A Two-Level Logic Perspective on (Simultaneous) Substitutions

Kaustuv Chaudhuri
Inria and LIX/École polytechnique
Palaiseau, France
kaustuv.chaudhuri@inria.fr

Abstract

Lambda-tree syntax (λTS), also known as *higher-order abstract syntax* (HOAS), is a representational technique where the pure λ-calculus in a meta language is used to represent binding constructs in an object language. A key feature of λTS is that capture-avoiding substitution in the object language is represented by β-reduction in the meta language. However, to reason about the meta-theory of (simultaneous) substitutions, it may seem that λTS gets in the way: not only does iterated β-reduction not capture simultaneity, but also β-redexes are not first-class constructs.

This paper proposes a representation of (simultaneous) substitutions in the *two-level logic approach* (2LLA), where properties of a specification language are established in a strong reasoning meta-logic that supports inductive reasoning. A substitution, which is a partial map from variables to terms, is represented in a form similar to typing contexts, which are partial maps from variables to types; both are first-class in 2LLA. The standard typing rules for substitutions are then just a kind of context relation that are already well-known in 2LLA. This representation neither changes the reasoning kernel, nor requires any modification of existing type systems, and does not sacrifice any expressivity.

CCS Concepts • **Theory of computation → Logic and verification**; *Constraint and logic programming*;

Keywords Meta-theory; substitution; two-level logic approach; λ-calculus; λProlog; Abella

ACM Reference Format:
Kaustuv Chaudhuri. 2018. A Two-Level Logic Perspective on (Simultaneous) Substitutions. In *Proceedings of 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3167093

1 Introduction

Meta-theoretic reasoning about higher-order object languages with binding constructs such as λ-abstraction or fixed-point expressions requires reasoning about subterms that occur within the scopes of such constructs. It is immediately apparent that any sensible notion of such subterms must not depend on the *names* of bound variables, which are allowed to α-vary without affecting the meaning of a term. Such a notion of subterm must also be compatible with the inductive definition of the structure of terms in the sense that one must be able to establish recursive properties of such terms. For instance, one must be able to define a recursive unary predicate that yields *true* if and only if its argument denotes a closed term, i.e., a term without free variables. A subtler but genre-defining desideratum is that the representation of terms must support *capture-avoiding substitution* of terms for free variables. This combination of requirements formed the basis of the POPLMark benchmark [1]. Section 5 gives an overview of various responses to this challenge in existing formal reasoning systems.

In this paper we are interested in a representational technique called *λ-tree syntax* (λTS) [14], sometimes also referred to as *higher-order abstract syntax* (HOAS) [25],[1] based on a simple idea that goes back to Church [9]: use *the* simply typed λ-calculus together with the equational theory determined by $\alpha\beta\eta$-equivalence (i.e., λ-equivalence) directly as the representation language for object terms. Concepts of the object language are treated as typed (non-logical) constants of the λ-calculus, and a closed family of such constants gives an inductive characterization of the structure of object terms. The λ-term structure is used to capture the binding structure of the object language, so reasoning up to α-equivalence of object terms is *for free*. If the constants are well-chosen, then normal λ-terms are isomorphic to object terms (*representational adequacy*) [24]. Furthermore, an object term with one free variable is represented by a λ-term with one λ-abstraction,[2] and hence the substitution of a term for that free variable is simply represented as the *application* of this λ-abstracted term to the representation of the substituted

[1]The term HOAS has been abused in the literature to cover both representational techniques that fail to live up to the name and also to programming techniques in implementations of higher-order object languages. We use λTS primarily to avoid confusion.
[2]Always up to λ-equivalence.

term; in other words, it is represented by a β-redex. A formal reasoning system that provides the λ-calculus as a representational language solves the problem of variable representation over a large class of object languages *uniformly* and *universally* [15].

Does λTS then provide *all* the desiderata for representation in meta-theory? The answer is not obvious. One place to look for potential missing features is in certain kinds of meta-theory where it is important not only to *perform* a given substitution but also to *reason about all* substitutions. Often, in formulating logical relations, (bi)simulations, reducibility arguments, and the like, a property that is defined on closed terms is *lifted* to open terms—terms with free variables—by means of quantifying over all grounding substitutions. For example, *applicative similarity* $\preceq$ on closed simply typed terms is a binary relation determined by the structure of the *types* of the terms. This is lifted to *open similarity*, $\preceq^{\mathrm{o}}$, by defining $s \preceq^{\mathrm{o}} t$ as: *for all grounding substitutions* ρ, it must be the case that $[\rho]s \preceq [\rho]t$. Since λTS represents substitutions as β-redexes, to reason generically about all substitutions would intuitively amount to quantifying over β-redexes.

Unfortunately, this intuition does not stand up to scrutiny. A β-redex is not an independent and atomic concept in the λ-calculus. If we attempt to make it one, we lose λ-conversion as an equational theory and hence lose the simplicity and the strength of λTS. Worse, it is unclear how to selectively quantify over only *grounding* substitutions, since a β-redex cannot easily encode closedness. Furthermore, we may want to establish inductive properties defined by recursion on the substitutions themselves; for instance, we may want open similarity to be closed under substitution: whenever $s \preceq^{\mathrm{o}} t$, so is $[\sigma]s \preceq^{\mathrm{o}} [\sigma]t$ for any substitution—not necessarily grounding—σ. The proof of this closure property would require reasoning about compositions of substitutions. It is unnatural to formulate such a theorem in terms of substitutions as β-redexes.

One way to solve this problem would be to make substitutions as first-class as λ-terms. A comprehensive approach that follows this path to its end is the Beluga proof system [27], which uses disjoint sorts of terms, substitutions, and typing contexts, and builds in the relations that hold between these different sorts. This is achieved in the framework of contextual modal type theory [23], an extension of ordinary type theory with a notion of *contextual type*, which is a type that is well-formed with respect to a *local context*. Terms of a contextual type are allowed to depend on the variables that are present in the local context of the type. A first-class simultaneous substitution is then seen as a finite map between local contexts, defined by induction on its structure [26]. The notion of applying a substitution to a term or to another substitution then becomes universal and uniform. One is tempted therefore to see the Beluga design as a natural completion of the λTS approach by adding the apparently missing features.

In this paper I will describe a different perspective that reuses concepts that already exist in a slightly different form instead of adding new concepts. It is easiest to illustrate it with an analogy to type-checking. A type system for a higher-order object language must prescribe a mechanism for assigning a type to every subterm of a given object term; in particular, it must describe what happens to subterms of a binding construct. Take the simply typed λ-calculus itself as an object language. The typing rule for λ-abstractions says: to assign the type $A \to B$ to the term $\lambda x.\, t$, it is sufficient to assign the type B to t *under the assumption* that x is assigned type A. This is usually written as an inference rule over contextual (or hypothetical) judgments:

$$\frac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B}$$

Here, the *typing context* Γ is a finite map from a collection of (distinct) variables to their types, and the notation $\Gamma, x{:}A$ implicitly requires that x is not already assigned a type in Γ. In the derivation tree for $\Gamma, x{:}A \vdash t : B$, whenever x is encountered in t, its corresponding type assignment is reconciled with its assumed type A; for instance if t is x, then $A = B$. Any formal reasoning system that is equipped to deal with the meta-theory of type-checking must be able to represent such typing contexts and reason about their properties.

I argue that *nothing more is needed to represent and reason about (simultaneous) substitutions.* Concretely:

- substitutions are a kind of context that map variables to terms in a substitution judgment, just as typing contexts map variables to types in a typing judgment;
- the action of applying a substitution is analogous to the action of type-checking a term; and
- the relation of a substitution to a pair of typing contexts is an inductively defined relation, analogous to the inductive definition of a typing context in isolation.

