# Uniqueness Typing Simplified—Technical Appendix

Edsko de Vries*
Department of Computer Science
Trinity College Dublin, Ireland
devriese@cs.tcd.ie

August 13, 2008

**Abstract**

This technical report is an appendix to *Uniqueness Typing Simplified* [7], in which we show how uniqueness typing can be simplified by treating uniqueness attributes as types of a special kind, allowing arbitrary boolean expressions as attributes, and avoiding subtyping. In the paper, we define a small core uniqueness type system (a derivative of the simply typed lambda calculus) that incorporates these ideas. We also outline how soundness with respect to the call-by-need semantics [11] can be proven, but we do not give any details. This report describes the entire proof, which is written using the proof assistant *Coq* [3]. The proof itself (as *Coq* sources) is also available and can be downloaded from the author's homepage[1].

## Contents



*Supported by the Irish Research Council for Science, Engineering and Technology.
[1] `http://www.cs.tcd.ie/~devriese`

# 1 Introduction

This technical report is an appendix to *Uniqueness Typing Simplified* [7], in which we show how uniqueness typing can be simplified by treating uniqueness attributes as types of a special kind, allowing arbitrary boolean expressions as attributes, and avoiding subtyping. In the paper, we define a small core uniqueness type system (a derivative of the simply typed lambda calculus) that incorporates these ideas. We also outline how soundness with respect to the call-by-need semantics [11] can be proven, but we do not give any details. This report describes the entire proof, which is written using the proof assistant *Coq* [3]. The proof itself (as *Coq* sources) is also available and can be downloaded from the author's homepage[2].

This report is structured as follows. In sections 2 and 3 we highlight some of the difficulties we faced when developing the proof, and discuss some of its more subtle aspects. In Section 4 we define the notion of an environment, various operations on environments, the kinding and typing relations, and the operational semantics for our language. Sections 5 and 6 prove numerous auxiliary lemmas that will be necessary in the

[2] `http://www.cs.tcd.ie/~devriese`

main proof, which is described in Section 7. Appendix A finally describes a formalization of boolean algebra, following Huntington's Postulates [10].

Every lemma in this report is preceded by a brief description of the lemma in informal language (English), followed by a precise statement of the lemma (in the syntax of *Coq*) and a brief description (again in English) of how the lemma can be proven. For most lemmas, this description will begin with "*By induction on...*" or "*By inversion on...*"; many descriptions will also include the most important other lemmas that the proof relies on. *Coq* verifies a proof strictly from top to bottom, so if a lemma $B$ relies on lemma $A$, $A$ must have been proven before lemma $B$; this therefore applies equally to the structure of this report. When the description of the proof does not mention induction or inversion, then these techniques are not necessary and the lemma can be proven by direct application of other lemmas.

What we do not show is the actual proofs themselves: there would be little point. The proofs have been verified by *Coq*, a widely respected proof assistant. If the reader nevertheless prefers to verify the proofs by hand, he will want to redo them himself; the short description of the proof should provide enough information to get started.

Besides, the proofs are written in the syntax of *Coq*. Coq is based on the calculus of constructions, a powerful version of the dependently typed lambda calculus. As such, a proof in *Coq* is a program (a term of the lambda calculus) that, given the premises, constructs a proof of the conclusion. However, in all but the most simple cases, these programs are too difficult to write by hand, and instead the proof consists of a list of calls to *tactics* which build up the program step-by-step.

Consider a simple example. Suppose we want to prove that $n + 0$ is equal to $n$ for all natural numbers $n$. Here is a full *Coq* proof of this property (this proof comes from the *Coq* standard library):

```
Lemma plus_n_O : forall n:nat, n = n + 0.
Proof.
  induction n; simpl in |- *; auto.
Qed.
```

Although it should be clear what `induction n` does, the purpose of the other tactics (such as `simpl` or `auto`) is less obvious, *even to an experienced Coq user*. Tactics interact with the current state of the proof assistant, which includes information such as which lemmas are available, the types of all variables, etc. Trying to interpret a *Coq* proof without *Coq* is akin to hearing one part of a telephone conversation: half the text is missing.

The actual proof constructed by these tactics is

$$\begin{aligned}\lambda(n:\texttt{nat})\cdot\texttt{nat\_ind}\ &(\lambda(m:\texttt{nat})\cdot m = m+0)\\ &(\texttt{refl\_equal}\ 0)\\ &(\lambda(m:\texttt{nat})(IH_m : m = m+0)\cdot \texttt{f\_equal}\ \mathsf{S}\ IH_m)\\ &n\end{aligned}$$

which makes use of various other lemmas, such as induction on natural numbers (`nat_ind`—essentially a fold operation), the fact that equality is reflexive (`refl_equal`) and a lemma that states that if $x = y$, then for all $f$, $f\ x = f\ y$ (`f_equal`). The details do not matter; the point is that this is hardly more readable than the original proof. In this report, we would simply describe this proof as "*By induction on n*".

# 2 Equivalence

Suppose we have a set $C$ of objects together with an equivalence relation $\approx$ on $C$, and some characterization $P$ of objects of $C$. We want $P$ to have the property that if $P\ x$ and $x \approx y$, then $P\ y$. There are three different ways in which we can guarantee that $P$ has this property.

- We can prove that $P$ has the required property.
- We can define $P$ over the quotient set $C/\approx$ instead. This will give us the desired property by definition.

- It may be possible to choose an alternative representation $C'$ of the objects in $C$, such that every equivalence set in $C'/\approx$ is a singleton set. In other words, so that the equivalence relation *is* the identity relation. The desired property of $P$ then holds trivially.

For example, take the set of lambda terms together with alpha-equivalence, and the property of being well-typed. Then,

- We can prove that well-typedness is equivariant: if $\lambda x \cdot x$ is well-typed, so is $\lambda y \cdot y$.
- We can define the well-typedness over the set of alpha-equivalent terms.
- We can represent lambda terms using De Bruijn notation, in which case $\lambda x \cdot x$ and $\lambda y \cdot y$ are both represented as $\lambda \cdot 0$.

Not all options are always practical, and each option has its advantages and disadvantages. For the specific example of alpha-equivalent terms, the first option may be possible, but cumbersome as we may have many properties over lambda-terms; we will have to prove equivariance for each one. The second approach is inconvenient when we need to refer to the name of the bound variable in an abstraction, for example in the typing rule for abstraction. The final approach does not have these shortcomings, but introduces new ones: many operations on lambda terms in De Bruijn notation must juggle with the indices, leading to additional complexity in proofs.

In informal proofs, we tend to gloss over this issue:

> In this situation the common practice of human (as opposed to computer) provers is to say one thing and do another. We say that we will quotient the collection of parse trees by a suitable equivalence relation of alpha-conversion, identifying trees up to renaming of bound variables; but then we try to make the use of alpha-equivalence classes as implicit as possible by dealing with them via suitably chosen representatives. How to make good choices of representatives is well understood, so much so that it has a name—the "Barendregt Variable Convention": choose a representative parse tree whose bound variables are fresh, i.e., mutually distinct and distinct from any (free) variables in the current context. This informal practice of confusing an alpha-equivalence class with a member of the class that has sufficiently fresh bound variables has to be accompanied by a certain amount of hygiene on the part of human provers: our constructions and proofs have to be independent of which particular fresh names we choose for bound variables. Nearly always, the verification of such independence properties is omitted, because it is tedious and detracts from more interesting business at hand. Of course this introduces a certain amount of informality into "pencil-and-paper" proofs that cannot be ignored if one is in the business of producing fully formalized, machine-checked proofs.
>
> —Andrew Pitts, *Nominal logic, a first order theory of names and binding* [12]

In the remainder of this section, we detail how we tackle this issue for the specific examples of terms under alpha-equivalence, typing environments under substructural rules and boolean expressions under Huntington's Postulates.

