Nominal Formalisations of Typical SOS Proofs

Julien Narboux\textsuperscript{a} Christian Urban\textsuperscript{a,}\textsuperscript{*}

\textsuperscript{a}TU Munich, Germany.

Abstract

Structural operational semantics (SOS) provides a framework for ascribing semantics to programming languages. This is typically done by stating rules for typing judgements, small-step transitions and rules for evaluating an expression of the language. Structural inductions over expressions and inductions over inference rules are thus the most fundamental reasoning techniques employed in SOS. While the SOS-techniques are characterised in Plotkin’s seminal notes as “symbol-pushing”, programming languages nearly always contain binders and then reasoning is in fact rather subtle. We describe in this paper formalisations of typical proofs in SOS within the Isabelle proof assistant using the nominal datatype package. We show how this package eases the subtleties when reasoning about binders.

Key words: structural operational semantics, proof assistants, nominal techniques, Isabelle/HOL

“It is the purpose of these notes to develop a simple and direct method for specifying the semantics of programming languages. Very little is required in the way of mathematical background all that will be involved is “symbol-pushing” of one kind or another of the sort which will already be familiar to readers with experience of either the non-numerical aspects of programming languages or else formal deductive systems of the kind employed in mathematical logic.” — G. D. Plotkin [10, Page 19]

1 Introduction

Structural operational semantics (SOS) introduced by Plotkin has been very successful in describing what programs are supposed to do. One reason for this success

* Corresponding author: Christian Urban

Email addresses: narboux@in.tum.de (Julien Narboux), urbanc@in.tum.de (Christian Urban).

Preprint submitted to Elsevier 26 February 2007
is that the mathematical theory underlying such descriptions is often very simple and involves, according to Plotkin, only “symbol-pushing”. However, if one is concerned with formally proving properties about these SOS-descriptions and the programming language at hand involves binders, then reasoning can be rather subtle. Therefore we like to examine Plotkin’s claim in the context of formal SOS-proofs.

The naïve method of representing binders using abstract syntax-trees is overly concrete: it does not take into account $\alpha$-equivalence where expressions are regarded as equal, if they only differ in the naming of bound variables. As a result one has to deal explicitly with naming issues and prove properties modulo $\alpha$-equivalence. This leads to formal proofs where one has to deal with many details, even if one proves only very simple properties. In any case the reasoning using the naïve method is not the simple kind of “symbol-pushing” Plotkin had certainly in mind. Of course one can reconcile abstract syntax-trees and binders by using de-Bruijn indices. This alleviates the problems about too many details and in some cases leads to very slick proofs. Unfortunately, by using de-Bruijn indices the “symbol-pushing” involves a rather large amount of arithmetic on indices, which is not present in informal descriptions. Another method of representing binders is by using higher-order abstract-syntax (HOAS) where the meta-language provides binding-constructs. The disadvantage we see with HOAS is that one has to encode the language at hand. In practice this means often that reasoning does not proceed as one would expect from the familiar informal reasoning.

In this paper we describe the nominal datatype package [12, 11], which provides an infrastructure in the theorem prover Isabelle/HOL [6] for representing binder as named $\alpha$-equivalence classes. We describe this package in a tutorial style in the context of typical proofs in SOS. The paper does not present any new results, just sharpens issues about formally stating and proving properties about SOS-descriptions. The language we formalise is a simply typed lambda-calculus extended with products, sums and natural numbers. The goals of our formalisation are similar to those of the PoplMark Challenge [1], that means we aim to formalise proofs that follow closely the informal reasoning, but are of course completely formal. The nominal datatype package is designed to support this kind of reasoning by providing an infrastructure which relieves the formal reasoner from many details. For example it provides automatically the notion of freshness [8], written $x \not\equiv e$, of a variable $x$ with respect to an expression $e$. This is a notion which seem to arise naturally in many arguments about $\alpha$-equivalence classes.

The paper is organized as follows: Terms and substitutions are defined in Sec. 2, together with a description of strong structural induction principles that have the usual variable convention already built in. Sec. 3 defines types and the typing-judgement for terms. We will explain in detail the problems that arise when one needs to match $\alpha$-equivalence classed against inductive definitions. Sec. 4 introduces the big-step evaluation relation for terms and in Sec. 5 we show how the proof of the termination property for the evaluation relation proceeds.
2 Terms and Substitutions

The language we consider here is the familiar lambda-calculus enriched with terms for products, sums and natural numbers. For building up these terms we assume the type of natural numbers and introduce the type \textit{name} for variables. The only property we need to know about \textit{name} is that it consists of infinitely many variables. The terms are defined by the clauses

\textbf{Definition 1 (Terms)}

\[
\text{trm ::= Var name} \\
\text{App trm trm} \\
\text{Lam name.trm} \\
\text{Const n} \\
\text{Pair trm trm} \\
\text{Fst trm} \\
\text{Snd trm} \\
\text{InL trm} \\
\text{InR trm} \\
\text{Case trm of InL name.trm | InR name.trm}
\]

where in the \textit{Lam}-clause, as usual, a variable is bound (similarly in the clause for \textit{Case} where a variable is bound in each of the two \textit{Case}-branches). Because the nominal datatype package allows us to write terms as $\text{Lam } x.e$, one might assume that this definition represents “raw”, or un-quotient, syntax-trees. However, this is \textit{not} the case: when stating the definition about terms, the nominal datatype package generates $\alpha$-equivalence classes. This is illustrated by the equation

\[
\text{Lam } x. \text{Lam } y. (\text{App (Var } x (\text{Var } y)) = \text{Lam } y. \text{Lam } x. (\text{App (Var } y (\text{Var } x))
\]

which holds for the terms defined above. The most important operation we need for terms is substitution. In the proofs we present later on it will be necessary to introduce the slightly more complicated notion of simultaneous substitution, which we represent as finite lists of $(\text{name, trm})$-pairs. One reason for this choice is that it is easier to deal with finite structures in the nominal datatype package than with infinite ones (an infinite representation of substitutions is, for example, partial maps from \textit{name to trm} as used by Plotkin [10]). Using our list representation we define:
Definition 2 (Simultaneous Substitution)