The principal contribution of this paper is the view embodied in the second and third bullets above. Seeing a substitution as a context is not particularly novel: it is just a restatement of the well-known concept of a *closure* – or, more accurately, the environment part of a closure. On the other hand, the application of a substitution to a term by induction on the structure of the term has a surprisingly simple definition that is almost never used: it is an identical *copy* of the original (unsubstituted) term, except the free variables in the domain of the substitution are replaced by their mappings. In other words, applying the substitution σ to the term s to yield t is akin to deriving the judgment $\sigma \vdash s \longmapsto t$ where $\longmapsto$ encodes a recursive transformation of s to t, defined by induction on the structure of s, that is an isomorphism except for the free variables of s. This is exactly analogous to the : relation for typing, which can also be seen as a relation defined by structural induction on its term argument.

$$
\begin{array}{rlr}
A, B, \ldots & ::= \mathsf{nat} \mid A \to B & \text{(types)} \\
v & ::= 0 \mid \mathsf{s}(t) \mid \lambda x.\, t & \text{(values)} \\
s, t, \ldots & ::= v \mid x \mid s\ t \mid \mathsf{fix}\, x.\, t & \text{(terms)} \\
& \quad \mid \mathsf{case}\ s\ \mathsf{of}\ \{0 \Rightarrow t_0, \mathsf{s}(x) \Rightarrow t_s\} &
\end{array}
$$

$$
\frac{}{\Gamma, x{:}A \vdash x : A} \qquad \frac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \frac{\Gamma, x{:}A \vdash t : A}{\Gamma \vdash \mathsf{fix}\, x.\, t : A}
$$

$$
\frac{\Gamma \vdash s : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\ t : B} \qquad \frac{}{\Gamma \vdash 0 : \mathsf{nat}} \qquad \frac{\Gamma \vdash t : \mathsf{nat}}{\Gamma \vdash \mathsf{s}(t) : \mathsf{nat}}
$$

$$
\frac{\Gamma \vdash s : \mathsf{nat} \qquad \Gamma \vdash t_0 : A \qquad \Gamma, x{:}\mathsf{nat} \vdash t_s : A}{\Gamma \vdash \mathsf{case}\ s\ \mathsf{of}\ \{0 \Rightarrow t_0, \mathsf{s}(x) \Rightarrow t_s\} : A}
$$

$$
\frac{}{v \Downarrow v} \qquad \frac{s \Downarrow \lambda x.\, u \qquad [t/x]u \Downarrow v}{s\ t \Downarrow v} \qquad \frac{[\mathsf{fix}\, x.\, t/x]t \Downarrow v}{\mathsf{fix}\, x.\, t \Downarrow v}
$$

$$
\frac{s \Downarrow 0 \qquad t_0 \Downarrow v}{\mathsf{case}\ s\ \mathsf{of}\ \{0 \Rightarrow t_0, \mathsf{s}(x) \Rightarrow t_s\} \Downarrow v}
$$

$$
\frac{s \Downarrow \mathsf{s}(u) \qquad [u/x]t_s \Downarrow v}{\mathsf{case}\ s\ \mathsf{of}\ \{0 \Rightarrow t_0, \mathsf{s}(x) \Rightarrow t_s\} \Downarrow v}
$$

Figure 1. Simply typed PCF with natural numbers

This re-imagining of substitutions as a kind of context will be depicted in the Abella system in this paper. Abella is an instantiation of the *two-level logic approach* (2LLA) to reasoning about relational specifications [14, 19, 36], and is used here because the notion of an inductively defined typing context is already well-established, as is the mechanism of transforming a rule-based specification of inductively defined relations into a corresponding inductive definition. To illustrate the proposed view of substitutions, we provide a proof of the substitutivity of the Howe relation (see [29] for a comprehensive treatment) for a simply typed variant of PCF with natural numbers and monomorphic lists. This illustration is part of a larger development that formalizes the fact that applicative similarity for PCF is a (pre-)congruence. Showing substitutivity has been claimed to be a challenge to formalize in Abella in the past because of the difficulty of quantifying over substitutions [21]. The full formalization may be browsed on-line;[3] and this example is also distributed in the `examples/programming-languages/howe` sub-directory of the `master` branch of Abella, available from Github.[4]

2 A Whistle-Stop Tour of Abella

The main contribution of this paper could be achieved in any implementation of the two-level logic approach (2LLA), but I will describe it in terms of the Abella implementation. A comprehensive introduction to Abella can be found in the tutorial [2]. This section will briefly touch upon the aspects of Abella most relevant to this paper, with a simply typed variant of PCF with natural numbers as our object of study.

[3] http://abella-prover.org/examples/programming-languages/howe/
[4] https://github.com/abella-prover/abella

We will use the grammar of *types* ($A, B, \ldots$), *values* (v), and *terms* ($s, t, \ldots$) depicted in Figure 1.

Representing syntax The syntax of terms and types of PCF will be represented in Abella by means of the two basic types `tm` and `ty` respectively, introduced with the `kind` keyword. These basic types are declared in a *type signature*, together with constants for each constructor for PCF terms and types, which are introduced with the `type` keyword.

```
kind ty  type.
type nat ty.
type arr ty -> ty -> ty.
```

With this signature of types, the type $\mathsf{nat} \to (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat}$ would be represented by the term `arr nat (arr (arr nat nat) nat))`.

```
kind tm   type.
type zero tm.
type succ tm -> tm.
type abs  (tm -> tm) -> tm.
type fix  (tm -> tm) -> tm.
type app  tm -> tm -> tm.
type case tm -> tm -> (tm -> tm) -> tm.
```

Observe that there are no constructors for variables. Following standard λTS practice, abstracted variables are represented by λ-abstractions in the representation language, for which the concrete syntax in Abella is `x\ M`, which binds `x` in the term `M`, and the infix operator `\` representing abstraction has the lowest precedence. Thus, the PCF term

$$\mathsf{fix}\, f.\, \lambda x.\, \mathsf{case}\ x\ \mathsf{of}\ \{0 \Rightarrow 0, \mathsf{s}(n) \Rightarrow \mathsf{s}(\mathsf{s}(f\ n)\}$$

would be represented by the concrete term

```
fix f\ abs x\
  case x zero (n\ succ (succ (app f n))).
```

Observe that the bound variables f, x, and n in the PCF term are represented by the abstracted concrete variables `f`, `x`, and `n` respectively. However, the names do not matter; indeed, the above concrete term is considered *equal* to the following.

```
fix g\ abs y\
  case y zero (u\ succ (succ (app g u)))
```

First-order judgments Consider determining if a given PCF term is a value. In Abella, this would be represented in terms of a unary *predicate* called `value` that would be *derivable* if indeed its argument represents a value. A predicate is a constant of target type `o`, which is reserved for predicates. The predicate is defined in the form of *rules*, which are written in standard λProlog notation; in this particular case they are all *unit clauses* consisting of only a *head*.

```
type value  tm -> o.
value zero.
value (succ T).
value (abs R).
```

A capitalized variable is always assumed to be universally quantified over the clause, so the second clause declares that for any term `T`, the predicate `value (succ T)` is derivable. The third clause for abstractions could have been written alternatively as:

```
value (abs x\ T x).
```

but we write it with `x\ T x` in an η-contracted form. Note that it would not be correct to write the following:

```
value (abs x\ T).
```

This clause would declare that `T` has no *dependency* on `x`, because `T` is quantified outside the scope of the `x`.

A more complex judgment is the evaluation judgment $\Downarrow$ that is defined on closed terms. The rules defining it sometimes have premises for a given conclusion. In Abella such rules are represented as logic programming clauses in λProlog notation consisting of the head, representing the conclusion, yielding the premises, which is a comma-separated collection of predicates (or formulas more generally) called the *body*. The head and body are separated by `:-`.

```
type eval tm -> tm -> o.
eval V V :- value V.
eval (app S T) V :-
  eval S (abs R), eval (R T) V.
eval (fix R) V :- eval (R (fix R)) V.
eval (case S Tz Ts) V :-
  eval S zero, eval Tz V.
eval (case S Tz Ts) V :-
  eval S (succ N), eval (Ts N) V.
```

For instance, if we define D to be the term

$$\mathrm{fix}\, D.\, \lambda x.\, \mathrm{case}\; x \;\mathrm{of}\; \{0 \Rightarrow 0, \mathrm{s}(y) \Rightarrow \mathrm{s}(\mathrm{s}(D\; y))\}$$

represented by `D` which is

```
  fix D\ abs x\
    case x zero (y\ succ (succ (app D y)))
```

then it will be the case that $D\;(\mathrm{s}(\mathrm{s}(0))) \Downarrow \mathrm{s}(\mathrm{s}(D\;(\mathrm{s}(0))))$ and also `eval (app D (succ (succ zero))) (succ (succ (D (succ zero))))` will be derivable.