## 2.1 Lambda terms

We already described the problem of dealing with terms under alpha-equivalence in the introduction to this section, so all that remains is to discuss the solution. There are various proposals in the literature; we will adopt the *locally nameless* approach suggested by Aydemir *et al.* in *Engineering Formal Metatheory* [1] (we refer the reader to the same paper for an overview of alternatives).

In the locally nameless approach, bound variables are represented by De Bruijn indices, but free variables are represented by ordinary names. This means that alpha-equivalent terms are represented by the same term (and so we do not have to reason explicitly about alpha-equivalence), but we do not have to perform any arithmetic operations on terms. We do however have to solve one problem. Consider the typing rule for application. In the locally nameless style, the rule is

$$\frac{\Gamma, x : \tau \vdash e^x : \sigma \qquad \text{fresh } x}{\Gamma \vdash \lambda \cdot e : \tau \to \sigma}\textsc{Abs}$$

When we typecheck the body $e$, we "open it up" using a fresh variable $x$, and then record the type of the variable as normal. That is, we replace bound variable 0 (the variable that was bound by the lambda) by a fresh variable (for some definition of "fresh"). This is a consequence of the locally nameless approach: every time a previously bound variable becomes free, we have to invent a fresh name for it.

Without the freshness condition, we would be able to derive

$$\frac{\dfrac{\vdots}{x : \tau, x : \sigma \vdash (x, x) : (\sigma, \sigma)}}{x : \tau \vdash \lambda \cdot (0, x) : \sigma \to (\sigma, \sigma)}$$

where the (original) free variable $x$ has suddenly changed type (the typing environment acts as a binder, and the variable $x$ has been "captured"). The minimal freshness condition is therefore that the variable that is used to open up a term, does not already occur free in the term:

$$\frac{\Gamma, x : \tau \vdash e^x : \sigma \qquad x \notin \text{fv}\, e}{\Gamma \vdash \lambda \cdot e : \tau \to \sigma}\text{M-}\textsc{Abs}$$

A weak premise ($x \notin \text{fv}\, e$) is good when using rule Abs to prove the type of a term since we only have to show that $\Gamma, x : \tau \vdash e^x$ holds for one particular $x$. It is however not so good when doing induction on a typing relation. In that case, we know that the $e^x$ has type $\sigma$ *for one particular* $x$. But that $x$ may not be fresh enough for our purposes, at which point we need to rename the term to avoid name clashes. To circumvent this problem, Aydemir *et al.* [1, Section 4] propose to use cofinite quantification[3]:

$$\frac{\forall x \notin L \cdot \Gamma, x : \tau \vdash e^x : \sigma}{\Gamma \vdash \lambda \cdot e : \tau \to \sigma}\text{C-}\textsc{Abs}$$

To use C-Abs, we have to show that the $e^x$ has type $\sigma$ for all $x$ not in some set $L$, but using this rule is no more difficult than using M-Abs: we simply pick an arbitrary variable not in $L$. The induction principle however is now much stronger: we now know that $e^x$ has type $\sigma$ *for any* $x$ not in some set $L'$. Then when we have to prove that $\lambda \cdot e$ has type $\tau \to \sigma$, knowing that $e^x$ has type $\sigma$ for all $x$ not in $L'$, and we need $x$ to be distinct from some other variable $y$, we can simply apply rule Abs choosing $L' \cup \{y\}$ for $L$. We still occasionally need renaming lemmas, but they too become much more straightforward to prove when using cofinite quantification (we prove a number of renaming lemmas in Section 5.2).

Arthur Charguéraud, one of the authors of the *Engineering Formal Metatheory* paper, has developed a *Coq* library [6] which facilitates the use of the locally nameless representation of terms and the use of cofinite quantification. The proofs in this report will make essential use of this library, which we will dub the *Formal Metatheory* library. As an example, here is a trivial lemma that we can always pick a variable that is distinct from all other variables in a typing environment:

```
Lemma fresh_from_env : forall E e T fvars,
  E |= e ~: T | fvars -> exists x, x \notin dom E.
intros.
  pick_fresh x.
  exists x ; auto.
Qed.
```

The proof is essentially just a call to the `pick_fresh` from the *Formal Metatheory* library. This tactic collects all variables in the environment, and then chooses a variable that is distinct from all these variables. The proof that $x$ satisfies the necessary freshness condition is also handled automatically. The use of the locally nameless approach, and in particular the use of the *Formal Metatheory* library, meant that little of our subject reduction proof needs to be concerned with alpha-equivalence or freshness.

[3] A cofinite subset of a set $X$ is a subset $Y$ whose complement in $X$ is a finite set.

## 2.2 Environments

Consider this definition of a simple linear lambda calculus:

$$\frac{}{x : \tau \vdash x : \tau}\textsc{Var} \qquad \frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x \cdot e : \tau \to \sigma}\textsc{Abs} \qquad \frac{\Gamma \vdash f : \tau \to \sigma \qquad \Delta \vdash e : \tau}{\Gamma, \Delta \vdash f\ e : \sigma}\textsc{App}$$

Suppose we want to prove an exchange lemma:

> **Lemma** (Exchange). If $\Gamma, \Delta \vdash e : \tau$, then $\Delta, \Gamma \vdash e : \tau$.

In informal practice, we might not even consider proving this lemma, because we might represent environments as (multi-)sets so that $\Gamma, \Delta$ and $\Delta, \Gamma$ are the same environment. In a formal (constructive) proof, however, we must choose a concrete representation. If we represent environments by lists, we must prove *Exchange*, since $\Gamma, \Delta$ and $\Delta, \Gamma$ are certainly not the same list. Unfortunately, *the definition of the typing relation above does not permit Exchange*: *Exchange* does not hold.

One solution is to choose a different concrete representation. For example, if we choose to represent environments by sorted lists of pairs of variables and types (for some arbitrary ordering relation) then $\Gamma, \Delta$ and $\Delta, \Gamma$ again denote the same environment. Although this approach may work well, we have chosen not to use it for two reasons. It is probably sufficient to define the ordering relation entirely syntactically (ignoring any equivalence relation between types), but this ordering relation will not be intuitive (is $\forall a. \forall b. a \to b$ equal to, less than or greater than $\forall a. \forall b. b \to a$?). Since *Coq* verifies our proofs, but naturally cannot verify our definitions, we prefer not to have these doubts about the foundations of the proof.

The second reason we have chosen not to use this solution is that our definition of an environment is actually taken from the *Formal Metatheory* library (discussed in Section 2.1). Our subject reduction proof is large enough as it is, and the more infrastructure we can re-use, the better. Replacing the definition of an environment would involve considerable refactoring of the *Formal Metatheory* library. One complicating factor is that the *Formal Metatheory* library abstracts over the "type of types" (the Coq datatype that is used to model types in the object language). This is useful, but if we want to keep the environment sorted, we cannot abstract over an arbitrary type, but require that the type comes with an ordering relation. Thus, not only would the implementation of the library have to be modified, its interface would also have to change.