\[
\begin{align*}
\theta(\text{Var } x) &= \text{lookup } \theta \ x \\
\theta(\text{App } e_1 \ e_2) &= \text{App } \theta(e_1) \ \theta(e_2) \\
\theta(\text{Const } n) &= \text{Const } n \\
\theta(\text{Pair } e_1 \ e_2) &= \text{Pair } \theta(e_1) \ \theta(e_2) \\
\theta(\text{Fst } e) &= \text{Fst } \theta(e) \\
\theta(\text{Snd } e) &= \text{Snd } \theta(e) \\
\theta(\text{InL } e) &= \text{InL } \theta(e) \\
\theta(\text{InR } e) &= \text{InR } \theta(e) \\
\theta(\text{Lam } x: e) &= \text{Lam } x: \theta(e) \quad \text{provided } x \# \theta \\
\theta(\text{Case } e \ of \ \text{InL } x.e_1 \ | \ \text{InR } y.e_2) &= \text{Case } \theta(e) \ of \ \text{InL } x.\theta(e_1) \ | \ \text{InR } y.\theta(e_2) \\
&\quad \text{provided } x \# \theta, y \# \theta
\end{align*}
\]

where in the first clause we use the auxiliary function \text{lookup} defined by the clauses:

\[
\begin{align*}
\text{lookup } [] \ x &= \text{Var } x \\
\text{lookup } ((y, e):\theta) \ x &= \text{if } x = y \text{ then } e \text{ else } \text{lookup } \theta \ x
\end{align*}
\]

Single substitutions are a derived concept by setting

\[
e[x := e'] \equiv [(x, e')](e)
\]

where \([(x, e')]\) is a singleton list.

Despite the side-conditions attached to the Lam- and Case-clause, the definition above yields a total function, since we work with \(\alpha\)-equivalence classes where renamings are always possible. Clearly, if defined over “raw” terms, this definition would be a partial function. While the totality of the substitution operation is rather convenient in a formal proofs, it also means that we must be careful when defining functions over the structure of terms. This is because we can specify functions over the structure of terms that lead to inconsistencies. One example is the function that returns the immediate subterms of a lambda-term, specified by

\[
\begin{align*}
\text{ist } (\text{Var } x) &= \emptyset \\
\text{ist } (\text{App } e_1 \ e_2) &= \{ e_1, e_2 \} \\
\text{ist } (\text{Lam } x. e) &= \{ e \}
\end{align*}
\]

If this function could be defined, then we can prove false. This is because we expect that functions return the same output for the same input. The problem with the inconsistency can then be seen by considering the “same” terms

\[
\text{Lam } x. (\text{Var } x) = \text{Lam } y. (\text{Var } y)
\]

and the calculation

\[
\begin{align*}
\text{ist } (\text{Lam } x. (\text{Var } x)) &= \{ \text{Var } x \} \\
\text{ist } (\text{Lam } y. (\text{Var } y)) &= \{ \text{Var } y \}
\end{align*}
\]
If we force both right-hand sides to be equal by assuming that \( ist \) is a function, we have an inconsistency: \( \{ Var\ x\} \neq \{ Var\ y\} \) provided \( x \neq y \).

In order to prevent such inconsistencies, the recursion combinator the nominal datatype package automatically provides for the terms in Def. 1 only allows us to define functions that respect \( \alpha \)-equivalence classes. For this we are required in our formalisation to manually check that certain conditions about the clauses in Def. 2 are satisfied. They are quite complex (see [9, 11]) and omitted here. Another complication is that the provided recursion combinator requires us to state the \( Case\)-clause as

\[
\theta(\text{Case } e \text{ of InL } x\. e_1 \mid \text{InR } y\. e_2) = \text{Case } \theta(e) \text{ of InL } x\. \theta(e_1) \mid \text{InR } y\. \theta(e_2)
\]

provided \( y \neq x, x \not\in (e, e_2, \theta) \) and \( y \not\in (e, e_1, \theta) \)

and we have to manually prove using explicit \( \alpha \)-conversions that the clause given in Def. 2 holds. This kind of manual reasoning involving explicit \( \alpha \)-conversions is necessary whenever a term-constructor has at least one binder and an argument which is not in the scope of this binder. That is why we do not have to do anything in case of \( Lam \).

To ease such manual \( \alpha \)-conversions the nominal datatype package defines automatically a renaming operation. This operation, written \( \pi \cdot e \), takes a term \( e \) and a permutation \( \pi \), which is a finite list of \((name, name)\)-pairs and permutes every variable in the term \( e \). We write such permutations as \((a_1, b_1)(a_2, b_2)\cdots(a_n, b_n)\); the empty list \([]\) stands for the identity permutation. The permutation operation acting on a term then is defined by

**Definition 3 (Permutations Acting on Terms)**

\[
\begin{align*}
\pi \cdot \text{Var } x &= \text{Var } \pi \cdot x \\
\pi \cdot \text{Lam } x\. e &= \text{Lam } \pi \cdot x.\pi \cdot e \\
\pi \cdot \text{App } e_1 \cdot e_2 &= \text{App } \pi \cdot e_1 \cdot \pi \cdot e_2 \\
\pi \cdot \text{Const } n &= \text{Const } n \\
\pi \cdot \text{Pair } e_1 \cdot e_2 &= \text{Pair } \pi \cdot e_1 \cdot \pi \cdot e_2 \\
\pi \cdot \text{Fst } e &= \text{Fst } \pi \cdot e \\
\pi \cdot \text{Snd } e &= \text{Snd } \pi \cdot e \\
\pi \cdot \text{InL } e &= \text{InL } \pi \cdot e \\
\pi \cdot \text{InR } e &= \text{InR } \pi \cdot e \\
\pi \cdot \text{Case } e \text{ of InL } x\. e_1 \mid \text{InR } y\. e_2 &= \text{Case } \pi \cdot e \text{ of InL } \pi \cdot x.\pi \cdot e_1 \mid \text{InR } \pi \cdot y.\pi \cdot e_2
\end{align*}
\]

using the auxiliary operation of a permutation acting on a variable

\[
[] \cdot a = a \\
(a_1, a_2) :: \pi \cdot a = \begin{cases} 
a_2 & \text{if } \pi \cdot a = a_1 \\
(a_1, a) & \text{if } \pi \cdot a = a_2 \\
\pi \cdot a & \text{otherwise}
\end{cases}
\]