Higher-order judgments Things get only a bit more complex when representing higher-order judgments such as the typing judgment (which is second-order). The rules are as follows.

```
type of  tm -> ty -> o.
of zero nat.
of (succ T) nat :- of T nat.
of (abs R) (arr A B) :-
  pi x\ (of x A => of (R x) B).
of (fix R) A :-
  pi x\ (of x A => of (R x) A).
of (case S Tz Ts) A :-
  of S nat, of Tz A,
  pi x\ (of x nat => of (Ts x) A).
```

Since we do not have a constructor for variables, we similarly do not have a typing rule for variables. Instead, whenever a variable is introduced into the context in any of the other rules, its type is known; thus, the program clause that represents it not only invents a *fresh* variable (using the `pi` keyword) but also assumes a (scoped) program clause about the type of that variable using the `=>` keyword.

For example, the clause for `abs` can be understood as follows: for any fresh `x`, derive `of (R x) B` by adding to the program the clause `of x A`. Of course this new clause `of x A` is only available while deriving the specific goal `of (R x) B` (and its subgoals); no other goal gets to see that clause. The body of the abstraction, `R`, is then allowed to use that new variable `x`, indicated with the application `R x`. It does not mean that `x` definitely occurs in `R x`, but that it is *allowed to*: if it is encountered in `R x`, then the assumed program clause `of x A` determines its type.

Reasoning about rule-based specifications The kinds of program clauses we have seen so far allow us to define a system of rules, but the system is always extensible – a new clause can always be added. For a system of rules that is completely specified, we need to view the collection of rules as an immutable, inductive specification; this will let us not only say when a predicate holds, but also when it doesn't, and in general will allow us to relate two different predicates. This forms the core of the *two-level logic approach* (2LLA) [14], where a *reasoning* meta-logic with support for induction (and, in general, also co-induction) is used to reason about λProlog specifications (or some similar *specification logic*) derivations. In Abella, this reasoning logic embeds derivability of predicates by means of a notation: `{L |- G}`, where `L` is a list of predicates (denoted by the type `olist` and with constructors `nil` and `::`) and `G` is a predicate of type `o`, to mean that the predicate `G` is derivable in λProlog from a given fixed and global collection of program clauses and the additional local scoped assumptions `L`. When the list `L` is empty, both it and the `|-` can be omitted; so `{G}` says that `G` is derivable from the global program clauses alone.

At the reasoning level, properties of the specified system are proved by induction over λProlog derivations. An example of such a property is subject-reduction, which can be written concisely in the concrete syntax of the reasoning logic as follows:

```
Theorem subjred : forall S V A,
   {of S A} -> {eval S V} -> {of V A}.
```

In Abella this would be proved by `induction on` one of the predicates, for instance the first one, `{of S A}`. Abella will take that instruction to mean that a new *inductive hypothesis* is generated of the form:

```
IH : forall S V A,
  {of S A}* -> {eval S V} -> {of V A}
```

Note the * annotation on the first assumption; it states that the IH can only be applied if we can find a derivation of {of S A} that is strictly smaller than that of the original assumption we were inducting on. To a decent approximation, this can be seen as a size annotation familiar from sized types, although the full technical basis is given in terms of the induction rule of the $\mathcal{G}$ logic that Abella's reasoning logic is based on [12, 13].

The mother of all substitution theorems The reasoning perspective on specification derivations relies crucially on the |- notation being interpreted as the sequent arrow for intuitionistic logic, which in turn means that meta-theoretic properties of intuitionistic logic can both be proved about the { |- } notation and be used as lemmas. As an illustration, a common kind of theorem we may want to prove about typing is the following substitution property:

```
forall G M N A B,
  (forall x, {G, of x A |- of (M x) B}) ->
  {G |- of N A} ->
  {G |- of (M N) B}.
```

We are using application at the meta-level to denote a substitution: the type of M is tm -> tm, and M N therefore stands for the replacement of the free variable in M with N. This is an example of a situation where a substitution is merely *performed* rather than analyzed. This theorem can potentially be proved by induction on one of the arguments, but it turns out that *all* theorems that are of the form {L, F |- G} -> {L |- F} -> {L |- G} are provable! This is a direct consequence of the admissibility of cut. Indeed, *all* substitution theorems about adequate specifications of rule-based systems boil down to either this kind of cut theorem, or a minor variation that deals with instantiation of free variables. These theorems were proved once and for all when the { } notation was defined, and can be freely used as a shortcut. Much more detail can be found in the tutorial [2] or in [13, 14].

Inductive and co-inductive definitions The reasoning logic is stronger than the specification logic in the sense that it can *define* atomic formulas in terms of fixed points with a least or greatest fixed point interpretation. Here, for instance, is a definition of an inductive predicate that holds for all natural numbers and another for all even numbers.

```
Define isnat : tm -> prop by
  isnat zero ;
  isnat (succ N) := isnat N.

Define iseven : tm -> prop by
  iseven zero ;
  iseven (succ (succ N)) := iseven N.
```

Note that we use the type prop for reasoning logic formulas. Like λProlog programs, definitions are given in the form of clauses (*definitional clauses* in this case), but unlike λProlog programs such a definition comes with an induction or co-induction principle. The induction principle would be used to prove a theorem such as:

```
forall M, iseven M -> isnat M.
```

Such a theorem cannot be proved in λProlog because a given λProlog derivation can only explore a finite structure, but the theorem holds for unboundedly large structures.

As a consequence of the induction principle, inductive definitions are much more restricted: they must be *stratified*, which is to say that recursive occurrences of the defined atom must only occur positively in the same definition. Thus, for instance, the of predicate could not be seen as an inductive definition because of the negative occurrences of of in the higher-order clauses. On the other hand, an inductive definition is a complete characterization of a relation, and it serves just as well to establish when a property holds as when it does not. So, for instance, to prove that there is no value V such that eval (fix x\ x) V holds would require us to appeal to an induction principle for eval. (Seen as a logic program, the query eval (fix x\ x) V would never terminate!) The { } notation is just a standard inductive definition that uses the same induction principle as any other inductive definition. Therefore, in the reasoning logic we are able to reason about when a λProlog program may not exist, i.e., we can *prove* forall V, {eval (fix x\ x) V} -> false by induction.

Generic reasoning A big difference between the inductive reasoning logic and the open-ended specification logic is the treatment of quantifiers. The specification logic has only a single kind of quantifier, pi, whose interpretation as goal (i.e., to the right of the |-) is always the same regardless of whether the { } formula is a hypothesis or a conclusion. In the reasoning logic, on the other hand, we must distinguish between a quantifier that stands for all its possible instances (the *extensional* reading) and a quantifier that stands for any generic instance (the *intensional* reading). The former quantifier, written $\forall$ (concretely forall), is used to enumerate solutions, which is essential for proofs by induction and case analysis. The latter is used, as expected, to represent the quantifier of the specification logic, and is depicted using ∇ (concretely, nabla) instead of pi, which is a reserved keyword of λProlog. The definition of { } is such that {L |- pi x\ G x} is equivalent to ∇x.{L |- G x}. In terms of the proof-theory of $\mathcal{G}$, the difference between the quantifiers amounts to the guarantee of *freshness* that ∇-bound variables are required to have to guarantee genericity, which in turn implements the semantics of the pi operator. The formula $\forall x.\, \forall y.\, x \neq y$, for instance, is not true (in a non-empty domain) because x and y can be instantiated to the same term, whereas $\nabla x.\, \nabla y.\, x \neq y$ is true because whatever generic thing x may be, y is another generic thing that cannot contain x as a subterm.

Inductive definitions of typing contexts The combination of inductive definitions and generic reasoning is strong enough to *characterize* all typing contexts. Consider, for instance, the local context of assumptions about the types of variables that result from trying to derive `of T A` in the reasoning logic. The derivation would proceed by analysis of the structure of `T`; at every binder, it would add a scoped `of` assumption about a new variable. Thus, the derivation of `of T A` would eventually construct a subgoal of the form

```
∇x1,...,xn.{of x1 A1, ..., of xn An |- of T' A'}
```

which in turn guarantees that each of the `xi` are distinct and therefore has a unique type assigned to it in the local context. Indeed, the general form of such a local *typing context* can be defined inductively in Abella as follows:

```
Define tyctx : olist -> prop by
  tyctx nil ;
  nabla x, tyctx (of x A :: G) := tyctx G.
```

The first clause states that the empty context is a valid typing context, while the second states that `G, of x A` is a typing context (written as a list using the `::` constructor) if and only if `G` is a typing context, and moreover `x` is a generic element that is fresh for (i.e., does not occur in) both `G` and `A`.

