Available online at www.sciencedirect.com
ScienceDirect

Electronic Notes in Theoretical Computer Science 228 (2009) 3–19

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# On the Expressivity of Minimal Generic Quantification

David Baelde

*INRIA Saclay - Île-de-France & LIX / École Polytechnique*
*Palaiseau, France*

**Abstract**

We come back to the initial design of the ∇ quantifier by Miller and Tiu, which we call minimal generic quantification. In the absence of fixed points, it is equivalent to seemingly stronger designs. However, several expected theorems about (co)inductive specifications can not be derived in that setting. We present a refinement of minimal generic quantification that brings the expected expressivity while keeping the minimal semantic, which we claim is useful to get natural adequate specifications. We build on the idea that generic quantification is not a logical connective but one that is defined, like negation in classical logics. This allows us to use the standard (co)induction rule, but obtain much more expressivity than before. We show classes of theorems that can now be derived in the logic, and present a few practical examples.

*Keywords:* Proof theory, generic quantification, fixed points, higher-order abstract syntax.

## 1 Introduction

A logic is a system that can represent objects, but also provides means to analyze them, reason about the relationships between these objects. Different logics allow more or less natural encodings of different object systems. The notion of *generic quantification* has been introduced in proof theory in order to obtain logics in which one can naturally specify systems involving variable bindings, such as programming languages, type systems or logics. Before detailing more that aspect, let us recall how algebraic specifications can be logically supported, for three increasingly demanding tasks: computing, model-checking and reasoning — from the proof-as-programs viewpoint, representing objects, then finite and finally infinite behaviour functions on those objects.

The purpose of logical frameworks is to allow declarative and adequate *representations* of various object systems. The classic example is that natural numbers are in bijection with normal intuitionistic proofs of $N \supset (N \supset N) \supset N$. Prolog built on that idea: the search for objects satisfying the specification is reduced to the search of some normal forms of proofs. Proof search in the framework provides a mean to *compute* according to the specification.

doi:10.1016/j.entcs.2008.12.113

A logic should also provide some means to reason about its judgments, and hence about encoded objects. This is the role of elimination rules. The least is to support case-analysis, which turns proof-search into *model-checking*, that is inspecting finite behaviours. This is obtained by moving to the notion of fixed points. In that setting, *nat* is defined as $\mu(\lambda X.\lambda n.n = 0 \vee \exists p.n = (s\ p) \wedge Xn)$, or simply $\mu(\lambda X.\top \vee X)$ if one regards proofs as programs. And the logic is equipped with the following case-analysis and back-chaining rules:

$$\frac{\Gamma, B(\mu B)t \vdash G}{\Gamma, \mu Bt \vdash G} \quad \frac{\Gamma \vdash B(\mu B)t}{\Gamma \vdash \mu Bt}$$

One does not need more for analyzing *finite* behaviours. Using the left unfolding rule, one can for example deduce that any natural number is either zero or a successor. It is interesting to notice that when hypothesis can be exhaustively analyzed in a finite number of steps, the initial rule (also called axiom rule) becomes useless: after the exhaustive case analysis, the goal can be built explictly from its fixed point definition. For example, the Bedwyr [2] system works under that assumption.

In order to reason about *infinite* behaviours or infinite collections of objects such as natural numbers, the initial rule becomes necessary. This raises the question of the identity of judgments, which is negligible in many cases but crucial for generic judgments. More expressivity is often needed, coming with the distinction of two dual kinds of fixed points and their associated deduction principles: least fixed points are represented by inductive specifications $\mu B$ and analyzed by induction; greatest fixed points are represented by coinductive specifications $\nu B$ and are built by coinduction. All this is well-established proof theory:

$$\frac{\Gamma, St \vdash G \quad BSx \vdash Sx}{\Gamma, \mu Bt \vdash G} \quad \frac{\Gamma \vdash St \quad Sx \vdash BSx}{\Gamma \vdash \nu Bt}$$

It is important to notice that the addition of more introspective abilities to the logic did not make it miss the initial target, that is adequate representations. Proofs of *nat* are still exactly natural numbers, and proofs of $nat \supset nat$ can still be seen as functions from natural numbers to natural numbers, computed via cut elimination. We shall now contrast that with the incomplete picture corresponding to the state of the art regarding the specification of systems involving variable binding.

The *higher-order abstract syntax* approach [12,6] pioneered by Twelf [14] and λProlog [9] provides an elegant way to represent systems involving variable binding, encoding the object-level binding by the notion of binding already present in the logic. In the proof-as-program approach, one uses the abstraction of proofs terms. In the λ-tree approach, term-level abstractions are used, *e.g.,* encoding the introduction of a generic variable by an universal quantification:

$$\forall f.\ (\forall x.\ term\ x \supset term\ (fx)) \supset term\ (abs\ (\lambda x.\ fx))$$

The $\alpha$-conversion of the logic is then inherited by the represented system. Of course, this technique only provides elegant encodings when the notions of scope and binding of the encoded system match those of the encoding logic.

However, the universal quantification only acts as a generic quantification when it occurs positively. On the left hand-side of a sequent, its "for all" semantic makes a difference: instead of introducing a generic variable, it is expecting a term to substitute for the variable. In spirit, generic quantification should not require to deal with terms inhabiting the types of generic variables. This mismatch required new proof-theoretic designs in order to obtain a satisfying support of case analysis. Several solutions [19,13,5] have been developped, revolving around an essential idea: keeping the generic variables in a context local to the formula. We shall focus on the earliest design, which is also the simplest incarnation of this idea: the $\nabla$ quantifier [10,11]. It extends sequents with generic contexts $\sigma$ surrounding formulas, denoted by $\sigma \triangleright P\sigma$ — we write $P\sigma$ to indicate that the generic variables can occur in the formula, also implicitly assuming that they don't occur free in $P$ anymore. We call this early design "minimal generic quantification" because of the absence of any structural rule for generic contexts. Despite its minimality, it is fully satisfying for reasoning about finite behaviours, where the identity of judgments does not matter. Notably, Miller and Tiu have shown that this core support of generic quantification, combined with least and greatest fixed points in the logic LINC [17], is enough for providing a fully declarative specification of finite $\pi$-calculus and its bisimulation [18]. Again, the ability to adequately represent objects has not been affected, since $\nabla$ and $\forall$ are identical when restricted to positive occurences. When representing objects in the logic, we are actually only moving object-level binders to the formula-level generic context, which keeps an exact track of the scope.