Using the permutation operation, the nominal datatype package defines automatically a notion of \( \alpha \)-equivalence for abstractions. This is defined by distinguishing
whether the binders of two abstractions are equal or not. This gives the following two rules:

\[
\begin{align*}
& e_1 = e_2 \\
& x = y \\
& x.e_1 = x.e_2 \\
& x \neq y \\
& e_1 = (x y) e_2 \\
& x \neq e_2 \\
& x.e_1 = y.e_2
\end{align*}
\]

Having the notion of \(\alpha\)-equivalence for abstractions in place, the package defines under which conditions two terms are equal. The rules for \textit{Lam}, \textit{Case} and \textit{App} are given below.

\[
\begin{align*}
& x.e_1 = y.e_2 \\
& Lam x.e_1 = Lam y.e_2 \\
& e = e' \\
& x.e_1 = x'.e_1' \\
& y.e_2 = y'.e_2' \\
& Case e \text{ of } InL x.e_1 \mid InR y.e_2 = Case e' \text{ of } InL x'.e_1' \mid InR y'.e_2' \\
& e_1 = e_1' \\
& e_2 = e_2' \\
& App e_1 e_2 = App e_1' e_2'
\end{align*}
\]

The rules for the other cases follow the pattern for \textit{App}. Armed with the rules about \(\alpha\)-equivalence, we can start to prove properties about terms and substitutions. Later on, for example, we will need the property how a single and a simultaneous substitution interact. For this we prove the following lemma:

**Lemma 4** If \(x \neq \emptyset\) then \(\theta(e)[x= e'] = ((x, e')::\theta)(e)\).

whose proof is by induction on the structure of \(e\). For such proofs the nominal datatype package provides two versions of the induction principle—a weak one and a strong one. The weak one proves a property \(P e\) for all terms \(e\) provided one establishes for each term-constructor an implication that assumes the property for the arguments and concludes the property for the term-constructor. This pattern follows what Plotkin [10, Page 49] describes as \textit{structural induction for expressions}.

As an inference rule the weak induction principle looks as follows:

\[
\begin{align*}
& \forall x. P \text{ Var } x \\
& \forall x e. P e \longrightarrow P \text{ Lam } x.e \\
& \forall e_1 e_2. P e_1 \land P e_2 \longrightarrow P \text{ App } e_1 e_2 \\
& \forall n. P \text{ Const } n \\
& \forall e_1 e_2. P e_1 \land P e_2 \longrightarrow P \text{ Pair } e_1 e_2 \\
& \forall e. P e \longrightarrow P \text{ Fst } e \\
& \forall e. P e \longrightarrow P \text{ Snd } e \\
& \forall e. P e \longrightarrow P \text{ InL } e \\
& \forall e. P e \longrightarrow P \text{ InR } e \\
& \forall e x e_1 y e_2. P e \land P e_1 \land P e_2 \longrightarrow P \text{ Case } e \text{ of } InL x.e_1 \mid InR y.e_2 \\
& P e
\end{align*}
\]
Using this principle all cases in Lem. 4 are quite routine, except the cases for Lam and Case where one has to analyse binders. In the Lam-case, for example, we have the induction-hypothesis

$$\forall x \theta. x \neq \theta \rightarrow \theta(x) = \theta(x)$$

and have to show

$$\theta(Lam y.e)[x:=e'] = ((x, e')::\theta)(Lam y.e)$$

for arbitrary $y$ and $e$. However we only know that $x \neq \theta$ holds. In order to apply the definition of substitution and subsequently use the induction hypothesis we need to rename the binder $y$ to a fresh variable $z$, say. This makes the proof quite clunky (especially in case of Case where two $\alpha$-conversions are needed) and too hard to be found by the automatic search tools available in Isabelle. In informal proofs establishing such properties by induction, one usually ignores the fact that one has to establish the property at hand for an arbitrary bound variable; rather one employs the convention that binders are always assumed to be suitable fresh (see for example [2]). In the case above this means we have the convention that $y$ is fresh for $\theta$, $x$ and $e'$, that is $y \neq \theta$, $y \neq x$ and $y \neq e'$ hold. With this convention also the cases Lam and Case are trivial.

To support this kind of informal reasoning where one does not consider truly arbitrary bound variables, but rather bound variables about which various freshness assumptions are made, the nominal datatype package derives automatically from the weak induction principle a strong version (see [12]). This strong induction principle has, roughly speaking, a variable convention already built in. It establishes the property $P c e$ for all terms $e$ and as inference rule is as follows:

$$\forall c x. P c x \quad \forall c x. P c x \quad \forall c e. x \neq c \rightarrow P c (Lam x.e) \quad \forall c e_1, e_2. (\forall c. P c e_1) \land (\forall c. P c e_2) \rightarrow P c (App e_1 e_2)$$

$$\forall c e. (\forall c. P c e) \rightarrow P c (Const n) \quad \forall c e. (\forall c. P c e) \rightarrow P c (Pair e_1 e_2) \quad \forall c e. (\forall c. P c e) \rightarrow P c (Fst e) \quad \forall c e. (\forall c. P c e) \rightarrow P c (Snd e)$$

$$\forall c e. (\forall c. P c e) \rightarrow P c (InL e) \quad \forall c e. (\forall c. P c e) \rightarrow P c (InR e)$$

$$\forall c e. y_1 y_2. x \neq y \land x \neq (c, e, e_1) \land y \neq (c, e, e_1) \land (\forall c. P c e) \land (\forall c. P c e_1) \land (\forall c. P c e_2) \rightarrow P c (Case e of InR x.e_1 \mid InR y.e_2)$$

The purpose of the parameter $c$, called the induction context, is to accommodate the assumptions we make in informal reasoning about the freshness of the binder. In the Lam-case we can then assume that the binder for which the property needs to be established is fresh with respect to this context (see highlighted formula). In case
of Case we can assume that both binders are fresh for this context and moreover are fresh for the terms that are not in their scope and are mutually distinct. With these assumptions in place the cases for Lam and Case are completely routine: we only have to instantiate the induction context with the tuple \((\theta, x, e')\). The only requirement we have to observe with this instantiations is that the context may only mention finitely many free variables. This holds in our case. We have the same induction hypothesis as in the weak version
\[
\forall x \theta e'. x \not\equiv \theta \rightarrow \theta(e)[x:=e'] = ((x, e')::\theta)(e)
\]
However additionally we have that \(y \not\equiv \theta\), \(y \not\equiv x\) and \(y \not\equiv e'\). To return to our proof obligation
\[
\theta(Lam y. e)[x:=e'] = ((x, e')::\theta)(Lam y. e)
\]
we can now move \(\theta\) and the single substitution under the lambda-abstraction on the left-hand side (similarly with \((x, e')::\theta\) on the right-hand side), and then apply the induction hypothesis. As a result all cases of Lem. 4 are routine and the formal proof is completely automatic, except for the need of mentioning some properties about lookup.