The benefit of such a definition is that we can precisely characterize exactly those sequents that represent hypothetical judgments that occur in any derivation of the typing judgment according to the rules of Figure 1. If `L`, `T`, and `A` represent Γ, t, and A, respectively, then the judgment $\Gamma \vdash t : A$ is denoted canonically by the conjunction:

```
tyctx G /\ {G |- of T A}.
```

By representational adequacy, any theorem that holds about derivable judgments $\Gamma \vdash t : A$ would then also hold for `tyctx G /\ {G |- of T A}` and vice versa.

Nominal constants and open terms Thus far we have dealt with closed terms or terms in their typing contexts, but what about open terms? To answer this question, we will need to understand what a hypothesis `tyctx G` means. It says that the elements of `G` must be of the form `of x A` where each of the `x`s are distinct generic elements.

In Abella, this kind of property is captured with the help of *nominal constants*, which are an infinite family of pairwise distinct constants that are assumed to exist at every type.[5] Nominal constants behave just as ordinary constants, save for one modification to the reasoning principle of $\mathcal{G}$ that relates hypotheses and conclusions: a hypothesis H can entail a conclusion C if H can be rewritten to C by a permutation of its free nominal constants, i.e., if H and C are *equivariant*. Such nominal constants are denoted in Abella using reserved identifiers `n1`, `n2`, etc. The formulas `p n1 n2` and `p n2 n1`, for example, are equivariant, and hence `p n1 n2 -> p n2 n1` is provable. On the other hand, `p n1 n1` and `p n1 n2` are not equivariant, so one does not necessarily imply the other. Note that equivariance does not interfere with the equational theory of λ-terms– the nominal constants `n1` and `n2` are not equal (and provably distinct). We just enlarge the notion of *logical entailment* to make it up to equivariance.

Nominal constants are used to represent the free variables of open terms. They can be seen as witnesses to the ∇-quantified variables. Whenever such a quantifier occurs among the hypotheses or the conclusion, it may be replaced by an arbitrary nominal constant that is not already free in its scope. Due to equivariance, the nominal constants of one hypothesis are independent of those of other hypotheses and of the conclusion.

Putting it together We have now seen enough of the features of Abella to reconstruct simultaneous substitutions. No further concepts need to be introduced, and nothing in the Abella implementation needs to be modified. Indeed, the description of Abella in this section has stayed essentially unchanged since its very first implementation over a decade ago [11], and it is based on a proof-theoretic pedigree that is at least another decade older [33]. Excepting nominal constants, which can be seen as a simplification, the same ingredients are also available in other implementations of the 2LLA, such as Hybrid [10].

3 Simultaneous Substitutions

Mathematical definition As already mentioned, a simultaneous substitution is a finite map from variables to terms, but it is important to be precise: which variables? What terms? To get to a precise formulation, it is common to view simultaneous substitution as defining a map between terms that are well typed in two different contexts. More precisely, a substitution σ maps a term that is well typed in a *source typing context* Δ to a term that is well typed in a *target typing context* Γ. This is usually depicted as $\Gamma \vdash \sigma : \Delta$.

On the other hand, a substitution is also a finite map from variables to terms, and can be written formally as a list of pairs with the following grammar.

$$\sigma, \tau, \ldots ::= \cdot \mid \sigma, t/x \qquad \text{(substitutions)}$$

The rules relating the context map view to the finite map view can be captured as the following rule-based inductive specification.

$$\frac{}{\Gamma \vdash \cdot : \cdot} \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Gamma \vdash t : A}{\Gamma \vdash \sigma, t/x : \Delta, x{:}A}$$

Right away we can see that the following theorem must hold.

Theorem 3.1 (Weakening). *If* $\Gamma \vdash \sigma : \Delta$ *then* $\Gamma, x{:}A \vdash \sigma : \Delta$.

Proof. By induction on the derivation of $\Gamma \vdash \sigma : \Delta$. □

Given such a substitution, the effect of *applying* that substitution to a suitably typed term or substitution can itself

[5] Nominal constants are not strictly essential for the 2LLA, but they greatly simplify the conceptual load of using inductive generic definitions.

$\boxed{[\sigma]s = t}$

$$[\sigma, t/x]x = t \qquad [\sigma, t/x]y = [\sigma]y \text{ if } x \neq y$$

$$[\sigma]0 = 0 \qquad [\sigma]\mathrm{s}(t) = \mathrm{s}([\sigma]t) \qquad [\sigma]s\ t = [\sigma]s\ [\sigma]t$$

$$[\sigma]\lambda x.\, t = \lambda x.\, [\sigma, x/x]t \qquad [\sigma]\mathrm{fix}\, x.\, t = \mathrm{fix}\, x.\, [\sigma, x/x]t$$

$$[\sigma]\mathrm{case}\, s \text{ of } \{0 \Rightarrow t_z, \mathrm{s}(x) \Rightarrow t_s\} =$$
$$\mathrm{case}\, [\sigma]s \text{ of } \{0 \Rightarrow [\sigma]t_z, \mathrm{s}(x) \Rightarrow [\sigma, x/x]t_s\}$$

$\boxed{[\sigma]\tau = \varphi}$

$$[\sigma]\cdot = \cdot \qquad [\sigma]\tau, t/x = [\sigma]\tau, [\sigma]t/x$$

Figure 2. Applying substitutions

be given by induction. The rules are listed in Figure 2. Note that the rules are partial; in particular, the case of applying a substitution to a variable that is not in the domain of the substitution is not defined. This case will never arise because substitutions will only ever be applied to terms that are well typed in the source typing context.

Theorem 3.2. *The following rules are admissible.*

$$\frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash t : A}{\Gamma \vdash [\sigma]t : A} \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash \tau : \Omega}{\Gamma \vdash [\sigma]\tau : \Omega}$$

Proof. By induction on the second premise derivations. □

Theorem 3.3. *When defined,* $[\sigma]([\tau]t) = [[\sigma]\tau]t$.

Proof. Induction on the structure of t. □

First-class substitutions via copying Consider the simple case of applying a substitution to a closed term, i.e., a term that is well typed in the empty typing context. By the definition of the substitution map, it must follow that the substitution itself needs to be empty. What happens if we trace the chain of equations for $[\cdot]t$ according the rules of Figure 2? The operation simply recreates the structure of the input term t. Whenever it descends into a binding construct such as $\lambda x.\, t$, it adds an identity mapping x/x to the substitution, so the recursive structure of the substitution always remains an identity map.

This recreation of an input structure is a *copy* operation, which can be represented as a λProlog program fairly easily.

```
type cp   tm -> tm -> o.
cp zero zero.
cp (succ N) (succ NC) :- cp N NC.
cp (app M N) (app MC NC) :-
  cp M MC, cp N NC.
cp (abs R) (abs RC) :-
  pi x\ (cp x x => cp (R x) (RC x)).
cp (fix R) (fix RC) :-
  pi x\ (cp x x => cp (R x) (RC x)).
cp (case S Tz Ts) (case SC TzC TsC) :-
  cp S SC, cp Tz TzC,
  pi x\ (cp x x => cp (Ts x) (TsC x)).
```

```
Define ofsub : olist -> olist -> olist -> prop by

  ofsub G nil nil ;

  nabla x,
  ofsub (G x) (cp x (M x) :: SS x) (of x T :: D) :=
    nabla x, {G x |- of (M x) T}
             /\ ofsub (G x) (SS x) D.
.............................................................................
Define comp : olist -> olist -> olist -> prop by

  comp SS nil nil ;

  nabla x,
  comp (SS x) (cp x (M x) :: TT x) (cp x (N x) :: UU x) :=
    nabla x, {SS x |- cp (M x) (N x)}
             /\ comp (SS x) (TT x) (UU x).
```

Figure 3. Typing and composing substitutions

The cp predicate can then be interpreted (i.e., moded) as copying its input term (the first argument) to its output (second argument). Similar predicates appear often in λProlog developments ever since they were first introduced by Miller in [18]; see also [20].[6]

What happens if we apply cp to an open term, i.e., a term that has free nominal constants? Since program clauses contain no nominal constants, none of them will match. So, it will be entirely up to the local context to define what the nominal constants get copied to. This gives us a straightforward representation of *substitutions as contexts*: the substitution $t_1/x_1, \ldots, t_n/x_n$ is represented by the local context `cp nn tn :: ... :: cp n1 t1 :: nil`, where `n1`, …, `nn` are the nominal constants that occur free in the open term, and `t1`, …, `tn` are their replacements. Given a context `SS` that represents a substitution σ, the fact that $[\sigma]s = t$ is therefore captured by the derivability of the specification judgment `{SS |- cp s t}` where `s` and `t` represent s and t respectively.