Things get significantly more complicated with infinite behaviours. In LINC, the initial rule requires an exact match (modulo $\alpha$-conversion) of the generic contexts:

$$\frac{}{\sigma \triangleright P\sigma \vdash \sigma \triangleright P\sigma}$$

Without any structural rule on the generic context, this often seems too restrictive. The design of the induction rule also turns out to be problematic. In the end, the interplay of minimal generic quantification and fixed points in LINC lacks in expressivity. Theorems which should intuitively hold are underivable, *e.g.*, $\forall x.\ (\nabla y.\ \mathit{nat}\ x) \supset \mathit{nat}\ x$. This has lead to the development of radically different notions of generic quantification. It is typically made stronger, in order to always have $\forall x.\ Px \supset \nabla x.\ Px$ and $\nabla x.\ Px \supset \exists x.\ Px$, which implies $P \equiv \nabla x.\ P$ when $x$ does not occur free in $P$. In our opinion, this is not a satisfying solution: this stronger semantic of generic quantification does not match the needs of some specifications. Consider for example the specification of provability in some object logic. The natural fixed-point clause for universal quantification is:

$$\mathit{prove}\ \Gamma\ (\hat{\forall}(\lambda x.\ Px)) := \nabla x.\ \mathit{prove}\ \Gamma\ (Px)$$

With minimal generic quantification, the generic context exactly represents the signature of the object sequent. With a stronger generic quantification this is no longer true. Indeed, we could derive immediately that $\mathit{prove}\ \Gamma\ (\hat{\forall}(\lambda x.\ P)) \supset \mathit{prove}\ \Gamma\ P$. This is not wanted in all cases. In particular, in presence of a cut rule, one would expect that a proof of such a statement provides a way to eliminate cuts for which the cut-formula contains an occurence of the (seemingly) vacuous $\hat{\forall}$. This inadequacy can be seen as analogous to the impossibility to represent directly a linear logic in an intuitionistic framework.

In this paper, we come back to the design of minimal generic quantification. In Section 2 we propose the logic $\mu\mathrm{LJ}^{\nabla}$ as an alternative that brings the expected expressivity. The key is a better interaction of fixed point constructions and generic quantificaton, which is not treated as a logical connective anymore. Structural rules on the generic context will still not be admitted in general, allowing adequate representations of systems following a strict management of variables. In Section 3 we study the meta-theory of $\mu\mathrm{LJ}^{\nabla}$, notably exhibiting a large syntactic class of formulas for which structural rules can in fact be derived. Finally, we illustrate how the logic can be practically used on a few significant examples in Section 4.

## 2 Definitions

We shall work on two logics: $\mu\mathrm{LJ}^{\nabla_0}$ which incarnates the initial design of minimal generic quantification, and its corrected version $\mu\mathrm{LJ}^{\nabla}$ where $\nabla$ is no more a logical connective. In both systems, we are not going to consider atoms, that is predicate constants. By essence, atoms have an undefined behaviour, unlike fixed points. If one wants to obtain $\nabla x.\ p \supset p$ for an atom $p$, there is no other option than adding it as a rule in the logic, whereas with a fixed point the theorem could be obtained by (co)induction. Not considering atoms is motivated practically, because users of a logic usually work on defined notions, but also theoretically, as we find important for the logician to study what can be derived from the simplest definition before playing the game of choosing which axioms to force uniformly.

In the following, the type of formulas is $o$, and $\gamma$ denotes a term type, that is any simple type in which $o$ does not occur. Terms are denoted by $u, v, t$; formulas by $P, Q, S$; term variables by $x, y, z$; and predicate variables by $p, q$. We write vectors as $\overrightarrow{t}$, and sometimes $(t_i)_i$ for readability. In all systems considered in this paper, formulas are of the following forms:

$$P ::= P \wedge P \mid P \vee P \mid P \supset P \mid \top \mid \bot$$
$$\mid\ \exists_\gamma x.Px \mid \forall_\gamma x.Px \mid \nabla_\gamma x.Px \mid s \overset{\gamma}{=} t \mid s \overset{\gamma}{\neq} t \mid \mu_{\gamma_1 \ldots \gamma_n} B \overrightarrow{t} \mid \nu_{\gamma_1 \ldots \gamma_n} B \overrightarrow{t}$$

The quantifiers have type $(\gamma \to o) \to o$ and the equality and inequality have type $\gamma \to \gamma \to o$. The connectives $\mu$ and $\nu$ have type $(\tau \to \tau) \to \tau$ where $\tau$ is $\gamma_1 \to \cdots \to \gamma_n \to o$ for some arity $n \geq 0$. Formulas with top-level connective $\mu$ or $\nu$ are called fixed point expressions and can be arbitrarily nested. The first argument of a fixed point expression, denoted by $B$, is called its *body*. For consistency, it is required to be *monotonic*: negative occurences of the bound predicate variable are forbidden.

Figure 1 defines $\mu\mathrm{LJ}^{=}$, the basic system on top of which are built logics supporting generic quantification. It consists in first-order propositional intuitionistic logic extended with fixed points and equality. For background and details about definitions, fixed points and the treatment of equality, we refer the reader to [16,7,17,2].

Generic contexts are lists of typed term variables, denoted by $\sigma$ or $\zeta$. If $\sigma$ is a generic context $(x_1 : \gamma_1, \ldots, x_n : \gamma_n)$ we denote by $\sigma \to \gamma$ the type $\gamma_1 \to \ldots \to \gamma_n \to \gamma$, and call it $\gamma$ *lifted over* $\sigma$. Analogously, we talk of lifted variables when they have a lifted type. We also use a generic context as a list of terms, writing

Propositional intuitionistic logic

$$\frac{}{\Gamma, P \vdash P} \quad \frac{}{\Gamma, \bot \vdash P} \quad \frac{}{\Gamma \vdash \top}$$

$$\frac{\Gamma, P, P' \vdash Q}{\Gamma, P \wedge P' \vdash Q} \quad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}$$

$$\frac{\Gamma, P \vdash Q \quad \Gamma, P' \vdash Q}{\Gamma, P \vee P' \vdash Q} \quad \frac{\Gamma \vdash P}{\Gamma \vdash P \vee P'} \quad \frac{\Gamma \vdash P'}{\Gamma \vdash P \vee P'}$$

$$\frac{\Gamma \vdash P \quad \Gamma, P' \vdash Q}{\Gamma, P \supset P' \vdash Q} \quad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \supset Q}$$

First-order structure

$$\frac{\Gamma, Ph \vdash Q}{\Gamma, \exists x.Px \vdash Q} \quad \frac{\Gamma \vdash Pt}{\Gamma \vdash \exists x.Px}$$

$$\frac{\Gamma, Pt \vdash Q}{\Gamma, \forall x.Px \vdash Q} \quad \frac{\Gamma \vdash Ph}{\Gamma \vdash \forall x.Px}$$

$$\frac{\{(\Gamma \vdash Q)\theta : t\theta \doteq t'\theta\}}{\Gamma, t = t' \vdash Q} \quad \frac{}{\Gamma \vdash t = t}$$

Fixed points

$$\frac{\Gamma, St \vdash P \quad BSx \vdash Sx}{\Gamma, \mu Bt \vdash P} \quad \frac{\Gamma \vdash B(\mu B)t}{\Gamma \vdash \mu Bt}$$

$$\frac{\Gamma, B(\nu B)t \vdash P}{\Gamma, \nu Bt \vdash P} \quad \frac{\Gamma \vdash St \quad Sx \vdash BSx}{\Gamma \vdash \nu Bt}$$

Fig. 1. Inference rules for $\mu$LJ$^{=}$ and $\mu$LJ$^{\nabla}$. The eigenvariable $h$ does not occur free in the conclusion sequent of $\forall$-R and $\exists$-L, and $S$ is closed in $\mu$-L and $\nu$-R.

$(F\sigma)$ for $((Fx_1)\ldots x_n)$. Similarly, $f(x_i)_i$ stands for $((fx_1)\ldots x_n)$.