To sum up this section: the nominal datatype package derives a strong version of the induction principle automatically for all term-calculi, not just the one defined in Def. 1. This makes reasoning by structural induction over \(\alpha\)-equivalence classes rather pleasant, because no explicit \(\alpha\)-conversions are needed. This is a theme which will thread through many proofs we shall describe the next sections.

3 Typing

Many SOS-proofs involve typing-information for terms. In this section we define types and a typing relation for our terms. Our definition of types, for which we use the letter \(T\), separates data-sorts, ranged over by using the letter \(S\), and function types:

**Definition 5 (Data and Types)**

\[
\begin{align*}
data & ::= \quad nat \\
& \quad data \times data \\
& \quad data + data \\
\end{align*}
\]
\[
\begin{align*}
ty & ::= \quad Data data \\
& \quad ty \rightarrow ty
\end{align*}
\]

Before we can define a typing-judgement, we need to state what typing contexts are. We use lists of \((name,ty)\)-pairs since, as mentioned before, in the nominal datatype package it is easier to work with finite structures than infinite ones (if we use sets of \((name,ty)\)-pairs instead, then it is inconvenient to exclude potentially infinitely large typing-contexts). The disadvantage of using lists is of course that
we distinguishing the order of how variables are associated to types. However, in
terms of convenience this choice will not cause any problem.

A typing-context \( \Gamma \) is valid, provided it includes only a single association for every
variable occurring in \( \Gamma \). This is defined as predicate by the two rules

\[
\begin{align*}
\text{valid } [] & \quad \text{valid } \Gamma & \quad \text{valid } (x, T)::\Gamma \\
\text{valid } [] & \quad \text{valid } \Gamma & \quad \text{valid } (x, T)::\Gamma
\end{align*}
\]

Armed with this notion, the definition of the typing-judgement (see Fig. 1) is rela-
tively standard: We need to enforce at the leaves (i.e. in \( T_{\text{VAR}} \) and in \( T_{\text{CONST}} \))
that the typing-contexts are valid. This implies that the typing-contexts are valid in
all derivable judgements. In rule \( T_{\text{VAR}} \) we use the notation \((x, T) \in \Gamma\) to stand for
list-membership.

Care, however, needs to be taken in the rules involving binders. The slightly non-
standard freshness condition in the rules \( T_{\text{LAM}} \) and \( T_{\text{CASE}} \) allow the nominal
datatype package to derive a strong induction principle for this definition (this is
similar to the strong induction principle that is derived for terms). This stronger
induction principle greatly simplifies proofs by induction over the typing rules.
The idea behind these freshness-conditions is that they ensure that no binder in a
rule occurs freely in its conclusion and in case a rule contains more than one binder,
then all of them are distinct. Let us illustrate this for the \( T_{\text{LAM}} \)-rule. Its conclusion
is \( \Gamma \vdash \text{Lam } x.e : T_1 \rightarrow T_2 \) and in the premise we have \( x \not\in \Gamma \). Now the fact that \( x \)
is not free in the conclusion is easily confirmed: it cannot occur in \( \Gamma \) because of
the freshness-condition; it is obviously fresh for \( \text{Lam } x.e \) and it cannot occur in the
type \( T_1 \rightarrow T_2 \) because in our setting types do not contain any variables.

We next prove the properties of weakening and type-substitutivity for the typing-
judgement (the former property is needed in the latter). Although these proper-
ties are quite simple, the formal reasoning needs some effort before the proofs go
through smoothly.

Proofs of the weakening lemma are often claimed to be straightforward or left as an
exercise to the reader ([7] is one example amongst many). Because it is purported
to be so straightforward, the literature does not seem to include any explicit proof
of this lemma that can be called straightforward. When attempting to construct a
formal proof of this lemma in our setting, several obstacles arise—some of them
are caused by our use of lists as typing-context, others are independent from this
representation and are caused by the binders in our language. To circumvent the
first kind of obstacles we define the notion of a sub-context as follows:

**Definition 6 (Sub-Contexts)** \( \Gamma_1 \subseteq \Gamma_2 \equiv \forall x T. \ (x, T)\in \Gamma_1 \longrightarrow (x, T)\in \Gamma_2 \).

and state the weakening-lemma in terms of sub-contexts:
\[
\begin{align*}
\text{valid } \Gamma & \quad (x, T) \in \Gamma \\
\frac{}{\Gamma \vdash \text{Var } x : T} & \quad \text{T_VAR} \\
\Gamma \vdash e_1 : T_1 \rightarrow T_2 & \quad \Gamma \vdash e_2 : T_1 \\
\frac{}{\Gamma \vdash \text{App } e_1 e_2 : T_2} & \quad \text{T_APP} \\
x \# \Gamma & \quad (x, T_1) :: \Gamma \vdash e : T_2 \\
\frac{}{\Gamma \vdash \text{Lam } x. e : T_1 \rightarrow T_2} & \quad \text{T_LAM} \\
\text{valid } \Gamma & \quad \frac{}{\Gamma \vdash \text{Const } n : \text{Data nat}} & \quad \text{T_CONST} \\
\Gamma \vdash e_1 : \text{Data } S_1 & \quad \Gamma \vdash e_2 : \text{Data } S_2 \\
\frac{}{\Gamma \vdash \text{Pair } e_1 e_2 : \text{Data } S_1 \times S_2} & \quad \text{T_PR} \\
\Gamma \vdash e_1 : \text{Data } S_1 \\
\frac{}{\Gamma \vdash \text{Fst } e : \text{Data } S_1} & \quad \text{T_FST} \\
\Gamma \vdash e_2 : \text{Data } S_2 \\
\frac{}{\Gamma \vdash \text{Snd } e : \text{Data } S_2} & \quad \text{T_SND} \\
\Gamma \vdash e_1 : \text{Data } S_1 \\
\frac{}{\Gamma \vdash \text{InL } e : \text{Data } S_1 + S_2} & \quad \text{T_INL} \\
\Gamma \vdash e_2 : \text{Data } S_2 \\
\frac{}{\Gamma \vdash \text{InR } e : \text{Data } S_1 + S_2} & \quad \text{T_INR} \\
\frac{x_1 \# (\Gamma, e, e_2, x_2 \quad x_2 \# (\Gamma, e, e_1, x_1)}{\Gamma \vdash \text{Case } e \text{ of InL } x_1. e_1 \mid \text{InR } x_2. e_2 : T} \quad \text{T_CASE}
\end{align*}
\]