We must therefore explicitly allow for exchange in the type system. The traditional way is to include the exchange lemma as an axiom[4]:

$$\frac{\Gamma, \Delta, \Theta \vdash e : \tau}{\Gamma, \Theta, \Delta \vdash e : \tau}\textsc{Exch}$$

The downside of this approach is that the inversion lemmas for the typing relation become more difficult to state. For example, in the original type system we could prove the following inversion lemma:

> **Lemma** (Inversion lemma for application). If $\Gamma \vdash f\ e : \tau$, then there exists $\Delta, \Theta$ such that $\Gamma = \Delta, \Theta$, and there exists $\sigma$ such that $\Delta \vdash f : \sigma \to \tau$ and $\Theta \vdash e : \sigma$.

In the modified type system, however, this lemma no longer holds. Instead, we would have to allow for an application of the exchange rule, which makes the inversion lemma harder to state. This problem is amplified by the presence of other substructural rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma, x : \sigma \vdash e : \tau}\textsc{Weak} \qquad \frac{\Gamma, y : \sigma, z : \sigma \vdash e : \tau}{\Gamma, x : \sigma \vdash e[x/z, x/y] : \tau}\textsc{Contr}$$

With these two rules, the inversion lemma for application becomes very difficult to state indeed. Fortunately, for an affine (as opposed to linear) substructural type system such as ours, weakening is unrestricted so that rule

[4] It is often presented as

$$\frac{\Gamma, \Delta \vdash e : \tau}{\Delta, \Gamma \vdash e : \tau}\textsc{Exch}'$$

but that rule is not strong enough. In particular, we cannot show Exch from Exch′.

WEAK can easily be integrated into the typing rule for variables. We do however need to control contraction (only unique variables can be used more than once), and it is not so obvious how to integrate CONTR into the other rules.

The solution we adopt is the one described in [13], where it is attributed to [5]. We define a generic context splitting operation, denoted $E = E_1 \circ E_2$, as follows:

$$\frac{}{\emptyset = \emptyset \circ \emptyset}\textsc{Split-Empty} \qquad \frac{E = E_1 \circ E_2}{E, x : t = E_1, x : t \circ E_2}\textsc{Split-Left}$$

$$\frac{E = E_1 \circ E_2 \qquad \text{non-unique } t}{E, x : t = E_1, x : t \circ E_2, x : t}\textsc{Split-Both} \qquad \frac{E = E_1 \circ E_2}{E, x : t = E_1 \circ E_2, x : t}\textsc{Split-Right}$$

We can use the context splitting operation in the rule for application as follows:

$$\frac{\Gamma \vdash f : \tau \to \sigma \qquad \Delta \vdash e : \tau}{\Gamma \circ \Delta \vdash f\ e : \sigma}\textsc{App}'$$

With this rule, lemma *Exchange* becomes admissible because we can prove an auxiliary result that if $E = E_1 \circ E_2$ then $E = E_2 \circ E_1$. This approach is attractive for two reasons. First, the inversion lemma is straightforward to state and prove. Second, we can reason about context splitting as a separate notion, and we will do so extensively (Section 5.10). This means that in those proofs where we need to reason about reordering of the environment (in particular lemmas *preservation_commute* and *preservation_assoc*, Section 7), this reasoning is explicit and usually done in separate lemmas.

## 2.3 Boolean expressions

In our type system, we allow for arbitrary boolean expressions as uniqueness attributes: $t^\bullet, t^\times, t^u, t^{u \vee v}, t^{u \wedge v}$ and $t^{\neg u}$ are all valid types. Moreover, we we want to identify "equivalent" boolean expressions: $t^{u \vee v}$ and $t^{v \vee u}$ are the same type. In other words, we want to identify uniqueness attributes (boolean expressions) that are equivalent under the usual set of axioms (Huntington's Postulates; see Appendix A).

Perhaps the most obvious solution is to quotient boolean expressions by Huntington's Postulates, and formally regard uniqueness attributes as equivalence classes of boolean expressions rather than boolean expressions. Since the equivalence class $[u \vee v]$ and $[v \vee u]$ are the same class (since both expressions are equivalent), the types $t^{[u \vee v]}$ and $t^{[v \vee u]}$ are then also identified.

Unfortunately, this solution is difficult to adopt for two reasons. First, since the equivalence class of a boolean expression is infinite, we would need to use coinduction to define the classes—not difficult conceptually, but technically awkward nevertheless. The other complication is that in our type system, and hence in the formalization, we do not distinguish between types and attributes (this is a key contribution of the paper). An attributed type $t^u$ is syntactic sugar for the application of a special type constant `Attr` to two arguments (`Attr` $t\ u$); a kind system weeds out ill-formed types. This approach does not combine well with treating uniqueness attributes as equivalence classes.

Instead, we explicitly allow to replace a type by an equivalent type as a non-syntax directed rule:

$$\frac{\Gamma \vdash e : \tau|_{fv} \qquad \tau \approx \sigma}{\Gamma \vdash : e : \sigma|_{fv}}\textsc{Equiv}$$

As it turns out, adding this lemma does not make the inversion lemmas more difficult to state (we prove the inversion lemmas in Section 6.6; see also Section 2.2). Moreover, adding this rule is sufficient to be able to replace a type anywhere in a typing derivation[5]; in particular, it is sufficient to be able to replace a type in an environment (lemma *typ_equiv_env*, Section 6.5). We will discuss the type equivalence relation proper in Section 3.2.

[5]This is not *quite* true; in the typing rule for variables, we must be careful to allow for a different (but equivalent) attribute in $E$ and $fv$.

# 3 Inversion

As we saw in the previous section, adding additional typing rules makes forward reasoning easier, but backward reasoning more difficult. For example, if we add a contraction rule to the type system, it becomes trivial to prove $\Gamma, x : \sigma, y : \sigma \vdash e : \tau$ from $\Gamma, z : \sigma \vdash [z/x, z/y]e : \tau$ (forward reasoning), but the inversion lemma for application becomes more difficult to state (backward reasoning). Generally, we want to make the definition of the type system permissive enough to facilitate forward reasoning, but not too permissive to complicate backward reasoning. We already saw one example of this: rather than adding a separate contraction rule, it is better to integrate contraction into the other rules (by introduction a generic context splitting operation; see Section 2.2). In this section, we will see a number of other examples of this tension between forward and backward reasoning.

## 3.1 Domain subtraction

In the definition of the type system we make use of a domain subtraction operation, denoted $\triangleright_x\, fv$, which removes $x$ from the domain of $fv$. In this section we discuss how we should define this operation. In particular: if $x$ occurs more than once in the domain of $fv$, should domain subtraction remove all of them, or only the first? Using an example, we will see that we will need to choose the latter option to be able to use backwards reasoning.

We will need a few definitions first. An environment is well-formed if it is *ok* and well-kinded: that is, if every variable occurs at most once in its domain and all the types in the codomain of the environment have the same kind. Two environments are equivalent, denoted $\Gamma \cong_k \Gamma'$, if they are both well-formed and map the same variables to the same types (the subscript $k$ denotes the kind of the types in the codomain of the environments; these definitions are given formally in Section 4).