The task of defining a substitution on the free nominal constants of an open term therefore reduces to defining the recursive structure of a local context of cp assumptions! To formalize this correspondence, we need to be a bit more careful about the typing contexts of s and t. In particular, we need to represent the judgment $\Gamma \vdash \sigma : \Delta$ with the rules given above as a ternary relation on contexts. The definition is in Figure 3, where `ofsub G SS D` stands for $\Gamma \vdash \sigma : \Delta$ where `G`, `SS`, and `D` respectively denote Γ, σ, and Δ. The first clause obviously corresponds to $\Gamma \vdash \cdot : \cdot$.

The second clause requires a bit of explanation because of the proliferation of dependencies on the variable `x` in the head; let us see why the dependencies are needed.

- The target context `G` can potentially depend on `x` because, in particular, $x{:}A \vdash x/x : x{:}A$. Indeed, any identity

[6]Indeed, representing substitutions in terms of copying is so natural that this use is explicitly remarked upon in [18].

substitution would simply re-create the variables in the source typing context.
- The binding M can depend on x because that is the only way we have of defining identity substitutions.
- The rest of the substitution SS can depend on x because that is how we can map multiple variables to the same variable. In particular, $x{:}A \vdash x/x, x/y : x{:}A, y{:}A$.
- However, D *must not* depend on x, because otherwise the context of `x T :: D` would not be a well-formed typing context. Recall that every variable in a typing context occurs exactly once.

Formalizing the meta-theorems Let us now explore the ramifications of this view of substitutions as contexts of cp assumptions in a ternary relation with source and target typing contexts. Weakening, for instance, is immediate.

```
Theorem ofsub_weak : forall G SS D,
  ofsub G SS D ->
  forall T, nabla x, ofsub (of x T :: G) SS D.
```

The proof is by a trivial induction on the first hypothesis.

Formalizing substitution for arbitrary terms (first rule in Theorem 3.2) is somewhat more involved. The definition in Figure 2 was given in a functional style, but cp obviously encodes a relation. Therefore, we have to prove that the relation is a function, i.e., that it is total and produces unique outputs.

```
Theorem cp_total : forall G SS D M T,
  ofsub G SS D -> {D |- of M T} ->
  exists MC, {SS |- cp M MC} /\
             {G |- of MC T}.
```

As expected, this is proved by a straightforward induction on the second hypothesis.

```
Theorem cp_determin : forall G SS D M MC1 MC2,
  ofsub G SS D ->
  {SS |- cp M MC1} -> {SS |- cp M MC2} ->
  MC1 = MC2.
```

Note that the = here is the built in λ-equivalence that is part of the reasoning logic of Abella. This theorem is proved by induction on the second hypothesis, but in the base case for a free variable of M that is in the domain of SS we need to appeal to a more basic well-formedness theorem about substitutions.

```
Theorem ofsub_determin : forall G SS D X M1 M2,
  ofsub G SS D ->
  member (of X M1) SS -> member (of X M2) SS ->
  M1 = M2.
```

In effect this says that the same variable, X, can have at most one binding in a well-formed substitution. The proof of this property relies crucially on the fact that in the second clause of ofsub, the variable D is *not* allowed to depend on x.[7]

[7]Nevertheless, the structure of this proof should be familiar from proofs of uniqueness of (Church-encoded) typing, as can be found in the Abella examples suite and in the tutorial [2].

Composition The second rule of Theorem 3.2 deals with applying a substitution to a substitution, i.e., with composing substitutions. Its definition can be found in the second part of Figure 3. As before with ofsub, there is a proliferation of dependencies of the source variable x to account for the fact that both the first and the second substitutions could have identity-like cases. Armed with this definition, we can show that whenever UU is the composition of SS and TT, then it is the case that applying UU to any term gives the same result as first applying TT and then SS. This formalizes Theorem 3.3.

```
Theorem comp_assoc :
  forall G SS D TT W UU M N K T,
  ofsub G SS D -> ofsub D TT W ->
  comp SS TT UU ->
  {W |- of M T} ->
  {TT |- cp M N} -> {SS |- cp N K} ->
  {UU |- cp M K}.
```

The proof, as expected, is by induction on the fourth hypothesis, `{W |- of M T}`, which is a proxy for induction on the structure of M.

In the base case where M is a free variable x that is assigned a type in W, we require an interesting lemma. Since x is in W and `ofsub D TT W`, it must be that x is copied to some term, say `N x`, in TT. By the definition of comp, it must therefore be that SS rewrites `N x` to some `K x` and, moreover, UU rewrites x to the very same `K x`. We thus obtain a version of the comp_assoc theorem specialized for free variables, which may seem to have a cumbersome statement without the benefit of the explanation above.

```
Theorem comp_var :
  forall G SS D TT W UU N K, nabla x,
  ofsub (G x) (SS x) (D x) ->
  ofsub (D x) (TT x) (W x) ->
  comp (SS x) (TT x) (UU x) ->
  member (cp x (N x)) (TT x) ->
  {SS x |- cp (N x) (K x)} ->
  {UU x |- cp x (K x)}.
```

This lemma is proved by a straightforward induction on the structure of TT, i.e., on the fourth hypothesis.

All that remains is to formalize the second rule in Theorem 3.2, but it is mostly trivial.

```
Theorem comp_total : forall G SS D TT W,
  ofsub G SS D -> ofsub D TT W ->
  exists UU, comp SS TT UU /\ ofsub G UU W.
```

(We can also show that composition is deterministic, but that is rarely needed in practice.)

Analysis One conclusion from these meta-theorems may be that the 2LLA as implemented by Abella is already expressive enough to be able to give a first-class treatment to simultaneous substitutions, which end up being just another kind of context, similar in spirit to typing contexts. The meta-theoretic properties of simultaneous substitutions can be stated and proved as ordinary Abella theorems. Since no modifications are required to the Abella reasoning kernel, we have a high level of assurance of correctness.

However, just because something is definable does not necessarily mean that it is the most convenient from a practical standpoint. Formal methods are as much art as science, and one must always pay attention to the kinds of tedium and boilerplate that are introduced by any technique. Too much tedium erases all the benefits of the higher degree of trust. Indeed, without any modifications to the Abella system, the user is expected to perform the following steps in order to obtain a first-class representation of substitutions whenever they set about encoding an object language.

1. Define the typing predicate `of` for the particular object language, and then characterize typing contexts using a relation such as `tyctx`.
2. Define the copying predicate `cp` and the corresponding ternary relation characterizing well-formed substitution contexts derived from `cp` assumptions – the `ofsub` predicate. Unless the object language is particularly exotic, both the `ofsub` and `comp` predicates are likely to remain unchanged from one object language to another.
3. Prove theorems such as `cp_total`, `cp_determin`, and `comp_assoc`, which proceed by induction

This is unarguably tedious since it is only difficult and enlightening the first time one does it. Moreover, the proofs of the meta-theorems are only long, not difficult; their inductive arguments are simple and predictable. We can therefore envisage that these theorems can be derived and proved automatically from a given signature of constants for an object language. Indeed, a predicate such as `cp` is merely a restatement of the inductive equality argument on higher-order terms, and so it can be fully automatically derived from the type signature. Once Abella is extended with this kind of automated meta-theory, we would be able to remove the tedium without sacrificing the high assurance level of certified meta-theorems.

Meta-theoretic aside: On two forms of copy An important consideration for our proposed view of substitutions as local contexts of `cp` predicates is that it relies on a formulation of `cp` that shares the same bound variables between input and output terms. In effect, our definition assumes that `cp` is reflexive on bound variables, and hence reflexive on all closed terms by structural induction.