Type annotations are omitted for conciseness most of the time, but can be recovered from the context. In order to make type inference easier, we use the convention of naming $x'$ a lifted version of $x$. Finally, we use the convention that when we write $(\lambda a.t)$, the variable $a$ does not occur free in $t$.

## 2.1 The logic $\mu LJ^{\nabla_0}$

The logic $\mu$LJ$^{\nabla_0}$ is essentially the same as LINC [17]. The only difference lies in the use of fixed points rather than definitions. This slight shift of point of view brings more expressivity, but most importantly gives more flexibility when working on the logic.

The system is obtained from $\mu$LJ$^{=}$ by enriching the sequent structure, surrounding each formula by a local *generic context* which binds *generic variables* in the formula. A formula surrounded by its generic context forms a generic judgment, denoted by $G$. Variables are introduced in the new context by generic quantification:

$$\frac{\Gamma, (\sigma, x) \rhd P\sigma x \vdash G}{\Gamma, \sigma \rhd \nabla x.P\sigma x \vdash G} \quad \frac{\Gamma \vdash (\sigma, x) \rhd P\sigma x}{\Gamma \vdash \sigma \rhd \nabla x.P\sigma x}$$

Propositional deduction rules are adapted in an orthogonal fashion. Under the generic context, the reasoning takes place as before, for example:

$$\frac{}{\sigma \rhd P\sigma \vdash \sigma \rhd P\sigma} \quad \frac{\Gamma \vdash \sigma \rhd P\sigma \quad \Gamma, \sigma \rhd Q\sigma \vdash G}{\Gamma, \sigma \rhd P\sigma \supset Q\sigma \vdash G} \quad \frac{\Gamma, \sigma \rhd P\sigma \vdash \sigma \rhd Q\sigma}{\Gamma \vdash \sigma \rhd P\sigma \supset Q\sigma}$$

The first-order rules interact with the generic context, causing a lifting of terms and term variables. In the rules for universal quantification, if $x$ has type $\gamma$ then $t$ and the eigenvariable $h$ have type $\sigma \to \gamma$:

$$\frac{\Sigma; \Gamma, \sigma \rhd P\sigma(t\sigma) \vdash G}{\Sigma; \Gamma, \sigma \rhd \forall x. P\sigma x \vdash G} \quad \frac{\Sigma, h; \Gamma \vdash \sigma \rhd P\sigma(h\sigma)}{\Sigma; \Gamma \vdash \sigma \rhd \forall x. P\sigma x}$$

$$\frac{\{(\Gamma \vdash G)\theta : t\theta \doteq t'\theta\}}{\Gamma, \sigma \rhd t\sigma = t'\sigma \vdash G} \quad \frac{}{\Gamma \vdash \sigma \rhd t\sigma = t\sigma}$$

At this point, we have described how the logic treats *finite behaviour* formulas, those that do not involve fixed points. In that setting, the deduction rules can be read as equivalences, for example $\nabla x.\ Px \wedge Qx \equiv (\nabla x.\ Px) \wedge (\nabla x.\ Qx)$, which allow to eliminate $\nabla$ by pushing it down through logical connectives, eventually disappearing, for example thanks to $(\nabla x.\ ux = vx) \equiv u = v$. A similar observation is that any proof of a finite behaviour can be $\eta$-expanded until it does not use the initial rule anymore, but where the only leafs are the rules for $\top, \bot$ and equality. Both observations imply that there is no need to ever compare two generic contexts, and hence no need to consider structural rules for them. Note, however that the dynamic aspect of the second one makes it stronger; for example it extends to fixed points, giving rise to the Bedwyr system.

The fixed point rules are adapted as follows:

$$\frac{\Gamma, \sigma \rhd S(t\sigma) \vdash P \quad BSx \vdash Sx}{\Gamma, \sigma \rhd \mu B(t\sigma) \vdash P} \quad \frac{\Gamma \vdash \sigma \rhd B(\mu B)(t\sigma)}{\Gamma \vdash \sigma \rhd \mu B(t\sigma)}$$

The extended unfolding rule states that liftings of a fixed point unfold just like the original version. The induction rule seems less natural, essentially stating that liftings of the invariants of a fixed point are invariants of its liftings. Tiu noticed [17] that it was too weak, since it does not allow any modification of the generic context $\sigma$ when inducting under it. As a consequence, it is impossible to prove things like $\forall x.\ (\nabla a.\ nat\ x) \supset nat\ x$. This seems unfortunate because *nat*, defined as a fixed point, clearly does not rely on the generic context.

## 2.2 *$\mu LJ^{\nabla}$: treating $\nabla$ as a non-logical connective*

The self-duality of generic quantification, as well as its ability to commute with almost all connectives of $\mu\mathrm{LJ}^{\nabla_0}$, suggests that it might not be a *logical connective* but rather a *defined* one, like negation in classical logics. Indeed, we shall define a formula transformation $\phi$ that transforms a formula into one where the $\nabla$ quantifiers only occur above bound predicate variables inside fixed point constructions. This is in slight contrast with the classical negation which can be eliminated statically even inside fixed points, because of the monotonicity constraint. The important point remains: since $\nabla$ does not occur anymore at toplevel in a sequent, it loses its logical role.

The logic $\mu\mathrm{LJ}^{\nabla}$ has the same rules as $\mu\mathrm{LJ}^{=}$, *cf.* Figure 1. Unlike $\mu\mathrm{LJ}^{\nabla_0}$, it has no extra sequent structure. The rules for fixed points are written the same way, but the implicit elimination of toplevel $\nabla$ quantifiers which might occur in $BS$ will play a critical role.

$$\frac{\Gamma, St \vdash P \quad BSx \vdash Sx}{\Gamma, \mu Bt \vdash P} \quad \frac{\Gamma \vdash B(\mu B)t}{\Gamma \vdash \mu Bt}$$

We define the connective $\nabla$ by identifying a formula $F$ at toplevel in a sequent with $\phi(F)$ where $\nabla$ does not occur anymore except inside fixed point bodies. The transformation is parametrized by two contexts initially empty, and written $\phi^{\Gamma}_{\sigma}(P|\Gamma|\sigma)$. Here, $\sigma$ is a generic context, *i.e.,* a list of term variables. The context $\Gamma$ contains associations of the form $\langle p, \sigma, p'\rangle$ where $p$ is a predicate variable of type $\gamma$, $\sigma = x_1 : \gamma_1, \ldots, x_n : \gamma_n$ is a generic context and $p'$ is an other predicate variable of type $\gamma_1 \rightarrow \ldots \rightarrow \gamma_n \rightarrow \gamma$. The support of $\Gamma$, written $|\Gamma|$, is the variables $p$. As indicated by the notation in $P|\Gamma|\sigma$, the predicate variables of $|\Gamma|$ and the term variables of $\sigma$ may occur in the original formula. These occurences are bound by the two contexts. In the transformed formula only the $p'$ will be found. Finally, the order does not matter in $\Gamma$ unlike in $\sigma$. The inductive definition of the transformation is given on Figure 2.