An important lemma is that if $\Gamma \vdash e : \tau|_{fv}$, $\Gamma \cong_* \Gamma'$ and $fv \cong_{\mathcal{U}} fv'$, then $\Gamma' \vdash e : \tau|_{fv'}$ (Lemma *env_equiv_typing*, Section 6.5). This lemma is important because it allows to change the order of the assumptions in the environment (Lemma *exchange*) or replace a type by an equivalent type in an environment (Lemma *typ_equiv_env*). The proof of the lemma is by induction on the typing relation.

Consider the case for the rule for abstraction. We know that $\Gamma \cong_* \Gamma'$ and $fv' \cong_{\mathcal{U}} fv'_0$. The induction hypothesis gives us[6]

$$(\Gamma, x : a \cong_* \Gamma', x : a) \to (fv', x : v \cong_{\mathcal{U}} fv) \to (\Gamma', x : a \vdash e^x : b|_{fv})$$

and we have to show that

$$\Gamma' \vdash \lambda \cdot e : a \xrightarrow{\bigvee fv'} b|_{fv'_0}$$

Replacing the attribute on the arrow by an equivalent one gives $\Gamma' \vdash \lambda \cdot e : a \xrightarrow{\bigvee fv'_0} b|_{fv'_0}$, at which point we can apply the typing rule for abstraction. Remains to show that

$$\Gamma', x : a \vdash e^x : b|_{fv}$$

where we know that $fv'_0 = \triangleright_x\, fv$ and $x \notin \Gamma \cup fv'_0$. We can use the induction hypothesis to complete the proof, but only if we can prove its two premises. The first one is straightforward, but the second is more tricky:

$$fv', x : v \cong_{\mathcal{U}} fv$$

To be able to show this equivalence, we need to be able to show that $fv$ is well-formed; in particular, we need to be able to show that it is *ok* (every variable occurs at most once in its domain). Since $\triangleright_x\, fv = fv'_0$, we know that $\triangleright_x\, fv$ is *ok* because $fv'_0 \cong_U fv'$, and we know that $x \notin \triangleright_x\, fv$ because $x \notin fv'_0$. However, it now depends on the definition of domain subtraction ($\triangleright$) whether we can show that $fv$ is *ok*.

[6] This is a minor simplification of the proof; in the actual proof, we need to distinguish between the case where the bound variable of the abstraction is used in the body (the case which is shown here), and the case where it is not used. We do not discuss the second (easier) case.

If $\triangleright_x fv$ removes *all* occurrences of $x$ from $fv$, then we will be unable to complete the proof: even if $\triangleright_x fv$ is ok, that does not allow us to conclude anything about the well-formedness of $fv$. On the other hand, if $\triangleright_x fv$ only removes the *first* occurrence of $x$, then $fv$ can contain at most one more assumption about $x$ than $\triangleright_x fv$; if additionally we know that $x \notin \triangleright_x fv$, then we can conclude that $fv$ must be ok.

Hence, we conclude that domain subtraction must remove the first occurrence of a variable only. This makes forward reasoning slightly more difficult, since where before we could prove a lemma that $x \notin \triangleright_x fv$, now that only holds if $fv$ is *ok*. Fortunately, we always require environments to be well-formed, so this is no problem in practice. On the other hand, backwards reasoning (proving that $fv$ is *ok* given that $\triangleright_x fv$ is *ok* and $x \notin \triangleright_x fv$) is impossible if domain subtraction removes all variables from the domain of an environment.

## 3.2 Type equivalence

Huntington's Postulates give us an equivalence relation $\approx_{\mathrm{B}}$ on types. For example, we have that $u \vee v \approx_{\mathrm{B}} v \vee u$ (commutativity of disjunction) or $u \wedge \bullet \approx_{\mathrm{B}} u$ (identity element for conjunction). We want to extend this equivalence relation to a more general equivalence relation ($\approx_{\mathrm{T}}$), which is effectively ($\approx_{\mathrm{B}}$) extended with a closure rule for type application:

$$\frac{t \approx_{\mathrm{B}} t'}{t \approx_{\mathrm{T}} t'} \qquad \frac{t \approx_{\mathrm{T}} t' \qquad s \approx_{\mathrm{T}} s'}{t\ s \approx_{\mathrm{T}} t'\ s'}$$

This allows us to derive that $t^{u \vee v} \approx_{\mathrm{T}} t^{v \vee u}$, for example, or that if $a \approx_{\mathrm{T}} a'$, then $a \xrightarrow{u} b \approx_{\mathrm{T}} a' \xrightarrow{u} b$ (recall that $a \xrightarrow{u} b$ is syntactic sugar for $\mathtt{Attr}\ (\mathtt{Arr}\ a\ b)\ u$). However, we also occasionally need to reason backwards on the typing equivalence relation: if we know that $t^u \approx_{\mathrm{T}} t^v$, we would like to be able prove that $u \approx_{\mathrm{T}} v$.

It would seem that the easiest way to prove that would be to prove the following inversion lemma: if $t\ s \approx_{\mathrm{T}} t'\ s'$, then $t \approx_{\mathrm{T}} t'$ and $s \approx_{\mathrm{T}} s'$. Unfortunately, that lemma does not hold. Recall that we do not distinguish between types and attributes in our type system. That is, the "attribute" $u \vee v$ is a type (which happens to have kind $\mathcal{U}$). Moreover, $u \vee v$ is really syntactic sugar for the application of a special type constant $\mathtt{Or}$ of kind $\mathcal{U} \to \mathcal{U} \to \mathcal{U}$ to two arguments ($\mathtt{Or}\ u\ v$). By Huntington's Postulates we have that $u \vee v \approx_{\mathrm{T}} v \vee u$, or desugared: $\mathtt{Or}\ u\ v \approx_{\mathrm{T}} \mathtt{Or}\ v\ u$. If the inversion lemma were true, we would thus be able to conclude that $u \approx_{\mathrm{T}} v$, for any $u$ and $v$.

So, to make backwards reasoning possible, we need to redefine $\approx_{\mathrm{T}}$ slightly:

$$\frac{t \approx_{\mathrm{B}} t' \qquad t : \mathcal{U}, t' : \mathcal{U}}{t \approx_{\mathrm{T}} t'} \qquad \frac{t \approx_{\mathrm{T}} t' \qquad s \approx_{\mathrm{T}} s' \qquad \neg(t\ s : \mathcal{U})}{t\ s \approx_{\mathrm{T}} t'\ s'}$$

(In addition, we need to introduce reflexivity, commutativity and transitivity rules; they were previously implied by ($\approx_{\mathrm{B}}$)). We can now prove the following inversion lemma: if $t\ s \approx_{\mathrm{T}} t'\ s'$, and $t\ s$ does not have kind $\mathcal{U}$, then $t \approx_{\mathrm{T}} t'$ and $s \approx_{\mathrm{T}} s'$. Restricting the closure rule to types of kind other than $\mathcal{U}$ is not strictly necessary to prove this inversion lemma, but makes proving other lemmas easier (for example, Lemma *typ_equiv_BA_equiv*, Section 5.11) without reducing the equivalence relation: closure for types of kind $\mathcal{U}$ is already implied by Huntington's Postulates.

