used as an alternative to generate open code which can be used at different levels and manage its dependency on previous levels cleanly.

4 Conclusion

We generalized and extended the original contextual type theory of Nanevski et al [14], where we distinguish between meta-variables and bound variables, to multiple levels. This streamlines the original

presentation with fewer typing rules, syntax and operations but with many of the same properties. Substitutions are defined, checked applied and manipulated in exactly the same way as meta-substitutions, for instance. We believe our framework provides already a suitable foundation for formalizing contexts in theorem proving and functional programming. Unlike other attempts to provide a multi-level calculus, we believe our work which is based on contextual modal types avoids and simplifies many of the issues which arise such as capture-avoiding substitution and the related issues of confluence.

While we have used a named calculus for expository purposes, the system also generalizes nicely a calculus based on de Bruijn indices and suspensions more typical of an implementation, such as that of Abel and Pientka [1]. Variables are then pairs of indexes into a substitution inside a stack of substitutions.

We have shown that it is possible to *express* meta^k-terms in this generalized contextual type theory but have left largely untouched the question of how to attach a computational behaviour to such objects and *compute* with them. A first step towards representing computations is to move to a non-canonical calculus that permits arbitrary (typed) terms, to which we could attach arbitrary rewrite rules as in deduction modulo [5]. We could then add meaningful recursors to some of the layers to write proofs by induction, or add more computational effects for a layer acting as a tactic layer for a programming and reasoning system such as Beluga. We would obtain a uniform framework for all of representations of syntax, proofs over these representations and tactics over these proofs. And indeed, such a system would be useful for implementing Beluga within Beluga itself.

References

[1] Andreas Abel & Brigitte Pientka (2010): *Explicit substitutions for contextual type theory*. In Karl Crary & Marino Miculan, editors: *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'10)*, *Electronic Proceedings in Theoretical Computer Science (EPTCS)* 34.

[2] Andreas Abel & Brigitte Pientka (2011): *Higher-Order Dynamic Pattern Unification for Dependent Types and Records*. In Luke Ong, editor: *10th International Conference on Typed Lambda Calculi and Applications (TLCA'11)*, Lecture Notes in Computer Science, Springer, p. to appear. To appear.

[3] Mirna Bognar & Roel de Vrijer (2001): *A calculus of lambda calculus context*. *Journal of Automated Reasoning* 27(1), pp. 29–59.

[4] Rowan Davies & Frank Pfenning (2001): *A modal analysis of staged computation*. *Journal of the ACM* 48(3), pp. 555–604.

[5] Gilles Dowek, Thérèse Hardin & Claude Kirchner (2003): *Theorem Proving Modulo*. *Journal of Automated Reasoning* 31(1), pp. 33–72.

[6] Murdoch Gabbay (2007): *Hierarchical Nominal Terms and Their Theory of Rewriting*. In B. Pientka & A. Momigliano, editors: *1st International Workshop on Logical Frameworks and Meta-Languages (LFMTP'06)*, 174(5), Electronic Notes Theorical Computer Science, pp. 37–52.

[7] Murdoch J. Gabbay & Stéphane Lengrand (2009): *The lambda-context calculus (extended version)*. *Information and Computation* 207, pp. 1369–1400.

[8] Herman Geuvers & G.I Jojgov (2002): *Open Proofs and Open Terms: a Basis for Interactive Logic*. In Julian C. Bradfield, editor: *Proceedings of the 16th International on Computer Science Logic (CSL'03) Edinburgh, Scotland, UK, September 22-25*, Lecture Notes in Computer Science (LNCS 2471), Springer, pp. 537–552.

[9] Fausto Giunchiglia (1993): *Contextual Reasoning*. *Epistemologia* 16, pp. 145–164.

[10] Fausto Giunchiglia & Luciano Serafini (1994): *Multilanguage Hierarchical Logics or: How we can do Without Modal Logics*. *Artificial Intelligence* 65(1), pp. 29–70.

[11] Robert Glück & Jesper Jørgensen (1995): *Efficient Multi-level Generating Extensions for Program Specialization*. In Manuel V. Hermenegildo & S. Doaitse Swierstra, editors: *7th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'95)*, Lecture Notes in Computer Science LNCS(982), Springer, pp. 259–278.

[12] Robert Glück & Jesper Jørgensen (1996): *Fast Binding-Time Analysis for Multi-Level Specialization*. In Dines Bjørner, Manfred Broy & Igor V. Pottosin, editors: *2nd International Andrei Ershov Memorial Conference, Lecture Notes in Computer Science LNCS(1181)* 1181, Springer, pp. 261–272.

[13] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics*. *Journal of the ACM* 40(1), pp. 143–184.

[14] Aleksandar Nanevski, Frank Pfenning & Brigitte Pientka (2008): *Contextual modal type theory*. *ACM Transactions on Computational Logic* 9(3), pp. 1–49.

[15] Frank Pfenning (2007): *On a Logical Foundation for Explicit Substitutions*. In Simona Ronchi Della Rocca, editor: *8th International Conference on Typed Lambda Calculi and Applications (TLCA'07), Lecture Notes in Computer Science* 4583, Springer, p. 1.

[16] Brigitte Pientka (2008): *A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions*. In: *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, ACM Press, pp. 371–382.

[17] Brigitte Pientka & Joshua Dunfield (2010): *Beluga: a Framework for Programming and Reasoning with Deductive Systems (System Description)*. In Jürgen Giesl & Reiner Haehnle, editors: *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pp. 15–21.

[18] Brigitte Pientka & Frank Pfenning (2003): *Optimizing higher-order pattern unification*. In F. Baader, editor: *19th International Conference on Automated Deduction, Miami, USA*, Lecture Notes in Artificial Intelligence (LNAI) 2741, Springer-Verlag, pp. 473–487.

[19] Masahiko Sato, Takafumi Sakurai, Yukiyoshi Kameyama & Atsushi Igarashi (2003): *Calculi of Meta-varaibles*. In Matthias Baaz & Johann A. Makowsky, editors: *Proceedings of the 17th International on Computer Science Logic (CSL'03) Vienna, Austria, August 25-30*, Lecture Notes in Computer Science (LNCS 2803), Springer, pp. 484–497.

[20] Antonis Stampoulis & Zhong Shao (2010): *VeriML: typed computation of logical terms inside a language with effects*. In Paul Hudak & Stephanie Weirich, editors: *15th ACM SIGPLAN International Conference on Functional Programming (ICFP'10)*, ACM, pp. 333–344.

[21] Kevin Watkins, Iliano Cervesato, Frank Pfenning & David Walker (2002): *A Concurrent Logical Framework I: Judgments and Properties*. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University.

[22] Yosihiro Yuse & Atsushi Igarashi (2006): *A modal type system for multi-level generating extensions with persistent code*. In Annalisa Bossi & Michael J. Maher, editors: *8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'06)*, ACM, pp. 201–212.