Fig. 1. Typing judgement for terms.

**Lemma 7 (Weakening)**

If \( \Gamma_1 \vdash e : T \) and \( \text{valid } \Gamma_2 \) and \( \Gamma_1 \subseteq \Gamma_2 \) then \( \Gamma_2 \vdash e : T \).

In the proof of this lemma all cases are routine, except the ones which involve binders. This is because the usual (that is weak) induction principle coming for "free" with the definition of the typing-rules does not deal well with binders. Consider the naïve attempt of proving the suitably generalised property of weakening, namely

\[
\Gamma_1 \vdash e : T \longrightarrow (\forall \Gamma_2. \text{valid } \Gamma_2 \longrightarrow \Gamma_1 \subseteq \Gamma_2 \longrightarrow \Gamma_2 \vdash e : T).
\]

Then in the lambda-case we have the induction hypothesis

\[
\forall \Gamma_2. \text{valid } \Gamma_2 \longrightarrow (x, T_1) :: \Gamma_1 \subseteq \Gamma_2 \longrightarrow \Gamma_2 \vdash e : T_2
\]

which we like to use with \( \Gamma_2 = (x, T_1) :: \Gamma_2 \). However this will not allow us to make any progress as we cannot obtain \( (x, T_1) :: \Gamma_2 \vdash e : T \). This is because we only know that \( x \) is fresh for the smaller typing context \( \Gamma_1 \) and we cannot infer anything for the bigger context \( \Gamma_2 \). Consequently we cannot ascertain whether \( \text{valid } ((x, T_1) :: \Gamma_2) \) holds. To get the proof through the naïve way, we have to rename the binder first, at
which point the simplicity of the often cited straightforward proof disappears (see [4, 5, 8]): the inductive hypothesis is much harder to show applicable because it mentions $e$, but the desired goal is in terms of $e[x:=z]$. This will require a lemma showing the invariance of the typing-judgement under renamings.

The renaming can be completely avoided if we use a strong version of the induction principle that has the usual variable convention built in. The formal proof is then very close to being straightforward. The implication in the lambda-case is in the strong induction principle as follows.

\[
\forall c \ x \ \Gamma \vdash e \ T_1 \ T_2, \ x \not\in \Gamma \land (\forall c. \ P \ c \ ((x, T_1) : : \Gamma) \ e \ T_2) \\
\rightarrow P \ c \ \Gamma \ Lam \ x. e \ (T_1 \rightarrow T_2)
\]

In order to obtain the desired freshness condition about $x \not\in \Gamma_2$ in our proof, we only have to set the induction context $c$ to $\Gamma_2$ (this also solves the same problem in the Case-rule). The resulting formalised proof is not completely automatic, but quite routine just as one would expect taking the simplicity of the informal proof into account where one appeals to the variable convention.

As an easy consequence of our version of the weakening lemma, we also obtain a context-exchange lemma for our representation of contexts as lists.

**Corollary 8** If $(x_1, T_1):=(x_2, T_2):\Gamma \vdash e : T$ then $(x_2, T_2):=(x_1, T_1):\Gamma \vdash e : T$.

Next we will establish the type-substitutivity lemma, which we will be crucial later on when showing the type-preservation property.

**Lemma 9 (Type-Substitutivity)**
If $(x, T') : : \Gamma \vdash e : T$ and $\Gamma \vdash e' : T'$ then $\Gamma \vdash e[x:=e'] : T$.

We prove this lemma by induction over the term $e$ using again the strong version of the induction principle. The principle will do again the “trick” of avoiding any renaming of binders. However, in this proof arises another problem to do with $\alpha$-equivalence classes: because of the two typing-judgements in the premise of this lemma we must be able to infer in the lambda-case for example that the typing-judgement $(x, T') : : \Gamma \vdash Lam \ y. e : T$ implies that there exists a $T_1$ and $T_2$ such that

\[(y, T_1):=(x, T') : : \Gamma \vdash e : T_2 \quad \text{and} \quad T = T_1 \rightarrow T_2\]

holds. We need this property in order to apply the induction hypothesis, which in this case is of the form

\[(x, T') : : (y, T_1) : : \Gamma \vdash e : T_2 \land (y, T_1) : : \Gamma \vdash e' : T' \rightarrow (y, T_1) : : \Gamma \vdash e[x:=e'] : T_2\]
In informal “pencil-and-paper” reasoning we would just take \((x, T')::\Gamma \vdash Lam y.e : T\) and match it against all rules defining the typing-judgements. This matching leaves us with the \(T_{\text{LAM}}\)-rule as the only possibility. Inverting this rule gives us on “paper” the desired facts. However, this matching-method is not directly available in Isabelle. What is available is an elimination rule that states for the typing-judgement \(\Gamma' \vdash e' : T'\)

\[
\exists \Gamma x T. \Gamma' = \Gamma \land e' = \text{Var} x \land T' = T \land \text{valid} \land (x, T) \in \Gamma
\]

\[
\lor \exists \Gamma x T_1 T_2 e. \Gamma' = (x, T_1)::\Gamma \land e' = Lam x.e \land T' = T_1 \rightarrow T_2 \land x \not\in \Gamma
\]

\[
\land (x, T_1)::\Gamma \vdash e : T_2
\]

\[
\lor \exists \Gamma T_1 T_2 e_1 e_2. \Gamma' = \Gamma \land e' = App e_1 e_2 \land T' = T_2 \land \Gamma \vdash e_1 : T_1 \rightarrow T_2
\]

\[
\land \Gamma \vdash e_2 : T_1
\]

\[
\ldots
\]

where the three dots stand for the other possibilities corresponding to the remaining seven typing-rules. Using this elimination rule we can complete the proof for the rules without binders. For example in case of \(\Gamma' \vdash App e_1 e_2' : T'\) we obtain

\[
\Gamma' = \Gamma \land \text{App} e_1' e_2' = \text{App} e_1, e_2 \land T' = T_2 \ldots
\]

from which we can infer that \(e_1' = e_1\) and \(e_2' = e_2\). This is because \(\text{App}\) is an injective term-constructor where equality is defined as

\[
\frac{e_1' = e_1 \quad e_2' = e_2}{\text{App} e_1', e_2' = \text{App} e_1, e_2}
\]