This modification to the type equivalence relation has an additional benefit. Recall the following rule for context splitting:

$$\frac{E = E_1 \circ E_2 \qquad \text{non-unique}\ t}{E, x : t = E_1, x : t \circ E_2, x : t}\ \textsc{Split-Both}$$

Since the context splitting operation is applied both to typing environments ($\Gamma$) and the lists of free variables ($fv$), we give the following two axioms to prove "non-unique":

$$\frac{u \approx_{\mathrm{T}} \times}{\text{non-unique}(t^u)}\ \mathrm{NU}_* \qquad \frac{u \approx_{\mathrm{T}} \times}{\text{non-unique}(u)}\ \mathrm{NU}_{\mathcal{U}}$$

Now consider proving the following lemma: if $a \xrightarrow{u} b$ is non-unique, then $u \approx_{\mathrm{T}} \times$. The proof proceeds by inversion on non-unique$(a \xrightarrow{u} b)$. The case for rule $\mathrm{NU}_*$ is trivial, but how can we dismiss the case for rule

$NU_{\mathcal{U}}$? Without the kind requirements added to the type equivalence relation, we would have to show that it is impossible that $a \xrightarrow{u} b$ is equivalent to $\times$ by Huntington's Postulates; not an easy proof![7]

### 3.3 Evaluation contexts

The operational semantics we use is the call-by-need semantics by Maraist *et al.* [11]. In this semantics, the definition of evaluation depends on the notion of an *evaluation context*, which is essentially a term with a hole in it (the difference between an evaluation context and the more general notion of a "context" [2] is that in an evaluation context, we restrict where the hole can appear in the term). There are various ways in which we can formalize an evaluation context in Coq. In simple cases, we can follow informal practice and define a context $E$ inductively, followed by a definition of plugging a term $M$ into the hole in the context $E[M]$. This is the approach taken in [4], for instance, but it does not apply here because we need the definition of $E[M]$ *when defining* $E[]$.

Another approach [8] is to define a context as an ordinary function on terms, and then (inductively) define which functions on terms can be regarded as evaluation contexts. This is an attractive and elegant approach, but does not work so well in the locally-nameless approach: since some evaluation contexts place a term within the scope of a binder but others do not, we must distinguish between *binding* contexts which have the property that if $t^x$ is a term for some fresh $x$, then $E[x]$ is also a term, and *regular* contexts (which do not have this property).

For example, consider the proof that reduction is regular: if $e \mapsto e'$, then both $e$ and $e'$ are locally closed[8]. The proof is by induction on $e \mapsto e'$. In the case for the closure rule, we know that $E[e]$ and $E[e']$ are locally closed, and we have to show that $e$ and $e'$ are locally closed. However, we may or may not be able to show this (depending on whether $E$ is a regular or a binding context). Thus, we need to distinguish the "closing" evaluation contexts from the others, at which point the elegance of the approach starts disappearing. We now need two closure rules (one for closing and one for regular contexts) and we have introduced a new characterization of evaluation contexts that we will need to reason about.

To avoid having to reason about closing contexts and regular contexts, we instead inline the definition of the evaluation contexts into the definition of the reduction relation. This gives only one more rule than when giving a closure rule for regular contexts and a closure rule for closing contexts, and moreover, the resulting closure rules correspond to intuitive notions about the semantics.

We still need to define the notion of an evaluation context, because the reduction relation depends on it in the other rules too. As mentioned before, we cannot define the notion of a context separately from plugging a term into the hole. The solution we adopt is to define $E$ as a binary relation between a term and a free variable, where $E\ t\ x$ should be read as $t$ evaluates $x$ (there is an evaluation context $E$ such that $t = E[x]$). This gives good inversion principles (suitable for backwards reasoning) and combines well with the locally nameless approach.

## Acknowledgements

Arthur Charguéraud has been extremely helpful in getting started with this proof and the use of his *Formal Metatheory* library. Many thanks! In addition, I would not have been able to complete this proof without the generous assistance of the people on the Coq mailing list, in particular (in alphabetical order): Adam Chlipala, Adam Megacz, Arnaud Spiwack, Benjamin Werner, Brian Aydemir, Carlos Simpson, Damien Pous, Eduardo Gimenez, Frédéric Besson, Frédéric Blanqui, Gyesik Lee, James McKinna, Jean Duprat, Jean-François Monin, Jevgenijs Sallinenes, Julien Forest, Julien Narboux, Lionel Elie Mamane, Matthieu Sozeau, Pierre Castéran, Pierre Courtieu, Pierre Letouzey, Pietro Di Gianantonio, Randy Pollack, Santiago Zanella, Stéphane Glondu, Vincent Aravantinos, Yevgeniy Makarov and Yves Bertot.

[7] If the proof seems trivial, perhaps the reader would like to attempt an even easier proof: prove that it is impossible to construct a proof using Huntington's postulates that "true" is equivalent to "false"—without using an interpretation function! (It is not clear how to define an interpretation function for the broader class of types, as opposed to just the types of kind $\mathcal{U}$.)

[8] We mentioned before that in the locally nameless approach to formal metatheory we distinguish between bound variables, represented by De Bruijn indices, and free variables, represented by ordinary names. A term is locally closed if it does not contain any "unbound bound variables"; that is, if it does not contain any De Bruijn indices without a corresponding binder.

# 4 Definitions

## 4.1 Types

A type is either a type constant or the application of one type to another.

```
Inductive typ : Set :=
  (** Type application *)
  | typ_app : typ → typ → typ
  (** Type constants *)
  | ARR : typ
  | ATTR : typ
  | UN : typ
  | NU : typ
  | OR : typ
  | AND : typ
  | NOT : typ.
```

For convenience, we define a number of functions to denote commonly used types, and some custom notation for attributed types.

```
Definition bi_app (f a b : typ) : typ := typ_app (typ_app f a) b.

Definition arr (a b : typ) : typ := bi_app ARR a b.
Definition attr (t u : typ) : typ := bi_app ATTR t u.
Definition or (u v : typ) : typ := bi_app OR u v.
Definition and (u v : typ) : typ := bi_app AND u v.
Definition not (u : typ) : typ := typ_app NOT u.

Notation "t ’ u" := (attr t u) (at level 60).
Notation "a 〈 u 〉 b" := ((arr a b) ’ u) (at level 68).
```

(A subset of the) language of types forms a boolean algebra.

```
Module TypeAsBooleanAlgebra <: BooleanAlgebraTerm.

Definition trm := typ.
Definition true := UN.
Definition false := NU.
Definition or := or.
Definition and := and.
Definition not := not.

End TypeAsBooleanAlgebra.

Module BA := BooleanAlgebra TypeAsBooleanAlgebra.
```

## 4.2 Kinding relation

The definition of kinds.

```
Inductive kind : Set :=
  | kind_T : kind
  | kind_U : kind
  | kind_star : kind
  | kind_arr : kind → kind → kind.
```