That full definition can be reduced to the definition of the behaviour of $\phi$ on term and predicate abstractions and variables, extended in an orthogonal way to the full language of formulas. For doing so, one should carefully separate the binding and logical aspects of $\forall, \exists, \mu$ and $\nu$. Along these lines, we shall write $\phi$ applied not only to formulas but also to formulas abstracted over terms such as in $\phi_{\sigma}(\mu B)$, or formulas and terms such as in $\mu\phi_{\sigma}(B)$.

$$\phi^{\Gamma}_{\sigma}(\nabla x.\ P|\Gamma|\sigma x) \equiv \phi^{\Gamma}_{\sigma,x}(P|\Gamma|\sigma x)$$
$$\phi^{\Gamma}_{\sigma}(\forall x.\ P|\Gamma|\sigma x) \equiv \forall x'.\ \phi^{\Gamma}_{\sigma}(P|\Gamma|\sigma(x'\sigma)) \quad \phi^{\Gamma}_{\sigma}(\exists x.\ P|\Gamma|\sigma x) \equiv \exists x'.\ \phi^{\Gamma}_{\sigma}(P|\Gamma|\sigma(x'\sigma))$$
$$\phi^{\Gamma}_{\sigma}(\mu(\lambda p \lambda \overrightarrow{x}.\ B|\Gamma|p\sigma\overrightarrow{x})\overrightarrow{(t\sigma)}) \equiv \mu(\lambda p' \lambda \overrightarrow{y}.\ \phi^{\Gamma,\langle p,\sigma,p'\rangle}_{\sigma}(B|\Gamma|p\sigma\overrightarrow{(y\sigma)}))\,\overrightarrow{t}$$
$$\phi^{\Gamma}_{\sigma}(\nu(\lambda p \lambda \overrightarrow{x}.\ B|\Gamma|p\sigma\overrightarrow{x})\overrightarrow{(t\sigma)}) \equiv \nu(\lambda p' \lambda \overrightarrow{y}.\ \phi^{\Gamma,\langle p,\sigma,p'\rangle}_{\sigma}(B|\Gamma|p\sigma\overrightarrow{(y\sigma)}))\,\overrightarrow{t}$$
$$\phi^{\Gamma}_{\sigma}(a(t\sigma)) \equiv \nabla\sigma.\ a(t\sigma) \quad \phi^{\Gamma,\langle p,\sigma,p'\rangle}_{\sigma\sigma'}(p(t\sigma\sigma')) \equiv \nabla\sigma'.\ p'(\lambda\sigma.\ t\sigma\sigma')$$
$$\phi^{\Gamma}_{\sigma}(u\sigma = v\sigma) \equiv u = v \quad \phi^{\Gamma}_{\sigma}(\top) \equiv \top \quad \phi^{\Gamma}_{\sigma}(\top) \equiv \top$$
$$\phi^{\Gamma}_{\sigma}(P|\Gamma|\sigma \wedge Q|\Gamma|\sigma) \equiv \phi^{\Gamma}_{\sigma}(P|\Gamma|\sigma) \wedge \phi^{\Gamma}_{\sigma}(Q|\Gamma|\sigma)$$
$$\phi^{\Gamma}_{\sigma}(P|\Gamma|\sigma \vee Q|\Gamma|\sigma) \equiv \phi^{\Gamma}_{\sigma}(P|\Gamma|\sigma) \vee \phi^{\Gamma}_{\sigma}(Q|\Gamma|\sigma)$$
$$\phi^{\Gamma}_{\sigma}(P|\Gamma|\sigma \supset Q|\Gamma|\sigma) \equiv \phi^{\Gamma}_{\sigma}(P|\Gamma|\sigma) \supset \phi^{\Gamma}_{\sigma}(Q|\Gamma|\sigma)$$

Fig. 2. The transformation $\phi$ defining $\nabla$

**Example 2.1** The elimination of $\nabla$ on finite-behaviour formulas described for $\mu\mathrm{LJ}^{\nabla_0}$ in Section 2.1 is now an identity. In $\mu\mathrm{LJ}^{\nabla}$ the two following formulas are identified:

$$\nabla x.\forall y.\nabla z.y = z \supset \bot \quad \equiv \quad \forall y'.(\lambda xz.y'x) = (\lambda xz.z) \supset \bot$$

**Example 2.2** We define the well-formedness of terms in a purely abstractive language, assuming constants $nil :: lst$, $cons :: \alpha \rightarrow lst \rightarrow lst$ and $abs :: (\alpha \rightarrow tm) \rightarrow tm$, as follows:

$$mem \stackrel{def}{=} \mu(\lambda M \lambda x \lambda \Gamma.\ \exists hd \exists tl.\ \Gamma = (cons\ hd\ tl) \wedge (x = hd \vee M\ x\ tl))$$
$$term \stackrel{def}{=} \mu(\lambda T \lambda \Gamma \lambda t.\ mem\ t\ \Gamma \vee \exists f.\ t = abs\ f \wedge \nabla_\alpha x.\ T\ (cons\ x\ \Gamma)\ (f\ x))$$

It does not look very different from $\mu\mathrm{LJ}^{\nabla_0}$. But the $\nabla$ inside the body of the fixed point *term* should be read as a suspensed lifting, waiting for the instantiation of $T$. In a sense, the fixed point does not only define a predicate, but rather a family of liftings. Let us now consider the lifting of the definitions of *mem* and *term*. For readability we lift over an other type than $\alpha$:

$$\phi_{(y:\beta)}(mem) \stackrel{def}{=}$$
$$\mu(\lambda M' \lambda x' \lambda \Gamma'.\ \exists hd' \exists tl'.\ \Gamma' = (\lambda_\beta y.\ cons\ (hd'\ y)\ (tl'\ y)) \wedge (x' = hd' \vee M'\ x'\ tl'))$$
$$\phi_{(y:\beta)}(term) \stackrel{def}{=} \mu(\lambda T' \lambda \Gamma' \lambda t'.\ \phi_{(y:\beta)}(mem)\ t'\ \Gamma' \vee$$
$$\exists f'.\ t' = (\lambda_\beta y. abs\ (f' y)) \wedge \nabla x.\ T'\ (\lambda_\beta y.\ cons\ x\ (\Gamma' y))\ (\lambda_\beta y.\ f' y x))$$

In other words, the induction on $\phi_{(y:\beta)}(term)$ corresponds to the following principle, where $S$ is the tentative invariant:

$$(\forall \Gamma' \forall x'.\ \phi_{(y:\beta)}(mem)\ x'\ \Gamma' \supset S\ \Gamma'\ x')$$
$$\supset (\forall \Gamma' \forall f'.\ \phi_{(x:\alpha)}(S)\ (\lambda x \lambda y.\ cons\ x\ (\Gamma' y))\ (\lambda x \lambda y.\ f' y x) \supset S\ \Gamma'\ (\lambda y.\ abs\ (f' y)))$$
$$\supset (\forall \Gamma' \forall t'.\ \phi_{(y:\beta)}(term)\ \Gamma'\ t' \supset S\ \Gamma'\ t')$$

For example it can be used with the invariant:

$$S := \lambda \Gamma' \lambda x'. \forall \Gamma \forall x.\ (\Gamma' = (\lambda y. \Gamma) \wedge x' = (\lambda y. x)) \supset term\ \Gamma\ x$$