It is entirely valid to define the copy predicate in a more general form that does not assume this reflexivity. One just has to change the clauses for copying bound variables: instead of sharing them, the clause would treat them as a transformation from related variables to related uses. Let us illustrate this by writing it as `cp2` to distinguish it from the `cp` defined earlier. Its definition is mostly the same as `cp`, except the clauses for binders are changed as follows (using `abs` as the example):

```
cp2 (abs R) (abs RC) :-
  pi x\ pi xc\ (cp2 x xc => cp2 (R x) (RC xc)).
```

The clauses for fix and case are modified similarly.

On closed terms, the two variants `cp` and `cp2` would be equivalent, but they would be different on open terms where the alternative presented above would only encode a partial equivalence relation. Somewhat surprisingly, it is possible to build a version of simultaneous substitutions that uses this kind of definition as well. Here is what the corresponding definition of `ofsub` would appear as, written as `ofsub2` to distinguish it from the earlier definition.

```
Define ofsub2 : olist -> olist -> olist -> prop by
  ofsub2 G nil nil ;
  nabla x, ofsub2 G (cp x M :: SS) (of x T :: D) :=
    {G |- of M T} /\ ofsub2 G SS D.
```

All the spurious dependencies on the source variable that existed in `ofsub` disappears in `ofsub2`, yielding a considerably cleaner definition.

So why not adopt this definition instead of the one based on `cp`? It turns out that this definition has a surprising feature: the source and target typing contexts in the `ofsub2` relation must have *disjoint domains*. As a consequence, there is no way to define an identity substitution. Or, more correctly, the equivalent concept to "identity substitution" is "fresh renaming substitution". More formally, this disjointedness is characterized as follows:

```
Define disjdom : olist -> olist -> prop by
  disjdom G nil ;
  nabla x, disjdom G (of x T :: D) := disjdom G D.
```

and we can prove:

```
Theorem ofsub2_disjdom : forall G SS D,
  ofsub2 G SS D -> disjdom G D.
```

Most of the meta-theorems that were proved using `cp` and `ofsub` can be proved using `cp2` and `ofsub2` as well. The sole wrinkle is with of `comp_assoc`, where when we assume that `ofsub2 G SS D` and `ofsub2 D TT W`, we must in addition assume that `disjdom G W` in order to build a composition `UU` for which `ofsub2 G UU W`. This is because disjointedness is not transitive.

4 Illustrative Example: Substitutivity of the Howe Relation

While the meta-theory of simultaneous substitution is important to get right, a more convincing illustration of the proposed perspective of substitutions as contexts would be to use it in a formalization where substitutions play a critical role. We use the example of showing the substitutivity of the Howe relation on PCF, which is an intermediate step to showing that applicative similarity is a pre-congruence. This very example has been attempted in Abella before on a restricted PCF with no recursive types [21]. That paper lamented the cumbersomeness of lifting a closed relation to open terms by means of inductive definitions instead of using the pen-and-paper definition based on universal quantification over substitutions. The purpose and peculiarities of the Howe relation are explained in much more detail in [21, 22, 29], so we omit it here.

Applicative similarity A family of relations $R : \mathsf{ty} \times \mathsf{tm} \times \mathsf{tm}$ on PCF terms—which we write as $s\ R\ t : A$ instead of $R(A, s, t)$—is said to be a *simulation* if it satisfies the following conditions.

1. $s\ R\ t : \mathsf{nat}$ iff:
   a. whenever $s \Downarrow 0$, also $t \Downarrow 0$; and
   b. whenever $s \Downarrow \mathsf{s}(s')$, also $t \Downarrow \mathsf{s}(t')$ and $s'\ R\ t' : \mathsf{nat}$.
2. $s\ R\ t : A \to B$ iff whenever $s \Downarrow \lambda x.\, s'$, also $t \Downarrow \lambda x.\, t'$ such that for every term $u{:}A$ we have $[u/x]s'\ R\ [u/x]t' : B$.

The union of all simulations is also a simulation relation that, by the standard Tarskian construction, can be seen as the greatest fixed point of a suitable functional. This greatest simulation is called *similarity* and is written $\preceq$; two terms $s, t{:}A$ are said to be *similar* if $s \preceq t : A$ holds.

In Abella, we would formulate such a greatest fixed point by means of a co-inductive definition.

```
CoDefine sim : tm -> tm -> ty -> prop by
  sim S T nat :=
     ({eval S zero} -> {eval T zero})
     /\ (forall S', {eval S (succ S')} ->
         exists T', {eval T (succ T')} /\
         sim S' T' nat) ;
  sim S T (arr A B) :=
     forall S', {eval S (abs S')} ->
     exists T', {eval T (abs T')} /\
     forall U, {of U A} -> sim (S' U) (T' U) B.
```

The definition of similarity is lifted to *open similarity* on open terms as the 4-ary relation $\preceq^{\mathrm{o}} : \mathsf{tyctx} \times \mathsf{tm} \times \mathsf{tm} \times \mathsf{ty}$, written infix, where $\Gamma \vdash s \preceq^{\mathrm{o}} t : A$ iff for every σ such that $\cdot \vdash \sigma : \Gamma$, it is the case that $[\sigma]s \preceq [\sigma]t : A$. Again this is straightforward to represent in Abella.

```
Define osim : olist -> tm -> tm -> ty -> prop by
  osim G S T A :=
    forall SS, ofsub nil SS G ->
    exists SC TC, {SS |- cp S SC} /\ {SS |- cp T TC}
                 /\ sim SC TC A.
```

A number of properties of open similarity are easy to state and prove given the meta-theorems about substitutions from the previous section. Below are the most important ones.

```
Theorem osim_weak : forall G S T A,
  osim G S T A ->
  forall B, nabla x, osim (of x B :: G) S T A.

Theorem osim_trans : forall G S T U A,
  osim G S T A -> osim G T U A ->
  osim G S U A.
```

Perhaps the most interesting of these theorems is closure under substitution (cus) of open similarity:

```
Theorem osim_cus : forall G SS D S T A SC TC,
  ofsub G SS D ->
  osim D S T A ->
  {SS |- cp S SC} -> {SS |- cp T TC} ->
  osim G SC TC A.
```

Its proof is a simple application of `comp_assoc`.

$$\frac{\Gamma, x{:}A \vdash x \preceq^{\mathrm{o}} t : A}{\Gamma, x{:}A \vdash x \preceq^{\mathrm{H}} t : A}$$

$$\frac{\Gamma \vdash 0 \preceq^{\mathrm{o}} t : \mathsf{nat}}{\Gamma \vdash 0 \preceq^{\mathrm{H}} t : \mathsf{nat}} \qquad \frac{\Gamma \vdash s \preceq^{\mathrm{H}} t : \mathsf{nat} \qquad \Gamma \vdash \mathsf{s}(t) \preceq^{\mathrm{o}} u : \mathsf{nat}}{\Gamma \vdash \mathsf{s}(s) \preceq^{\mathrm{H}} u : \mathsf{nat}}$$

$$\frac{\Gamma, x{:}A \vdash s \preceq^{\mathrm{H}} t : B \qquad \Gamma \vdash \lambda x.\, t \preceq^{\mathrm{o}} u : A \to B}{\Gamma \vdash \lambda x.\, s \preceq^{\mathrm{H}} u : A \to B}$$

$$\frac{\Gamma, x{:}A \vdash s \preceq^{\mathrm{H}} t : A \qquad \Gamma \vdash \mathsf{fix}\, x.\, t \preceq^{\mathrm{o}} u : A}{\Gamma \vdash \mathsf{fix}\, x.\, s \preceq^{\mathrm{H}} u : A}$$

$$\frac{\Gamma \vdash s \preceq^{\mathrm{H}} s' : A \to B \qquad \Gamma \vdash t \preceq^{\mathrm{H}} t' : A \qquad \Gamma \vdash s'\ t' \preceq^{\mathrm{o}} u : B}{\Gamma \vdash s\ t \preceq^{\mathrm{H}} u : B}$$

$$\frac{\begin{array}{c}\Gamma \vdash s \preceq^{\mathrm{H}} s' : \mathsf{nat} \quad \Gamma \vdash t_z \preceq^{\mathrm{H}} t_z' : A \quad \Gamma, x{:}\mathsf{nat} \vdash t_s \preceq^{\mathrm{H}} t_s' : A \\ \Gamma \vdash \mathsf{case}\, s'\, \mathsf{of}\ \{0 \Rightarrow t_z', \mathsf{s}(x) \Rightarrow t_s'\} \preceq^{\mathrm{o}} u : A\end{array}}{\Gamma \vdash \mathsf{case}\, s\, \mathsf{of}\ \{0 \Rightarrow t_z, \mathsf{s}(x) \Rightarrow t_s\} \preceq^{\mathrm{H}} u : A}$$