Kinding relation.

```
Inductive kinding : typ → kind → Prop :=
  | kinding_app : ∀ t1 t2 k1 k2,
      kinding t1 (kind_arr k1 k2) →
      kinding t2 k1 →
      kinding (typ_app t1 t2) k2
  | kinding_ARR : kinding ARR (kind_arr kind_star (kind_arr kind_star kind_T))
  | kinding_ATTR : kinding ATTR (kind_arr kind_T (kind_arr kind_U kind_star))
  | kinding_UN : kinding UN kind_U
  | kinding_NU : kinding NU kind_U
  | kinding_OR : kinding OR (kind_arr kind_U (kind_arr kind_U kind_U))
  | kinding_AND : kinding AND (kind_arr kind_U (kind_arr kind_U kind_U))
  | kinding_NOT : kinding NOT (kind_arr kind_U kind_U).
```

```
Hint Constructors kinding.
```

Equivalence between types

```
Inductive typ_equiv : typ → typ → Prop :=
   (** The type equivalence includes the boolean equivalence relation *)
  | typ_equiv_attr : ∀ u v,
      kinding u kind_U →
      kinding v kind_U →
      BA.equiv u v →
      typ_equiv u v
   (** Closure (does not apply to types of kind U) *)
  | typ_equiv_app : ∀ s t s' t',
      ¬ kinding (typ_app s t) kind_U →
      typ_equiv s s' →
      typ_equiv t t' →
      typ_equiv (typ_app s t) (typ_app s' t')
   (** Structural rules *)
  | typ_equiv_refl : ∀ t,
      typ_equiv t t
  | typ_equiv_sym : ∀ t s,
      typ_equiv t s → typ_equiv s t
  | typ_equiv_trans : ∀ t s r,
      typ_equiv t s → typ_equiv s r → typ_equiv t r.
```

```
Hint Constructors typ_equiv.
```

## 4.3 Environment

The definition of an environment comes from the Formal Metatheory library; we just need to instantiate it with our definition of a type.

```
Definition env : Set := Env.env typ.
```

An environment is well-formed if it is *ok* and well-kinded.

```
Definition env_kind (k : kind) : env → Prop :=
  env_prop (fun t ⇒ kinding t k).
```

```
Definition env_wf (E : env) (k : kind) : Prop :=
  ok E ∧ env_kind k E.
```

Two environments are considered equivalent if they both bind the same variables to equivalent types, and both are wellformed. For clarity, we introduce a special syntax to denote equivalence.

Definition *env_equiv* (*E1 E2* : *env*) (*k* : *kind*) : Prop :=
  *env_wf E1 k* ∧ *env_wf E2 k* ∧
  (∀ *x t*, *binds x t E1* → ∃ *t'*, *binds x t' E2* ∧ *typ_equiv t t'*) ∧
  (∀ *x t*, *binds x t E2* → ∃ *t'*, *binds x t' E1* ∧ *typ_equiv t t'*).

Notation "*E1* ≅ *E2*" := (*env_equiv E1 E2*) (*at level* 70).

The definition of the context split operation, as explained in the introduction. The context split is used both to split *E*, the typing environment and *fvars*, the list of free variables and their uniqueness attributes in the typing rules. For this reason, we introduce a separate "non_unique" property of types, which applies to types of kind $*$ when they have a non-unique attribute, and to attributes (types of kind $\mathcal{U}$) when they are non-unique themselves.

*Reserved Notation* "'*split_context*' *E* '*as*' ( *E1* ; *E2* )".

Inductive *non_unique* : *typ* → Prop :=
  | *NU_star* : ∀ *t u*,
      *typ_equiv u NU* → *non_unique* (*t* ' *u*)
  | *NU_U* : ∀ *u*,
      *typ_equiv u NU* → *non_unique u*.

Inductive *context_split* : *env* → *env* → *env* → Prop :=
  | *split_empty* :
      *split_context empty* as (*empty* ; *empty*)
  | *split_both* : ∀ *E E1 E2 x t*, *split_context E* as (*E1* ; *E2*) → *non_unique t* →
      *split_context* (*E* & *x* ¬ *t*) as (*E1* & *x* ¬ *t*; *E2* & *x* ¬ *t*)
  | *split_left* : ∀ *E E1 E2 x t*, *split_context E* as (*E1* ; *E2*) →
      *split_context* (*E* & *x* ¬ *t*) as (*E1* & *x* ¬ *t* ; *E2*)
  | *split_right* : ∀ *E E1 E2 x t*, *split_context E* as (*E1* ; *E2*) →
      *split_context* (*E* & *x* ¬ *t*) as (*E1* ; *E2* & *x* ¬ *t*)
  *where*
    "'*split_context*' *E* '*as*' ( *E1* ; *E2* )" := (*context_split E E1 E2*).

Hint *Constructors non_unique context_split*.

## 4.4 Operations on the typing context

Disjunction of all types on the range of the environment

Fixpoint *rng* (*E* : *env*) : *typ* :=
  match *E* with
  | *nil* ⇒ *NU*
  | (*x*, *u*) :: *tail* ⇒ *or u* (*rng tail*)
  end.

Remove the first occurrence of *x* in *E*

Fixpoint *dsub* (*x* : *var*) (*E* : *env*) {*struct E*} : *env* :=
  match *E* with
  | *nil* ⇒ *nil*
  | (*y*, *t*) :: *tail* ⇒ if *x* == *y* then *tail* else (*y*, *t*) :: *dsub x tail*
  end.

Call *dsub* for every *x* in *xs*.

Fixpoint *dsub_list* (*xs* : *list var*) (*E* : *env*) : *env* :=
  match *xs* with

| *nil* ⇒ *E*
| *x* :: *xs'* ⇒ *dsub_list xs'* (*dsub x E*)
end.

Variation on *dsub_list* working on sets *xs* rather than lists.
Definition *dsub_vars* (*xs* : *vars*) (*E* : *env*) : *env* := *dsub_list* (*S.elements xs*) *E*.

## 4.5 Typing relation

The rule for variables *typing_var* is subtle in two ways: since it only requires that *binds x* (*t* ' *u*) *E*, and therefore allows for other assumptions in *E*, it implicitly allows weakening on *E*. However, it is much more strict on *fvars* (the only assumption in *fvars* must be the assumption $x : u$; hence, no weakening is allowed on *fvars*). This is important, because while additional assumptions in *E* cannot affect the type of a term, additional assumptions in *fvars* can (by unnecessarily forcing an abstraction to be unique). The typing rule for abstraction uses the cofinite quantification discussed in the introduction.

*Reserved Notation* "*E* ⊢ *t* : *T* | *fvars*" (*at level* 69).

Inductive *typing* : *env* → *trm* → *typ* → *env* → Prop :=
| *typing_var* : ∀ *E x t u v*,
*env_wf E kind_star* →
*binds x* (*t* ' *u*) *E* →
*typ_equiv u v* →
*E* ⊢ (*trm_fvar x*) : *t* ' *u* | *x* ¬ *v*
| *typing_abs* : ∀ *L E a b e fvars'*,
(∀ *x fvars*, *x* \notin *L* → *fvars'* = *dsub x fvars* →
(*E* & *x* ¬ *a*) ⊢ *e* ^ *x* : *b* | *fvars*) →
*E* ⊢ (*trm_abs e*) : *a* ⟨ *rng fvars'* ⟩ *b* | *fvars'*
| *typing_app* : ∀ *E E1 E2 fvars fvars1 fvars2 e1 e2 a b u*,
*E1* ⊢ *e1* : *a* ⟨ *u* ⟩ *b* | *fvars1* →
*E2* ⊢ *e2* : *a* | *fvars2* →
*split_context E* as (*E1* ; *E2*) → *env_wf E kind_star* →
*split_context fvars* as (*fvars1* ; *fvars2*) → *env_wf fvars kind_U* →
*E* ⊢ (*trm_app e1 e2*) : *b* | *fvars*
| *typing_equiv* : ∀ *E e a b fvars*,
*E* ⊢ *e* : *a* | *fvars* →
*typ_equiv a b* →
*E* ⊢ *e* : *b* | *fvars*
*where* "*E* ⊢ *t* : *T* | *fvars*" := (*typing E t T fvars*).