In contrast, when we use the elimination rule in case of \(\Gamma' \vdash Lam x'.e' : T'\) then reasoning becomes awkward: In this case we obtain

\[
\Gamma' = (x, T_1)::\Gamma \land Lam x'.e' = Lam x.e \land T' = T_1 \rightarrow T_2 \ldots
\]

from which we cannot infer \(x' = x\) and \(e' = e\) as there are two possibilities for two lambda-abstractions to be equal, namely

\[
\frac{x' = x \quad e' = e}{Lam x'.e' = Lam x.e}
\]

\[
\frac{x' \neq x \quad e' = (x' \cdot x) \cdot e \quad x' \neq e}{Lam x'.e' = Lam x.e}
\]

As a consequence we have to check whether both possibilities give us the desired substitutivity property. And in the second case this is annoyingly burdensome. We fare slightly better if we strengthen the elimination rule so that we can “eliminate” a typing-judgement as follows:

**Lemma 10** If \(\Gamma' \vdash Lam x'.e' : T'\) and \(x' \neq \Gamma'\) then

\[
\exists T_1 T_2. (x', T_1)::\Gamma' \vdash e' : T_2 \land T' = T_1 \rightarrow T_2.
\]

Note that in order to prove this lemma we need the condition \(x' \neq \Gamma'\)—essentially we can only apply the elimination rule familiar from informal matching if we know
That the binder does not occur freely in the relation we want to match. Such a lemma is needed for all inference rules that contain somewhere a binder. What is unpleasant about this is the fact that in the nominal datatype package there is not yet any substantial support for automating these lemma. Clearly, they should be provided by the infrastructure, but at the moment they need to be derived manually. Once this is done, however, the proof of the type-substitutivity lemma is quite routine, although for this lemma only little help is provided by Isabelle's automatic proving-tactics.

4 Big-Step Evaluation Relation

In this section we define the usual big-step call-by-value operational semantics. The inference rules are given in Fig. 2. We further introduce a predicate which filters out values. Its definition is
An important property we can to establish for evaluation is that it preserves types.

**Lemma 11 (Subject Reduction)** If $e \downarrow e'$ and $\Gamma \vdash e : T$ then $\Gamma \vdash e' : T$.

The proof of this lemma is quite routine when we have a strong induction principle for the evaluation relation at our disposal. The only case that needs manual interference is the B_APP where we have to use Lem. 9 and the elimination rules from Lem. 10. Another important property is that the evaluation relation produces unique results. This can be stated as follows.

**Lemma 12 (Unicity)** If $e \downarrow e_1$ and $e \downarrow e_2$ then $e_1 = e_2$.

The proof of this lemma is by rule induction over the evaluation relation. Unlike the proof of the subject reduction property where the strong induction principle is extremely helpful, in this lemma the strong version does not give much mileage. In fact the weak induction principle is adequate to get the induction through. The only cases where one has to establish the lemma by giving enough hints are the rules B_APP, B_CASEL and B_CASER.

A small lemma which is often overlooked in informal reasoning is that freshness is preserved by evaluation.

**Lemma 13 (Freshness Preservation)** If $e \downarrow e'$ and $x \not\in e$ then $x \not\in e'$.

This lemma can in our formalisation be discharged by a completely automatic induction on the evaluation relation. It will play an important rôle when we show in the next section that evaluation terminates for well-typed terms.

## 5 Termination

The last property we show here is that for every typable closed expression evaluates to a value, which means that the evaluation relation terminates.

**Theorem 14 (Termination)** If $\emptyset \vdash e : T$ then $\exists v. e \downarrow v \land val v$.

The proof of the this lemma is not that simple and only works if one proves a stronger result. For this we use the well-known techniques of introducing a logical relation and using then the properties of this relation to get the proof through. The specific logical relation we use here we will call *valuation*. They are sets of terms and come in two forms $V$ for types and $V'$ for data. They are defined as follows:
\[ \begin{align*}
V'\text{nat} & = \{\text{Const } n \mid n \in \text{UNIV}\} \\
V' S_1 \times S_2 & = \{\text{Pair } x \ y \mid x \in V' S_1 \land y \in V' S_2\} \\
V' S_1 + S_2 & = \{\text{InL } x \mid x \in V' S_1\} \cup \{\text{InR } y \mid y \in V' S_2\} \\
V (\text{Data } S) & = V' S \\
V (T_1 \to T_2) & = \{\text{Lam } x . e \mid \forall v \in V T_1. \exists v'. e[x:=v] \Downarrow v' \land v' \in V T_2\}
\end{align*} \]

The first clause states that \( V'\text{nat} \) includes all constants \( \text{Const } n \) (the set \( \text{UNIV} \) with appropriate typing annotations, which are omitted here, stands for the set of all natural numbers). The last clause includes the standard closure property for lambda-abstractions.

In the main lemma we will show that a typable term together with a closing substitution evaluates. In order to define what is meant by a closing substitution we introduce for simultaneous substitutions the notion \( \theta \text{ maps } x \to e \), which ensures that \( \theta \) contains the association \((x, e)\) not shadowed by any other association for \( x \).

**Definition 15** \( \theta \text{ maps } x \to e \equiv \text{lookup } \theta \ x = e \).

Next, we introduce a notion for when a substitution \( \theta \) closes a typable term, that means has an assignment for every \((x, T)\)-pair in a typing context \( \Gamma \), whereby the assignment in \( \theta \) must come from the valuation \( V T \).