. . . .

$$\frac{}{\Gamma \vdash \cdot \preceq^{\mathrm{H}} \cdot : \cdot} \qquad \frac{\Gamma \vdash \sigma \preceq^{\mathrm{H}} \tau : \Delta \qquad \Gamma \vdash s \preceq^{\mathrm{H}} t : A}{\Gamma \vdash \sigma, s/x \preceq^{\mathrm{H}} \tau, t/x : \Delta, x{:}A}$$

Figure 4. The Howe relation and its substitutive extension

The Howe relation The main lemma needed to show open similarity, and hence applicative similarity, is a pre-congruence is that it is *substitutive*. However, it is not possible to prove substitutivity of open simulation by simple induction, because the inductive hypothesis is too weak in the case of bindings. Howe's method of establishing substitutivity is by means of a different candidate relation, called the Howe relation, that can be shown to be substitutive by a simple induction [29]. (Separately, it is shown to both contain and be contained in the open simulation relation; this is easier than showing substitutivity and we ignore it here.)

The Howe relation, written $\preceq^{\mathrm{H}}$, has the same type as the open similarity relation and is defined inductively with the collection of inference rules in the first part of Figure 4. It is extended to relate substitutions in a point-wise manner, with the rules in the second part of the figure. Both relations are straightforwardly represented in Abella.

```
Define howe : olist -> tm -> tm -> ty -> prop by
  howe G X T A :=
    member (of X A) G /\ osim G X T A ;
  howe G zero T nat := osim G zero T nat ;
  howe G (succ S) U nat :=
    howe G S T nat /\ osim G (osim T) U nat ;
  howe G (abs S) U (arr A B) :=
    exists T,
      (nabla x, howe (of x A :: G) (S x) (T x) B)
      /\ osim G (abs T) U (arr A B) ;
  howe G (fix S) U A :=
    exists T,
      (nabla x, howe (of x A :: G) (S x) (T x) A)
      /\ osim G (fix T) U A ;
  howe G (app S T) U B :=
    exists S' A, howe G S S' (arr A B) /\
      exists T',   howe G T T' A /\
      osim G (app S' T') U B ;
    howe G (case S Tz Ts) U A :=
      exists S', howe G S S' nat /\
      exists Tz', howe G Tz Tz' A /\
      exists Ts', (nabla x, howe (of x nat :: G)
                             (Ts x) (Ts' x) A) /\
      osim G (case S' Tz' Ts') U A.

  Define howesub : olist -> olist -> olist -> olist
                   -> prop by
    howesub G nil nil nil ;
    nabla x, howesub (G x)
                     (cp x (M x) :: SS x)
                     (cp x (N x) :: TT x)
                     (of x A :: D) :=
      nabla x, howesub (G x) (SS x) (TT x) D /\
      howe (G x) (M x) (N x) A.
```

The main lemma about the Howe relation is that it is substitutive, meaning that if Howe-related substitutions are applied to Howe-related terms, they stay Howe-related.

```
  Theorem howe_subst : forall G SS TT D S T A SC TC,
    howesub G SS TT D ->
    howe D S T A ->
    {SS |- cp S SC} -> {TT |- cp T TC} ->
    howe G SC TC A.
```

This theorem is proved by a simple induction on the second hypothesis, `howe D S T A`. In the base case for free variables it uses an additional lemma that is proved by induction on the `howesub` hypothesis.

Evaluation As should be expected, the entire conceptual difficulty in this formalization is contained in the definition of the Howe relation. Once it is properly defined, proving substitutivity is merely lengthy. Importantly, this proof does not need to resort to any definitional tricks that were used in the earlier attempt without first-class substitutions [21]. The formalization presented here formalizes the pen-and-paper definitions without any shortcuts.

5 Other Related Work

Alternative definitions of substitutions Of course, explicit substitutions can be defined as first-order list-like data structures. This style can be found in the existing examples on explicit substitutions in Abella [3, 31]. The difference between our proposal to use the 2LLA and first-order encodings of substitutions is that ours has a cleaner separation of concerns. The traversal of the structure of terms is compartmentalized to a simple copy predicate that can be automatically derived from the type signature. We then simply augment this traversal with a binding context to obtain substitutions on open terms. With list-like encodings of substitutions, one ends up conflating the issues of variable management and traversal. It may ultimately be a matter of taste if one prefers explicit lists or if one exploits the 2LLA as proposed here.

Substitutions in other representations There has been a number of proposals for representing binding constructs in formal reasoning systems. Among first-order representations at one extreme are fully *named* representations that represent variables using first-order datatypes (e.g., strings); both α-equivalence and capture-avoiding substitution have to be programmed manually by means of explicit sets of "used" names and variable naming conventions such as the Barendregt convention. This was the approach taken in the self-certification project for HOL-Light, and complications resulting from encoding capture-avoiding substitutions was remarked to be a pain point [16]. The other extreme of first-order representations uses De Bruijn indexes, which obviates α-equivalence and yields a simple induction principle. However, capture-avoiding substitution still has to be programmed by means of term-traversal operations and index management. Between these two extremes are a family of mixed representations that use both names and indexes, usually for free and bound names respectively. A popular variant is the *locally nameless* approach that systematizes the index management operations by means of an open/close protocol [4]. Explicit simultaneous substitutions in any of these first-order representations amount to simple lists of terms or term-variable pairs. Nothing in particular comes for free from the reasoning framework. Many libraries have been built to handle these first-order representations; some examples: [8, 32].

A close cousin of first-order representations is the use of nominal sets [28] as available in the Nominal Isabelle/HOL system [35], for instance. The need for α-equivalence disappears since terms are considered identical up to equivariance. Moreover, since all binding constructs become definable by means of permutations and name restriction [6], there is a straightforward induction principle for the structure of encoded terms. However, capture-avoiding substitution is not intrinsically available and has to be coded. In fact, many other kinds of substitution are also definable, including *capturing* substitutions, which are semantically questionable.[8]

For higher-order representations, there are two main alternatives to λTS. The first is a so-called *weak* HOAS that uses a different sort for variables and a separate variable constructor [17]. Weak HOAS retains the free α-equivalence of HOAS. The main reason for this change is that it makes the type signature for the constants purely positive, which means that they can be seen as the constructors of an inductive type. The proof system thus derives the induction principle for terms. Unfortunately, the induction principle that is derived is too general as it covers not only the encodings of terms but also *exotic terms* that are not in the image of any object language term but are inhabitants of the inductive type. To restore adequacy, one then needs to reason modulo a filter that removes these exotic terms. Explicit substitutions in

[8] It was suggested in [34] that first-class simultaneous substitutions were inexpressible using HOAS while they are easy to encode using nominal sets. The present paper can perhaps be a counterpoint: simultaneous substitutions are as first-class as typing contexts when it comes to meta-theory.

weak HOAS are definable as long as the type of variables is assumed to behave classically, i.e., have a decidable equality.

A different kind of higher-order encoding comes from *parametric HOAS* (PHOAS), which exploits polymorphism to define the object language at a higher kind [7]. This approach gives α-equivalence for free and many kinds of capture-avoiding substitutions are trivially definable. Unfortunately, the induction principle that is derived for a PHOAS encoding does not correspond to the standard structural induction on terms in a suitable grammar.[9] As a consequence, many simple inductive properties of object terms are not definable by structural recursion. I am not aware of any efforts to define first-class substitutions in PHOAS.

A qualitative comparison with [22] As already remarked in the introduction, the Beluga system has had built-in support for (simultaneous) substitutions from the very beginning. In fact, much of the meta-theory that has been formalized in this paper is also built-in and part of Beluga's type-checker. It's a trite observation that there is an obvious trade-off between more explicit (and therefore) certified meta-theory and the efficiency (and hence concision) that is afforded by a more sophisticated framework. In this regard, with the exception of the formalization of Theorems 3.2 and 3.3, and modulo the differences in the style of expressing theorems (functional dependently typed programs vs. tactics-based proof scripts) the formalizations in Beluga and Abella are evenly matched. It is more remarkable that the 2LLA allows the Abella-based formalization to avoid excessive clutter of auxiliary lemmas and definitions, which is a common experience when formalizing meta-theory in Abella without exploiting the 2LLA, as we observed in [5].