Hint *Constructors typing*.

## 4.6 Semantics

We treat "let $x = y$ in $z$" as syntactic sugar for $(\lambda x \cdot z)\, y$.
Notation "'*lt*' *x* '*in*' *y*" := (*trm_app* (*trm_abs y*) *x*) (*at level* 70).

Definition of *answer*, *eval* and *red* as in [11]; again, we're using cofinite quantification.

Inductive *answer* : *trm* → Prop :=
| *answer_abs* : ∀ *M*, *term* (*trm_abs M*) →
*answer* (*trm_abs M*)
| *answer_let* : ∀ *L M A*, *term* (*lt M* in *A*) →
(∀ *x*, *x* \notin *L* → *answer* (*A* ^ *x*)) →

*answer* (*lt M* in *A*).

Definition of an evaluation context

Inductive *evals* : *trm* → *var* → Prop :=
| *evals_hole* : ∀ *x*,
*evals* (*trm_fvar x*) *x*
| *evals_app* : ∀ *x E M*, *evals E x* →
*evals* (*trm_app E M*) *x*
| *evals_let* : ∀ *L x E M*,
(∀ *y*, *y* \notin *L* → *evals* (*E* ^ *y*) *x*) →
*evals* (*lt M* in *E*) *x*
| *evals_dem* : ∀ *L x E M*, *evals E x* →
(∀ *y*, *y* \notin *L* → *evals* (*M* ^ *y*) *y*) →
*evals* (*lt E* in *M*) *x*.

Hint *Constructors evals*.

As mentioned before, the reduction relation we use is the standard reduction from [11], except that *red_value* is defined as in [11, Section "*On types and logic*", p. 38] (adapted for standard reduction). None of these rules adjust any of the bound variables (which are after all De Bruijn variables); this is justified by lemma *red_regular*, given in Section 5.7, which states that the reduction relation is defined for *locally closed* terms only (that is, they may contain free variables, but no unbound De Bruijn indices).

Inductive *red* : *trm* → *trm* → Prop :=
(** *Standard reduction rules* *)
| *red_value* : ∀ *L M N*, *term* (*lt* (*trm_abs M*) in *N*) →
(∀ *x*, *x* \notin *L* → *evals* (*N* ^ *x*) *x*) →
*red* (*lt* (*trm_abs M*) in *N*) (*N* ^^ *trm_abs M*)
| *red_commute* : ∀ *L M A N*, *term* (*trm_app* (*lt M* in *A*) *N*) →
(∀ *x*, *x* \notin *L* → *answer* (*A* ^ *x*)) →
*red* (*trm_app* (*lt M* in *A*) *N*) (*lt M* in *trm_app A N*)
| *red_assoc* : ∀ *L M A N*, *term* (*lt* (*lt M* in *A*) in *N*) →
(∀ *x*, *x* \notin *L* → *answer* (*A* ^ *x*)) →
(∀ *x*, *x* \notin *L* → *evals* (*N* ^ *x*) *x*) →
*red* (*lt* (*lt M* in *A*) in *N*) (*lt M* in *lt A* in *N*)
(** *Compatible closure* *)
| *red_closure_app* : ∀ *E E' M*, *term* (*trm_app E M*) →
*red E E'* →
*red* (*trm_app E M*) (*trm_app E' M*)
| *red_closure_let* : ∀ *L E E' M*, *term* (*lt M* in *E*) →
(∀ *x*, *x* \notin *L* → *red* (*E* ^ *x*) (*E'* ^ *x*)) →
*red* (*lt M* in *E*) (*lt M* in *E'*)
| *red_closure_dem* : ∀ *L E0 E0' E1*, *term* (*lt E0* in *E1*) →
*red E0 E0'* →
(∀ *x*, *x* \notin *L* → *evals* (*E1* ^ *x*) *x*) →
*red* (*lt E0* in *E1*) (*lt E0'* in *E1*).

Hint *Constructors answer red*.

# 5 Preliminaries

## 5.1 Some additional lemmas about *ok* and *binds*

Every variable occurs at most once.
Lemma *ok_mid* : ∀ (*E2 E1* : *env*) *x t*,
  *ok* (*E1* & *x* ¬ *t* & *E2*) → *x* # *E1* ∧ *x* # *E2*.
By induction on *E2*.

If two environments are both *ok* and their domains are disjoint, then their concatenation is also *ok*.
Lemma *ok_concat* : ∀ (*E2 E1* : *env*),
  *ok E1* → *ok E2* →
  (∀ *x*, *x* \in *dom E1* → *x* \notin *dom E2*) →
  (∀ *x*, *x* \in *dom E2* → *x* \notin *dom E1*) →
  *ok* (*E1* & *E2*).
By induction on *E2*.

If the concatenation of two environments is *ok*, then their domains must be disjoint.
Lemma *ok_concat_inv_2* : ∀ (*E2 E1* : *env*),
  *ok* (*E1* & *E2*) →
  (∀ *x*, *x* \in *dom E1* → *x* \notin *dom E2*) ∧
  (∀ *x*, *x* \in *dom E2* → *x* \notin *dom E1*).
By induction on *E2*.

We can change the order of the assumptions in an environment without affecting *ok*.
Lemma *ok_exch* : ∀ (*E1 E2* : *env*),
  *ok* (*E1* & *E2*) → *ok* (*E2* & *E1*).
By induction on *E1*.

Generalization of *ok_exch*.
Lemma *ok_exch_3* : ∀ (*E1 E2 E3* : *env*),
  *ok* (*E1* & *E2* & *E3*) → *ok* (*E1* & *E3* & *E2*).
Follows from *ok_concat_inv_2* and *ok_exch*.

If an environment binds a variable *x*, then *x* must be in the domain of the environment.
Lemma *binds_in_dom* : ∀ (*A* : Set) *x* (*T* : *A*) *E*,
  *binds x T E* → *x* \in *dom E*.
By induction on *E*.

Inverse of *binds_in_dom*: if a variable *x* is in the domain of an environment, then the environment must bind *x*.
Lemma *in_dom_binds* : ∀ (*E* : *env*) *x*,
  *x* \in *dom E* → ∃ *t*, *binds x t E*.
By induction on *E*.

Binds is unaffected by the order of the assumptions in an environment.
Lemma *binds_exch* : ∀ (*E1 E2* : *env*) *x t*, *ok* (*E1* & *E2*) →
  *binds x t* (*E1* & *E2*) →
  *binds x t* (*E2* & *E1*).
Follows from *ok_concat_inv_2*.