After a similar sub-induction on $\phi_{(y:\beta)}(mem)$ one will have obtained a derivation of:

$$\forall \Gamma \forall x.\ (\nabla y.\ term\ \Gamma\ x) \supset term\ \Gamma\ x$$

## 3 The theory of $\mu\mathrm{LJ}^{\nabla}$

The lifting transformation behaves nicely with respect to first-order abstraction, *i.e.*, $\phi_\sigma(F\sigma(t\sigma)) \equiv ((\lambda x'.\ \phi_\sigma(F\sigma(x'\sigma)))t)$. Unfortunately, the same does not hold for second-order abstraction: $\phi_\sigma(B)\phi_\sigma(S)$ is not necessarily the same as $\phi_\sigma(BS)$. Consider an abstraction $B := (\lambda p \ldots \nabla \sigma'(\ldots p \ldots))$. In $\phi_\sigma(BS)$ the occurence of $p$ will become $\phi_{\sigma\sigma'}(S)$ whereas in $\phi_\sigma(B)\phi_\sigma(S)$ it becomes $\phi_{\sigma'}(\phi_\sigma(S))$.

This is technically complicating the metatheory, but our attempts to change the definition and avoid that have been unsuccessful. In fact, this permutation of names can be understood as inherent to the identifaction of $\nabla\sigma.\ \mu B$ and $\mu\phi_\sigma(B)$. Indeed, these two fixed points reveal two different prefixes of generic quantifiers after $n$ unfoldings, respectively $\sigma(\sigma')^n$ and $(\sigma')^n\sigma$.

We shall establish, however, that the permutation of names does not affect provability. This can be derived in the logic itself, not only at the meta level. It will provide a way to bridge the gap between $\phi_\sigma(BS)$ and $\phi_\sigma(B)\phi_\sigma(S)$ by translating all permuted instances.

**Proposition 3.1** *For any formula $P$, and any two generic contexts $\sigma$ and $\sigma^*$ permutations of each other, it is provable that $(\nabla\sigma.\ P\sigma) \supset (\nabla\sigma^*.\ P\sigma)$.*

The proof basically builds a corrected $\eta$-expansion, carrying the permutability from elementary formulas (equality, $\top$ and $\bot$) through all connectives including $\mu$ and $\nu$. In a sense, there is nothing clever in it: at any point there is only one choice that keeps things well-typed. That said, it is a good test for the logic to check that everything goes as expected. The details of this proof, as for other propositions of this section, can be found in [1].

**Corollary 3.2** *Proposition 3.1 actually provides a mean to transform a derivation into another one establishing the same sequent where some instances of $\phi_\sigma(F)$ have been replaced by some permutation $\phi_{\sigma^*}(F)$: this is done by cutting against $\eta$-expansions of the derivations provided by the proposition. In particular, it allows to compensate the difference between $\phi_\sigma(B)\phi_\sigma(S)$ and $\phi_\sigma(BS)$.*

**Definition 3.3** We extend the notion of lifting to sequents:

$$\phi_\sigma(x_1,\ldots,x_n; P_1(x_i)_i,\ldots,P_m(x_i)_i \vdash Q(x_i)_i) :=$$
$$x'_1,\ldots,x'_n; \phi_\sigma(P_1(x'_i\sigma)_i),\ldots,\phi_\sigma(P_m(x'_i\sigma)_i) \vdash \phi_\sigma(Q(x'_i\sigma)_i)$$

**Proposition 3.4 (Lifting derivations)** *For any $\sigma$, the provability of $\Sigma;\Gamma \vdash P$ implies that of $\phi_\sigma(\Sigma;\Gamma\vdash P)$* [1] *.*

This allows one to read a proof of $(\forall t.\ (\nabla a.\ p\ t) \supset p\ t)$ as not only establishing that the provability of $p$ in *the context* of one unused generic variable entails that of $p$ in the empty context, but more generally that the provability is stable by removal of an unused variable from *any* context. This is what makes our system expressive without any need for concrete manipulations of the context and other complex devices such as quantifications over all generic contexts.

**Proposition 3.5 (Conservativity & expressivity)** *We call 0-provability the provability without any use of the (co)induction rules. Let $P$ be a formula, possibly involving $\nabla$ quantifications.*

(i) *The 0-provability of $P$ in $\mu LJ^{\nabla_0}$ is equivalent to its 0-provability in $\mu LJ^{\nabla}$.*

(ii) *Moreover, the provability of $P$ in $\mu LJ^{\nabla_0}$ implies its provability in $\mu LJ^{\nabla}$,*

(iii) *but the converse is false.*

**Proof.** Each direction of (i) is done by induction on the 0-derivation. Both are straightforward proof transformation similar to those detailed before, including corrective cuts (*cf.* Corollary 3.2) as in Proposition 3.4. For (ii) we add the translation

[1] We do not follow the notational conventions here: of course, the variables of $\Sigma$ can occur in $\Gamma$ and $P$, unlike those of $\sigma$.

of an induction in $\mu\mathrm{LJ}^{\nabla_0}$ to $\mu\mathrm{LJ}^{\nabla}$, which amounts to lift the invariant and the invariance proof. Finally, (iii) shall be seen later, with the ability to weaken the generic context in some cases in $\mu\mathrm{LJ}^{\nabla}$, which is impossible in $\mu\mathrm{LJ}^{\nabla_0}$. □

## 3.1 Cut-elimination

We adapt the proof reductions involved in cut-elimination, and argue that the termination is not affected, leaving a detailed proof of that for future work. The only novelty is the transformation $\phi$. It only affects second-order instantiations in fixed point unfoldings, induction and coinduction. It does not affect several important properties of the system: proofs can be instantiated, the signature can be enriched, etc. The non-trivial part is adapting the reduction for eliminating a cut on a fixed point. We only show the case of the least fixed point, the greatest being similar.

The essential reduction for least fixed point is the following:

$$\dfrac{\dfrac{\dfrac{\Pi_S}{BS\vec{x} \vdash S\vec{x}} \quad \dfrac{\Pi'}{\Gamma, S\vec{t} \vdash P}}{\Gamma, \mu B\vec{t} \vdash P} \quad \dfrac{\Pi}{\Gamma \vdash \mu B\vec{t}}}{\Gamma \vdash P} \longrightarrow \dfrac{\dfrac{\Pi'}{\Gamma, S\vec{t} \vdash P} \quad \dfrac{fold(\Pi, \Pi_S)}{\Gamma \vdash S\vec{t}}}{\Gamma \vdash P}$$

Where the $fold(\Pi, \Pi_S)$ transformation replaces in $\Pi$ all unfoldings of $\mu B$ by a cut against $\Pi_S$. More precisely, since the unfoldings might be lifted, occurences of $\phi_\sigma(\mu B) = \mu(\phi_\sigma(B))$ are replaced by $\phi_\sigma(S)$:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \phi_\sigma(B)(\mu(\phi_\sigma(B)))\vec{t}}}{\Gamma \vdash \mu(\phi_\sigma(B))\vec{t}}\ \mu R \longrightarrow \dfrac{\dfrac{\vdots}{\phi_\sigma(B)\phi_\sigma(S)\vec{t} \vdash \phi_\sigma(S)\vec{t}} \quad \Gamma \vdash \phi_\sigma(B)\phi_\sigma(S)\vec{t}}{\Gamma \vdash \phi_\sigma(S)\vec{t}}\ cut$$

To complete this, one must build from $\Pi_S$, for a given $\sigma$ and $\vec{t}$, a proof of $\phi_\sigma(B)\phi_\sigma(S)\vec{t} \vdash \phi_\sigma(S)\vec{t}$. Using Proposition 3.4 on $\Pi_S$ we get a derivation of $\phi_\sigma(BS\vec{x}) \vdash \phi_\sigma(S\vec{x})$. Then, Corallary 3.2 gives a proof of $\phi_\sigma(B)\phi_\sigma(S)\vec{x} \vdash \phi_\sigma(S)\vec{x}$ where $\vec{x}$ can finally be instantiated by $\vec{t}$.