6 Perspectives

This paper shows how to represent (simultaneous) substitutions as a first-class concept in the two-level logic approach by treating substitutions as a kind of context related to the usual typing contexts. The purported benefit of this perspective is twofold: (1) the trusted part of the formal reasoning system remains untouched, and (2) the meta-theory of substitutions can be certified. A potential benefit that is not expanded on here but that will be a topic for followup work will be the automatic derivability of the boilerplate meta-theory of substitutions from a given type signature. We expect that this can be done similarly to certified context schemas [30].

While the mathematical contributions of this paper are slim, there is one aspect that may be worth further investigation: what does the decomposition of substitution as a computational traversal and a meta-theoretic bookkeeping tell us about the nature of substitutions? For instance, one aspect of this factorization that was not particularly relevant to this paper is that the notion of applying an explicit substitution would be difficult to treat as a pure computation in λTS, because a λProlog program has no way of telling if a given term represents a variable or not. This seems to indicate that substitutions are more naturally part of the meta-language, and therefore one should resist the temptation to add them to the object language. Another interesting question is what other aspects of meta-theory can be decomposed in a similar way. A good place to look may be among logical relations style meta-theorems.

[9]It has become idiomatic to build the expected induction principle by means of "closedness" axioms.

References

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The POPLmark Challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference (Lecture Notes in Computer Science)*. Springer, 50–65.

[2] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. 2014. Abella: A System for Reasoning about Relational Specifications. *Journal of Formalized Reasoning* 7, 2 (2014). https://doi.org/10.6092/issn.1972-5787/4650

[3] Horace Blanc and Beniamino Accattoli. 2016. Relating π-calculus and λ-calculus by means of the linear substitution calculus. Available at https://github.com/abella-prover/abella/tree/master/examples/process-calculi/pic_lambda. (2016).

[4] Arthur Charguéraud. 2011. The Locally Nameless Representation. *Journal of Automated Reasoning* (May 2011), 1–46. https://doi.org/10.1007/s10817-011-9225-2

[5] Kaustuv Chaudhuri, Leonardo Lima, and Giselle Reis. 2016. Formalized Meta-Theory of Sequent Calculi for Substructural Logics. In *Workshop on Logical and Semantic Frameworks, with Applications (LSFA)*. Post proceedings version to appear; Formalization https://github.com/meta-logic/abella-reasoning.

[6] James Cheney. 2005. A Simpler Proof Theory for Nominal Logic. In *8th International Conference on the Foundations of Software Science and Computational Structures (FOSSACS) (Lecture Notes in Computer Science)*, Vol. 3441. Springer, 379–394.

[7] Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 143–156. https://doi.org/10.1145/1411204.1411226

[8] Adam Chlipala. 2010. A Verified Compiler for an Impure Functional Language. In *Proceedings of POPL'10*.

[9] Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *J. of Symbolic Logic* 5 (1940), 56–68.

[10] Amy Felty and Alberto Momigliano. 2012. Hybrid: A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax. *J. of Automated Reasoning* 48 (2012), 43–105.

[11] Andrew Gacek. 2008. The Abella Interactive Theorem Prover (System Description). In *Fourth International Joint Conference on Automated Reasoning (Lecture Notes in Computer Science)*, A. Armando, P. Baumgartner, and G. Dowek (Eds.), Vol. 5195. Springer, 154–161. http://arxiv.org/abs/0803.2305

[12] Andrew Gacek. 2009. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. Ph.D. Dissertation. University of Minnesota.

[13] Andrew Gacek, Dale Miller, and Gopalan Nadathur. 2011. Nominal abstraction. *Information and Computation* 209, 1 (2011), 48–73. https://doi.org/10.1016/j.ic.2010.09.004

[14] Andrew Gacek, Dale Miller, and Gopalan Nadathur. 2012. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning* 49, 2 (2012), 241–273. https://doi.org/10.1007/s10817-011-9218-1

[15] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (1993), 143–184.

[16] John Harrison. 2006. Towards self-verification of HOL-Light. In *Proceedings of IJCAR'06 (Lecture Notes in Computer Science)*, Vol. 4130. Springer, 177–191.

[17] Furio Honsell, Marino Miculan, and Ivan Scagnetto. 2001. An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In *Proc. ICALP'01 (Lecture Notes in Computer Science)*. Springer, 963–978.

[18] Dale Miller. 1991. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. In *Extensions of Logic Programming: International Workshop, Tübingen (Lecture Notes in Artificial Intelligence)*, Peter Schroeder-Heister (Ed.), Vol. 475. Springer, 253–281.

[19] Dale Miller. 2010. Reasoning about Computations Using Two-Levels of Logic. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS'10) (Lecture Notes in Computer Science)*, K. Ueda (Ed.). 34–46. http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/aplas10.pdf

[20] Dale Miller and Gopalan Nadathur. 2012. *Programming with Higher-Order Logic*. Cambridge University Press. https://doi.org/10.1017/CBO9781139021326

[21] Alberto Momigliano. 2012. A supposedly fun thing I may have to do again: a HOAS encoding of Howe's method. In *Proceedings of LFMTP'12*. ACM, 33–42.

[22] Alberto Momigliano, Brigitte Pientka, and David Thibodeau. 2017. A case-study in programming coinductive proofs: Howe's method. Submitted. (2017).

[23] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Model Type Theory. *ACM Trans. on Computational Logic* 9, 3 (2008), 1–49.

[24] Frank Pfenning. 2001. Logical Frameworks. In *Handbook of Automated Reasoning (in 2 volumes)*, John Alan Robinson and Andrei Voronkov (Eds.). Elsevier and MIT Press, 1063–1147.

[25] Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 199–208.

[26] Brigitte Pientka. 2008. 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)*. ACM, 371–382.

[27] Brigitte Pientka and Joshua Dunfield. 2010. Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description). In *Fifth International Joint Conference on Automated Reasoning (Lecture Notes in Computer Science)*, J. Giesl and R. Hähnle (Eds.). 15–21.

[28] Andrew M. Pitts. 2003. Nominal Logic, A First Order Theory of Names and Binding. *Information and Computation* 186, 2 (2003), 165–193.

[29] Andrew M. Pitts. 2011. Howe's method for higher-order languages. In *Advanced Topics in Bisimulation and Coinduction*, David Sangiorgi and Jan Rutten (Eds.). Cambridge University Press, Chapter 5, 197–232.

[30] Olivier Savary-Bélanger and Kaustuv Chaudhuri. 2014. Automatically Deriving Schematic Theorems for Dynamic Contexts. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2014)*. ACM.

[31] Anders Schack-Nielsen and Carsten Schürmann. 2011. *The lambda sigma calculus and strong normalization*. Technical Report. IT University of Copenhagen, Denmark. TR-2011-150.

[32] Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Proceedings of ITP'15 (Lecture Notes in Artificial Intelligence)*, Xingyuan Zhang and Christian Urban (Eds.). Springer.

[33] Peter Schroeder-Heister. 1993. Rules of Definitional Reflection. In *8th Symp. on Logic in Computer Science*, M. Vardi (Ed.). IEEE Computer Society Press, IEEE, 222–232. https://doi.org/10.1109/LICS.1993.287585

[34] Christian Urban. 2008. Nominal Reasoning Techniques in Isabelle/HOL. *Journal of Automated Reasoning* 40, 4 (2008), 327–356.

[35] Christian Urban and Christine Tasson. 2005. Nominal Techniques in Isabelle/HOL. In *20th Conf. on Automated Deduction (CADE) (Lecture Notes in Computer Science)*, R. Nieuwenhuis (Ed.), Vol. 3632. Springer, 38–53.

[36] Yuting Wang, Kaustuv Chaudhuri, Andrew Gacek, and Gopalan Nadathur. 2013. Reasoning about Higher-Order Relational Specifications. In *Proceedings of the 15th International Symposium on Princples and Practice of Declarative Programming (PPDP)*, Tom Schrijvers (Ed.). Madrid, Spain, 157–168. https://doi.org/10.1145/2505879.2505889