**Definition 16** \( \theta \text{ Vcloses } \Gamma \equiv \forall x \ T. \ (x, T) \in \Gamma \longrightarrow (\exists v. \ \theta \text{ maps } x \to v \land v \in V T) \).

The first lemma we show is that \( \text{Vcloses} \) is preserved under suitable additional assignments to simultaneous substitutions and typing-contexts. This property is often called the **monotonicity**, or **preservation under weakening** [3].

**Lemma 17 (Monotonicity)** If \( \theta \text{ Vcloses } \Gamma \) and \( e \in V T \) and valid \((x, T)::\Gamma\) then \((x, e)::\theta \text{ Vcloses } (x, T)::\Gamma\).

The proof of this lemma is a routine case-distinction on the extended typing-context and simultaneous substitution. Now we are in a position to give a proof Theorem 14, where, however, we do not prove termination just for closed expressions, but for arbitrary typable terms.

**Lemma 18 (Termination on open Terms)** If \( \Gamma \vdash e : T \) and \( \theta \text{ Vcloses } \Gamma \) then \( \exists v. \ \theta(e) \Downarrow v \land v \in V T \).

**PROOF.** This proof is by a strong structural induction on \( e \), where we generalise over \( T \) and set up the induction so that in the cases \( \text{Lam} \) and \( \text{Case} \) we can assume the binders are fresh for \( \Gamma \) and \( \theta \). The interesting cases are \( \text{App}, \text{Lam} \) and \( \text{Case} \); the others are either completely automatic in our formalisation or quite straightforward. To illustrate our reasoning we give the details for \( \text{App} \) and \( \text{Lam} \).

**Case App:** By induction hypothesis we know that:
By assumption we know

\[ \text{(as}_1) \quad \theta \text{ Vcloses } \Gamma \quad \text{and} \quad \text{(as}_2) \quad \Gamma \vdash \text{App } e_1, e_2 : T \]

From the second assumption we can derive that \( \Gamma \vdash e_1 : T' \rightarrow T \) and \( \Gamma \vdash e_2 : T' \) hold by inversion of \( \text{t_APP} \) for some type \( T' \). Using the induction hypotheses and the first assumption we can derive that there exists a \( v_1 \) and \( v_2 \) such that:

\[ (i) \quad \theta(e_1) \Downarrow v_1 \quad \text{and} \quad v_1 \in V(T' \rightarrow T) \]

\[ (ii) \quad \theta(e_2) \Downarrow v_2 \quad \text{and} \quad v_2 \in V T' \]

From the first fact, we obtain by definition of \( V \) that \( v_1 \) must be of the form \( \text{Lam } x. e' \) whereby \( x \) can be assumed to be fresh for \( \theta, e_1 \) and \( e_2 \) (in this step we need again a strong elimination rule for the function \( V \)). This also implies that \( x \) is fresh for \( \theta(e_1) \) and \( \theta(e_2) \). We can further infer from the definition of \( V \) that:

\[ (iii) \quad \forall v \in V T'. \exists v'. e'[x:=v] \Downarrow v' \land v' \in V T \quad \text{and} \]

\[ (iv) \quad \theta(e_1) \Downarrow \text{Lam } x. e' \]

Now we combine (ii) and (iii) to obtain a \( v_3 \) such that

\[ (v) \quad e'[x:=v_2] \Downarrow v_3 \quad \text{and} \quad v_3 \in V T \]

holds. Since \( x \) is fresh for \( \theta(e_2) \) and freshness is preserved under evaluation (see Lem. 13), we have by (ii) that \( x \) is fresh for \( v_2 \). In turn this means that \( x \) is fresh for \( e'[x:=v_2] \), and hence by (v) also for \( v_3 \). Now we have \( x \not\in (\theta(e_1), \theta(e_2), v_3) \) which we can combine with (iv), (ii) and (v) to obtain by rule \( \text{B_APP} \) that

\[ \text{App } \theta(e_1) \theta(e_2) \Downarrow v_3 \]

holds. Using \( v_3 \) we can conclude that there exists a \( v \) such that:

\[ \theta(\text{App } e_1, e_2) \Downarrow v \quad \text{and} \quad v \in V T. \]

This completes the application-case. Its formalised version formulated in the readable Isar-language of Isabelle [13] can be found in Fig. 5.

Case \text{Lam}: By induction hypothesis we know that:

16
case $\text{App } e_1, e_2 : \Gamma \theta T$

have $ih_1 : \forall \theta \, \Gamma T. \, \theta \text{ Closes } \Gamma \implies \Gamma \vdash e_1 : T \implies \exists v. \, \theta(e_1) \downarrow v \land v \in V T$ by fact

have $ih_2 : \forall \theta \, \Gamma T. \, \theta \text{ Closes } \Gamma \implies \Gamma \vdash e_2 : T \implies \exists v. \, \theta(e_2) \downarrow v \land v \in V T$ by fact

have $as_1 : \forall \theta \, \Gamma \text{ Closes } \Gamma$ by fact

have $as_2 : \forall \theta \, \Gamma \vdash \text{App } e_1, e_2 : T$ by fact

from $as_2$ obtain $T'$ where $\Gamma \vdash e_1 : T' \implies T$ and $\Gamma \vdash e_2 : T'$ by auto

then obtain $v_1, v_2$

where $i) \theta(e_1) \downarrow v_1 \land v_1 \in V (T' \rightarrow T)$

and $ii) \theta(e_2) \downarrow v_2 \land v_2 \in V T'$ using $ih_1, ih_2, as_1$ by blast

from $i)$ obtain $x \ e^i$

where $v_1 = \text{Lam } x \ e^i$

and $ii) \forall v \in V T'. \, \exists v'. \, e'[x:=v] \downarrow v' \land v' \in V T$

and $iv) \theta(e_1) \downarrow \text{Lam } x \ e^i$

and $fr : x \neq \theta(e_1)$ and $fr_2 : x \neq \theta(e_2)$ by (simp-all add: fresh-psubst)

from $ii) (i)$ obtain $v_3$ where $v) \ e'[x:=v_2] \downarrow v_3 \land v_3 \in V T$ by auto

from $fr_2$ $ii)$ have $x \neq v_2$ by (auto simp add: fresh-preserved)

then have $x \neq e'[x:=v_2]$ by (simp add: fresh-subst-fresh)

then have $fr_3 : x \neq v_3$ using $v)$ by (auto simp add: fresh-preserved)

from $fr_1, fr_2, fr_3$ have $x \neq \theta(e_1), \theta(e_2), v_3$ by simp

with $iv) (ii) (v)$ have $\text{App } \theta(e_1), \theta(e_2) \downarrow v_3$ by auto

then show $\exists v. \, \theta(\text{App } e_1, e_2) \downarrow v \land v \in V T$ using $v)$ by auto