We leave the termination of the reduction for further work. Strictly speaking, cut-elimination has not been established even for $\mu\mathrm{LJ}^{\nabla_0}$, which does not have the stratification constraints on which is built the proof for LINC [17]. However, based on earlier work on $\mu\mathrm{MALL}^{=}$ [3], we believe that it holds under the simple constraint of monotonicity. But even with the stratification constraint on fixed points, the termination of cut-elimination for LINC does not carry easily to $\mu\mathrm{LJ}^{\nabla}$, because of the extra cuts inserted in the above reduction for translating between $\phi(BS)$ and $\phi(B)\phi(S)$.

## 3.2 Structural rules on the generic context

Minimal generic quantification in $\mu\mathrm{LJ}^{\nabla_0}$ did not assume an infinity of undistinguishable generic variables, hence the absence of an immediate way of obtaining strengthening or exchange on generic contexts. This has not been changed with

$\mu$LJ$^\nabla$. For example, $(\nabla_\gamma x.\ \top) \supset (\exists_\gamma x.\ \top)$ still can not be derived without assuming the non-vacuity of $\gamma$, which would make even $(\exists_\gamma x.\ \top)$ alone derivable. Symmetrically, $\forall$ does not imply $\nabla$ in general. And vacuous generic quantifications can not *a priori* be added or removed.

We show, however, that the missing generic strengthening and weakening are actually derivable for a reasonable class of formulas. On such formulas, the minimality does not make generic quantification weaker than other approaches, just as in the finite behaviour case. The essential idea for obtaining generic strengthening, that is $(\nabla x.\ P) \supset P$, is to forbid positive occurences of existential quantification, unless they are guarded by a formula that ensures that the existential variable does not depend on the extra generic variable $x$.

**Definition 3.6** A *guard* is a formula $G$ such that for any $\sigma$:

$$\forall y'.\ \phi_{x\sigma}(G\sigma(y'x\sigma)) \supset \exists y.\ y' = (\lambda x.\ y) \wedge \phi_\sigma(G\sigma(y\sigma))$$

A typical guard would be an equality $(\lambda\sigma\lambda y.\ u\sigma y = v\sigma y)$ such that all unifiers of $(\lambda x\sigma.\ u\sigma(y'x\sigma) = v\sigma(y'x\sigma))$ set $y' := \lambda x\lambda\sigma.y\sigma$ for some $y$. This holds for the equalities found in most fixed point definitions, which fully define the newly introduced existential variable as a compound or a sub-term of the pre-existing terms.

**Definition 3.7** The fragments $\mathcal{W}$ and $\mathcal{S}$ (respectively standing for $\mathcal{W}$eakening and $\mathcal{S}$trengthening) are mutually defined by the following grammar, where $\mathcal{G}$ denotes a guard:

$$\begin{aligned}\mathcal{W} &::= \mathcal{W} \wedge \mathcal{W} \mid \mathcal{W} \vee \mathcal{W} \mid \top \mid \bot \mid u = v \mid \nabla x.\mathcal{W} \mid \mu\mathcal{W} \mid \nu\mathcal{W} \\ &\quad \mid \mathcal{S} \supset \mathcal{W} \mid \exists x.\ W \mid \forall x.\ \mathcal{G}x \supset \mathcal{W} \\ \mathcal{S} &::= \mathcal{S} \wedge \mathcal{S} \mid \mathcal{S} \vee \mathcal{S} \mid \top \mid \bot \mid u = v \mid \nabla x.\mathcal{S} \mid \mu\mathcal{S} \mid \nu\mathcal{S} \\ &\quad \mid \mathcal{W} \supset \mathcal{S} \mid \forall x.\ \mathcal{S} \mid \exists x.\ \mathcal{G}x \wedge \mathcal{S}\end{aligned}$$

**Proposition 3.8** *For any formula $W \in \mathcal{W}$ (resp. $S \in \mathcal{S}$) it is provable that $W \supset \nabla x.\ W$ (resp. $\nabla x.\ S \supset S$).*

An other way to put it, for example, is that the following rules are admissible in $\mu$LJ$^\nabla$:

$$\frac{\Gamma, \sigma, \sigma' \triangleright S\sigma\sigma' \vdash G}{\Gamma, \sigma, x, \sigma' \triangleright S\sigma\sigma' \vdash G} \quad \frac{\Gamma \vdash \sigma, \sigma' \triangleright W\sigma\sigma'}{\Gamma \vdash \sigma, x, \sigma' \triangleright W\sigma\sigma'}$$

Proposition 3.8 is very useful in pratice. Indeed, most common fixed points (*nat*, *mem*, *append*, *term*, *typeof*...) are both in $\mathcal{S}$ and $\mathcal{W}$: they do not involve any universal quantification and the existential quantifications are guarded by equalities. Most of the time, the existentials are actually very weak in that they do not even require any invention, such as in $\exists y.\ x = s\ y \wedge \ldots$.

**Example 3.9** Coming back to the introductory discussion about adequacy, the typical example of a fixed point that does not fall in $\mathcal{S}$ is *provability in an object* *logic with a cut rule.* Indeed, the cut rule involves an existential quantification that is not guarded at all, and can notably range over variables present in the object-level signature (that is the generic context) even though not anywhere else in the object sequent (that is the parameters of our fixed point). Thus, deriving $\mathit{prove}\ \Gamma\ (\hat{\forall}(\lambda x.\ P)) \supset \mathit{prove}\ \Gamma\ P$ necessarily involves the description of at least a limited form of cut-elimination that removes the cut-formulas involving the allegedly useless generic variable $x$. It seems reasonable that such a non-trivial property of the encoded logic is not given for free by the logical framework but has to be worked out the user.

Another example of guard would be $\mathit{nat}$ itself, as it forces its parameter to be fully defined in terms of the constants zero and successor. We let the reader check that the following can be derived in $\mu\mathrm{LJ}^{\nabla}$: $(\nabla x.\ \mathit{nat}(y'x) \supset \exists y.\ y' = (\lambda x.y) \wedge \mathit{nat}\ y$. More interestingly, it should be possible to characterize a fragment of valid guards, which could build on top of guards like our $\mathcal{W}$ and $\mathcal{S}$ did, such that from a basic guard (*e.g.,* $\lambda y.\ x = s\ y$) one could derive an other (*e.g.,* $\mathit{nat}$), and an other (*e.g.,* $\mathit{natlist}$), etc.

## 4 Practical use of $\mu\mathrm{LJ}^{\nabla}$

We describe here a few significant examples of what can be done with $\mu\mathrm{LJ}^{\nabla}$. These examples have been checked [2] using the interactive theorem prover Taci [15], an unreleased tool from the Slimmer project. Taci is a simple generic tactic-based interactive theorem prover, in which a logician can plug his own logic as an OCaml module. Starting with the initial specification of $\mu\mathrm{LJ}^{=}$, we added support for the transformation $\phi$, obtaining a convenient implementation of $\mu\mathrm{LJ}^{\nabla}$. Formulas are not always manipulated in $\phi$-normal form, but generic variables can occur in an explicit generic context. The elimination of toplevel generic quantifications and contexts is triggered by the tactic `abstract`, usually before applying the induction rule.

It currently provides little automation, incarnated by the tactic `prove` which performs a focused proof-search. It treats fixed points as described in [3] but is still not able to infer invariants, and is thus limited to unfoldings. Ongoing work in that direction promises much more automation in these simple developments. In particular, instances of all properties proved in this paper would actually be established automatically by simple heuristics: in our proofs, when we build a derivation using the induction rule, the invariant can be infered from the conclusion. Bureaucratic lemmas such as strengthening of generic quantification, but also more subtle things, detailed in the next example, would not be a burden for the user anymore: the cost of having a minimal generic quantification would vanish.

[2] However, the code shown in this section is often not valid for Taci, but has been simplified for readability.

## *4.1 The `copy` program*

In that example we shall work on a representation of untyped $\lambda$-terms. We assume a signature with a type $tm$ and two constants: $app$ of type $tm \to tm \to tm$ and $abs$ of type $(tm \to tm) \to tm$. Notice that since these are term-level constants and not fixed point definitions, there is nothing wrong with negative occurence of $tm$ in the type of $abs$. The `copy` program is defined as follows in $\lambda$Prolog:

```
copy (app M N) (app P Q) := copy M P, copy N Q.
copy (abs M) (abs P) :=
    pi x\ pi y\ copy x y -> copy (M x) (N y).
```

It can be used for example to substitute a term into another: `copy A B -> copy M N` requires that `N` is `M` where some (but not necessarily all) occurences of `A` are changed into `B`.

There are mainly two approaches for encoding such a program in our logic. The first one is essentially the two-level approach [8] taken for example in the Abella system [4]. It consists in specifying $\lambda$Prolog proof-search as an object logic, and reasoning about its behaviour on the `copy` program. Given the initial formulation of the problem, this is best from an adequacy point of view, but it is heavy and notably does not allow direct inductions on the structure of `copy`. Instead, we encode the program directly. We argue that this preserves most essential points, and is adequate.

The encoding of the universal quantification in the abstraction clause is a $\nabla$ quantification, reflecting the introduction of a generic variable. In order to encode in a monotonic way the implication in that `abs` clause, we introduce a context parameter representing the `copy` atoms present in the $\lambda$Prolog context. In the encoding, a new clause appears, stating that if `copy M N` is found in the context, then it holds.

$$\begin{aligned}
copy := \mu\ copy.\ \lambda \Gamma M N.\ & \\
& \langle M, N\rangle \in \Gamma \\
\vee\ & (\exists M_1 M_2 N_1 N_2.\ M = (app\ M_1\ M_2) \wedge N = (app\ N_1\ N_2) \wedge \\
& \quad copy\ \Gamma\ M_1\ N_1 \wedge copy\ \Gamma\ M_2\ N_2) \\
\vee\ & (\exists M_1 N_1.\ M = (abs\ M_1) \wedge N = (abs\ N_1) \wedge \\
& \quad \nabla xy.\ copy\ (\langle x, y\rangle :: \Gamma)\ (M_1 x)\ (N_1 y))
\end{aligned}$$

We shall prove an useful fact about that inductive definition:

$$\forall MN.\ copy\ []\ M\ N \supset M = N$$

This property should be proved by induction over *copy*. A naive invariant would be that the context contains only pairs $(m, m)$. It does not hold: when going under an abstraction, two generic variables are introduced, marked equal in the context $\Gamma$. Since they are generic, it does not break the result, but does make the invariant more complex.

In fact, we proceed by proving that *copy* implies *eq*, where *eq* is defined as a least fixed point mostly like *copy* except for its abstraction clause:

$$eq\ (abs\ M)\ (abs\ N) := \nabla x.\ eq\ (\langle x, x\rangle :: \Gamma)\ (M\ x)\ (N\ x)$$

For *eq*, it is now easy to show that if the context contains only pairs $(m, m)$, it will remain true and hence *eq* implies equality. It remains to show that *copy* implies *eq*, which actually holds for any $\Gamma$. This is done by induction over *copy*, the interesting case being that of abstraction:

$$\begin{array}{rl} \forall \Gamma M N. & (\nabla ab.\ eq\ ((a,b) :: \Gamma)\ (M\ a)\ (N\ b)) \\ \supset & (\nabla a\ .\ eq\ ((a,a) :: \Gamma)\ (M\ a)\ (N\ a)) \end{array}$$

This goal really expresses the heart of our problem with the shape of the context. It requires an induction under two generic quantifications. Basically, the invariant should state that two generic variables can be merged. After having lifted *eq* over the generic quantifications on $a$ and $b$, this can actually be written elegantly as a simple invariant, and the proof of invariance is straightforward:

$$\lambda \Gamma'' M'' N''.\ \nabla a.\ eq\ (\Gamma''\ a\ a)\ (M''\ a\ a)\ (N''\ a\ a)$$

We do not know of any other system where it is possible to obtain the name merging principle from the induction rule in such a direct way. The proofs cited above can however be carried out in other logics, for example in LG [19] as pointed out by Gacek, thanks to the strengthening on names. Since strengthening and exchange are also admissible in $\mu\mathrm{LJ}^{\nabla}$ for *copy* and *eq*, that gives alternate, less direct proofs in our system.

## *4.2 λ-calculus*

We now discuss the specification of simply typed λ-calculus à la Church, and the proofs of subject reduction and determinacy of typing. The signature for terms will consist of two types: *ty* for simple types and *tm* for λ-terms; two constants for terms: $app :: tm \to tm \to tm$ and $lambda :: ty \to (tm \to tm) \to tm$; the constant $arrow :: ty \to ty \to ty$ for types, as well as some arbitrary base types.