Fig. 3. The formal proof of termination for the $\text{App}$-case.

$$\forall \theta \, \Gamma T. \, \theta \text{ Closes } \Gamma \implies \Gamma \vdash e : T \implies (\exists v. \, \theta(e) \downarrow v \land v \in V T)$$

By assumption we know

$$(as_1) \quad \theta \text{ Closes } \Gamma$$

and

$$(as_2) \quad \Gamma \vdash \text{Lam } x \ e : T$$

Since we use a strong induction principle we know further the freshness conditions that $x \neq \Gamma$ and $x \neq \theta$. We can use them and the second assumption to infer that

$$(i) \quad (x, T_1) :: \Gamma \vdash e : T_2$$

$$(ii) \quad T = T_1 \rightarrow T_2$$

$$(iii) \quad \text{valid } ((x, T_1) :: \Gamma)$$

Where $iii)$ follows from $i)$ since the judgment $(x, T_1) :: \Gamma \vdash e : T_2$ implies that $(x, T_1) :: \Gamma$ must be valid.

Next we are going to show that $\text{Lam } x.\theta(e) \in V (T_1 \rightarrow T_2)$. By definition of $V$, it therefore suffices to show that

$$\exists v'. \, \theta(e)[x:=v] \downarrow v' \land v' \in V T_2$$
case (Lam x e Γ θ T)
  have ih:∀θ Γ T. θ Vcloses Γ ⊢ e : T =⇒ ∃v. θ(e) ⊑ v ∧ v ∈ V T by fact
  have as₁: θ Vcloses Γ by fact
  have as₂: Γ ⊢ Lam x.e : T by fact
  have fs: x # θ by fact
  from as₂ obtain T₁ T₂
    where (i): (x,T₁)::Γ ⊢ e:T₂ and (ii): T = T₁ → T₂ using fs by auto
  from (i) have (iii): valid ((x,T₁)::Γ) by (simp add: typing-valid)
  have ∀ v ∈ (V T₁). ∃v'. θ(e)[x:=v] ⊑ v' ∧ v' ∈ V T₂
  proof
    fix v
    assume v ∈ (V T₁)
    with (iii) as₁ have (x,v)::θ Vcloses (x,T₁)::Γ using monotonicity by auto
    with ih (i) obtain v' where ((x,v)::θ)(e) ⊑ v' ∧ v' ∈ V T₂ by blast
    then have θ(e)[x:=v] ⊑ v' ∧ v' ∈ V T₂ using fs by (simp add: psubst-psubst-bsubst)
    then show ∃v'. θ(e)[x:=v] ⊑ v' ∧ v' ∈ V T₂ by auto
  qed
  then have Lam x.θ(e) ∈ V (T₁ → T₂) by auto
  moreover
  have θ(Lam x.e) ⊑ Lam x.θ(e) using fs by auto
  ultimately show ∃v. θ(Lam x.e) ⊑ v ∧ v ∈ V T using (ii) by auto

Fig. 4. The formal proof of termination for the App-case.

holds for all v ∈ V T₁. We can use Lemma 17, (iii) and the first assumption to infer that (x,v)::θ Vcloses (x,T₁)::Γ. We can use this and (i) to instantiate the induction hypothesis, which gives us a v' such that

\[ ((x,v)::θ)(e) ⊑ v' ∧ v' ∈ V T₂ \]

holds. We know that this is equivalent to θ(e)[x:=v] ⊑ v' ∧ v' ∈ V T₂, since x is fresh for θ. This means we have shows that Lam x.θ(e) ∈ V (T₁ → T₂) holds and we also know by the freshness of x that θ(Lam x.e) is equal to Lam x.θ(e) and evaluates to Lam x.θ(e). Using this and (ii) we can take v to be Lam x.θ(e) and conclude with

\[ θ(Lam x.e) ⊑ v ∧ v ∈ V T. \]

The formalised version of this case is shown in Fig. 5.

\[ \square \]

6 Conclusion

We have described a formalisation of some very typical proofs from SOS. The main point we want to convey is that such proofs can be done relatively easily using the nominal datatype package in Isabelle/HOL. This must however be qualified
insofar as the nominal datatype package only supports languages involving simple, lambda-calculus-like binders. Although they can be of different type and can be iterated, more complicated binding structures such as binding a finite set of variables are not supported. One can encode such general binders using the simple binding, but this makes proofs quite complicated. The second qualification we must mention is that even though we based our formalisation on $\alpha$-equivalence classes reasoning about them can be quite subtle. Many of the complications can be hidden from the user, for example by automatically providing strong versions of the induction principles, but they cannot be hidden completely. Most notably issues about $\alpha$-equivalence show up in definitions of functions by structural recursion. On “paper” one is usually not concerned with questions about whether a function is compatible with $\alpha$-equivalence classes or whether it leads to an inconsistency. In the formal framework of the nominal datatype package, condition need to be verified which guarantee the compatibility with $\alpha$-equivalence classes.

The most annoying aspect in the presented formalisation is the need of deriving manually strong versions of the elimination rules for inductive definitions. For them one clearly would like to have substantial support or even wants that they can be derived automatically. Nevertheless there are a number of attractive points in our formalisations. Several of the presented formal proofs really proceed like the corresponding informal proofs done with “pencil-and-paper”. In this way we can give a formal justification for the informal reasoning and can also partly justify Plotkin’s remark about the “symbol-pushing” nature of SOS-methods.

Acknowledgements: To do this formalisation we have been inspired by Adam Chlipala who mailed some of the problems as challenge on the PoplMark list. We are very grateful to Nick Benton who outlined the proof for the termination property described in Sec. 5.

References