We shall not detail the definition of a predicate `bind` such that when $\Gamma$ is a list of pairs representing bindings, `bind` $\Gamma\ k\ v$ expresses that $\langle k, v\rangle \in \Gamma$. We define a least fixed point `typeof` such that the typing judgement ($\Gamma \vdash^{\Lambda^{\to}} m : t$) is represented by (`typeof` $\Gamma\ m\ t$):

```
inductive typeof G M T :=
  (bind G M T) ;
  (sigma t\m1\m2\
     M = (app m1 m2), typeof G m1 (arrow t T), typeof G m2 a) ;
  (sigma t\t'\f\
     M = lambda t f, T = arrow t t',
     nabla x\ typeof (cons (pair x t) G) (f x) t').
```

Along these lines, we also specify one-step β-reduction as a least fixed point called `one`, and prove subject reduction as well as determinacy of typing. However, the statement of the theorems required particular care. In this style of specification, a variable is nothing but a placeholder for a term. Note for example that in the typing relation, nothing forbids the typing context Γ to contain constructed terms, instead of only variables as usual. This seems interesting, but certainly differs from the informal practice. One can try to stick to the usual notion of context; for example we established subject reduction under the assumption that the context does not contain constructed terms:

```
theorem subject_reduction :
  pi m\n\ one m n =>
   pi G\
    (pi t\a\t'\ bind G (lambda t a) t' => false) =>
    (pi a\b\t'\ bind G (app a b) t' => false) =>
    pi t\ typeof G m t => typeof G n t.
```

For typed determinacy, we used an alternative definition of the notion of context. Instead of assuming that keys are unique and not constructed, we assumed that each binding satisfied type determinacy:

```
context G := pi x\t\ bind G x t => pi t'\ typeof G x t' => t=t'.
theorem type_determinacy :
  pi g\x\t\ typeof g x t => context g => pi t'\ typeof g x t' => t=t'.
```

This formulation implies new branches compared to the usual proof. However they are trivially treated, and overall the proof is not more complex than with more traditional notions of contexts. Moreover, the resulting theorem is also slightly stronger than the usual one, as it allows richer contexts.

We also explored a *named* style of specification, which involves the addition of a type $n$ for free variables and a constructor $var :: n \rightarrow tm$. The typing relation can then be written as follows:

```
inductive typeof G M T :=
  (sigma v\ M = var v, bind G v T) ;
  (sigma t\m1\m2\
     M = app m1 m2, typeof G m1 (arrow t T), typeof G m2 t) ;
  (sigma t\t'\f\
     M = lambda t f, T = arrow t t',
     nabla x\ typeof (cons (pair x t) G) (f (var x)) t').
```

With that specification, typing contexts now contain only variables, which are objects of a different kind than terms. This lead to a significantly shorter proof of type determinacy, requiring less bureaucracy.

With both styles of specification, the $\mu\mathrm{LJ}^{\nabla}$ proof of subject reduction could not be completly built within Taci, because our tool currently only strictly supports higher-order patterns. With the first specification, only a small gap was left because of that. However, the named style of specification was very inconvenient to work with, producing much more unifications outside of the fragment, for example `(x\var x) = (x\m (var x))`. In any case, this is only a problem of the implementation, not of the logic.

## 5 Conclusion

Thanks to a reformulation of the $\nabla$ quantifier as a defined connective, we have revealed the expressivity of minimal generic quantification in presence of fixed points. This resulted in the logic $\mu\mathrm{LJ}^{\nabla}$, which is a good candidate for adequately representing objects, and reasoning in an expressive way about them.

We have notably exhibited classes of formulas for which generic weakening and strengthening could be derived. It should be possible to build other classes corresponding to other common logical principles, such as $(\nabla x.\ Px) \supset (\forall x.\ Px)$, or maybe even more exotic ones like $(\nabla xy.\ Pxy) \supset (\nabla x.\ Pxx)$. More importantly, these results should be integrated in mechanized theorem provers to simplify their use. In fact, we claim that simple inductive theorem proving heuristics should be able to achieve this goal, since the derivations do not involve clever invariants. Regarding automated theorem proving, an other strength of $\mu\mathrm{LJ}^{\nabla}$ is that its rules are standard ones, the only novelty being in the quotienting of the formulas modulo the lifting $\phi$. Thus, results and heuristics for $\mu\mathrm{LJ}^{=}$ should immediately extend to $\mu\mathrm{LJ}^{\nabla}$.

From a theoretical point of view, we expect that the essential ideas behind the design of $\mu\mathrm{LJ}^{\nabla}$ could be put to work in other settings if needed. For example, adding structural rules to $\mu\mathrm{LJ}^{\nabla_0}$, *i.e.,* working in LG, brings enough expressivity for many non-trivial examples, but it is still unknown if that approach is strictly more expressive. It might not be, as some trivial theorems of $\mu\mathrm{LJ}^{\nabla}$ do not seem to be provable in LG, such as $\forall l x'.\ (\nabla n.\ \mathit{mem}\ (x'\ n)\ l) \supset \exists x.\ x' = (\lambda n.\ x)$.

Our system seems closely related in spirit to [13] which also manages contexts in a strict way. But it is difficult to clearly relate the two approaches, since ours is based on first-order logic while Pientka's is purely computational but higher-order. Extending our system to higher-order logic would be challenging but interesting, and should allow for a better comparison with systems based on the proof-as-program viewpoint.

## Acknowledgement

I am grateful to Andrew Gacek, Dale Miller, Gopalan Nadathur and Alwen Tiu for many insightful discussions on this topic.

## References

[1] David Baelde. On the expressivity of minimal generic quantification: Extended version. Technical report, 2008. Available from http://hal.inria.fr/inria-00284186.

[2] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. The Bedwyr system for model checking over syntactic expressions. In Frank Pfenning, editor, *21th Conference on Automated Deduction (CADE)*, number 4603 in LNAI, pages 391–397. Springer, 2007.

[3] David Baelde and Dale Miller. Least and greatest fixed points in linear logic. In N. Dershowitz and A. Voronkov, editors, *International Conference on Logic for Programming and Automated Reasoning (LPAR)*, volume 4790 of *LNCS*, pages 92–106, 2007.

[4] Andrew Gacek. The Abella interactive theorem prover (system description). Available from http://arxiv.org/abs/0803.2305. To appear in IJCAR'08, 2008.

[5] Daniel R. Licata, Noam Zeilberger, and Robert Harper. Focusing on binding and computation. Technical Report CMU-CS-08-101, Department of Computer Science, Carnegie Mellon University, 2008.

[6] Dale Miller. Abstract syntax for variable binders: An overview. In John Lloyd and et. al., editors, *Computational Logic - CL 2000*, number 1861 in LNAI, pages 239–253. Springer, 2000.

[7] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In Glynn Winskel, editor, *12th Symp. on Logic in Computer Science*, pages 434–445, Warsaw, Poland, July 1997. IEEE Computer Society Press.

[8] Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.

[9] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[10] Dale Miller and Alwen Tiu. A proof theory for generic judgments: An extended abstract. In *18th Symp. on Logic in Computer Science*, pages 118–127. IEEE, June 2003.

[11] Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, October 2005.

[12] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

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

[14] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th Conference on Automated Deduction (CADE)*, number 1632 in LNAI, pages 202–206, Trento, 1999. Springer.

[15] Zach Snow, David Baelde, and Dale Miller. Taci: an interactive theorem proving framework. 2007.

[16] Peter Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 222–232. IEEE Computer Society Press, IEEE, June 1993.

[17] Alwen Tiu. *A Logical Framework for Reasoning about Logical Specifications.* PhD thesis, Pennsylvania State University, May 2004.

[18] Alwen Tiu. Model checking for $\pi$-calculus using proof search. In Martín Abadi and Luca de Alfaro, editors, *CONCUR*, volume 3653 of *LNCS*, pages 36–50. Springer, 2005.

[19] Alwen Tiu. A logic for reasoning about generic judgments. In A. Momigliano and B. Pientka, editors, *Int. Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'06)*, 2006.