Generalization of *binds_exch*.
Lemma *binds_exch_3* : ∀ (*E1 E2 E3* : *env*) *x t*, *ok* (*E1* & *E2* & *E3*) →
  *binds x t* (*E1* & *E2* & *E3*) →
  *binds x t* (*E1* & *E3* & *E2*).
Trivial.

A variable can only be bound to one type.
Lemma *binds_head_inv* : ∀ (*E* : *env*) *x a b*,
  *binds x a* (*E* & *x* ¬ *b*) → *a* = *b*.
Trivial.

## 5.2 Renaming Lemmas

All these renaming lemmas are proven in the same way. We first prove a substitution lemma which states that the names of the free variables do not matter, and then we prove the renaming lemma using the substitution lemma and the fact that *t* ^^ *u* = [$x \rightsquigarrow u$] *t* ^ *x*, as long as *x* \notin *fv t*.

If *e* is an answer, then it will still be an answer when we rename any of its free variables.
Lemma *subst_answer* : $\forall$ *e x y*,
*answer e* $\rightarrow$ *answer* ([$x \rightsquigarrow$ *trm_fvar y*] *e*).
By induction on *answer e*.

If *t* ^ *x* is an answer, then *t* ^ *y* will also be an answer for any *y*.
Lemma *answer_rename* : $\forall$ *x y t*,
*x* \notin *fv t* $\rightarrow$
*answer* (*t* ^ *x*) $\rightarrow$ *answer* (*t* ^ *y*).
Follows from *subst_answer*.

If *M* evaluates *x* (by the evaluation context relation defined previously) then if we rename *y* to *z* in *M*, *M* will still evaluate *x* if $x \neq y$, or *M* will evaluate *z* otherwise.
Lemma *subst_evals* : $\forall$ *M x y z*,
*evals M x* $\rightarrow$ *evals* ([$y \rightsquigarrow$ *trm_fvar z*] *M*) (if *x* == *y* then *z* else *x*).
By induction on *evals M x*.

If *M* ^ *x* evaluates *x*, then *M* ^ *y* will evaluate *y* for any *y*.
Lemma *evals_rename* : $\forall$ *M x y*,
*x* \notin *fv M* $\rightarrow$
*evals* (*M* ^ *x*) *x* $\rightarrow$ *evals* (*M* ^ *y*) *y*.
Follows from *subst_evals*.

Specialization of *subst_evals*, excluding the case that *x* = *y*.
Lemma *subst_evals_2* : $\forall$ *M x y z*, $x \neq y \rightarrow$
*evals M x* $\rightarrow$ *evals* ([$y \rightsquigarrow$ *trm_fvar z*] *M*) *x*.
Follows from *subst_evals*.

Generalization of *evals_rename*.
Lemma *evals_rename_2* : $\forall$ *M x y z*,
*x* \notin *fv M* $\rightarrow z \neq x \rightarrow$
*evals* (*M* ^ *x*) *z* $\rightarrow$ *evals* (*M* ^ *y*) *z*.
Follows from *subst_evals_2*.

If *e* reduces to *e'*, then if we rename a free variable by another in both terms the reduction relation will still hold.
Lemma *subst_red* : $\forall$ *e e' x y* ,
*red e e'* $\rightarrow$ *red* ([$x \rightsquigarrow$ *trm_fvar y*] *e*) ([$x \rightsquigarrow$ *trm_fvar y*] *e'*).
By induction on *red e e'*; uses *subst_evals_2*.

If *M* ^ *x* reduces to *N* ^ *x*, then *M* ^ *y* will reduce to *N* ^ *y* for any *y*.
Lemma *red_rename* : $\forall$ *x y M N*,
*x* \notin *fv M* $\rightarrow$ *x* \notin *fv N* $\rightarrow$
*red* (*M* ^ *x*) (*N* ^ *x*) $\rightarrow$ *red* (*M* ^ *y*) (*N* ^ *y*).
Follows trivially from *subst_read*.

## 5.3 Term opening

Auxiliary lemma used to prove *in_open*, below.
Lemma *in_open_aux* : $\forall$ *M x y k l*, $x \neq y \rightarrow$
*x* \in *fv* ({$k \rightsquigarrow$ *trm_fvar y*} *M*) $\rightarrow$ *x* \in *fv* ({$l \rightsquigarrow$ *trm_fvar y*} *M*).

By induction on *M*.

If *x* is free in $M \hat{} y$ and $y \neq x$, then *x* is free in *M*.
Lemma *in_open* : $\forall$ *M x y*,
  *x* \in *fv* ($M \hat{} y$) $\rightarrow y \neq x \rightarrow$ *x* \in *fv M*.
By induction on *M*; uses *in_open_aux*.

If *x* is free in *e*, then *x* will still be free when we substitute any bound variable in *e*.
Lemma *in_open_2* : $\forall$ *e e' k x*,
  *x* \in *fv e* $\rightarrow$ *x* \in *fv* ({$k \rightsquigarrow e'$} *e*).
By induction on *e*.

If *x* is not free in *t*, then if we replace a bound variable *k* by *y* (where $x \neq y$) in *t*, *x* will still not be free in *t*.
Lemma *open_rec_fv* : $\forall$ *t x y k*,
  *x* \notin *fv t* $\rightarrow x \neq y \rightarrow$ *x* \notin *fv* ({$k \rightsquigarrow$ *trm_fvar y*} *t*).
By induction on *t*.

If $t \hat{} x$ is locally-closed, then substituting for any bound variables larger than 0 in *t* has no effect.
Lemma *open_rec_term_open* : $\forall$ *t x*,
  *term* ($t \hat{} x$) $\rightarrow \forall$ *k t'*, $k \geq 1 \rightarrow t$ = {$k \rightsquigarrow t'$} *t*.
Trivial.

## 5.4 Domain subtraction

Subtracting an element *x* from the domain of an environment *fvars* has no effect when *x* wasn't in the domain of *fvars* to start with.
Lemma *dsub_not_in_dom* : $\forall$ (*fvars* : *env*) *x*, *x* # *fvars* $\rightarrow$
  *fvars* = *dsub x fvars*.
By induction on *fvars*.

$\triangleright_x$ removes *x* from a domain
Lemma *not_in_dom_dsub* : $\forall$ *fvars x*, *ok fvars* $\rightarrow$
  *x* # *dsub x fvars*.
By induction on *fvars*.

Removing *x* from *E* & *x* ¬ *t* gives *E*.
Lemma *dsub_head* : $\forall$ *E x t*, *dsub x* (*E* & *x* ¬ *t*) = *E*.
Trivial.

($\triangleright$) distributes over (++).
Lemma *dsub_app* : $\forall$ *E1 E2 x*, *ok* (*E1* ++ *E2*) $\rightarrow$
  *dsub x* (*E1* ++ *E2*) = *dsub x E1* ++ *dsub x E2*.
By induction on *E1*.

($\triangleright$) distributes over (&).
*Corollary dsub_concat* : $\forall$ *fvars1 fvars2 x*, *ok* (*fvars1* & *fvars2*) $\rightarrow$
  *dsub x* (*fvars1* & *fvars2*) = *dsub x fvars1* & *dsub x fvars2*.
Follows trivially from *dsub_app*.

If removing *x* from *fvars* is the empty environment, then *y* cannot be in the domain of *fvars*.
Lemma *not_in_dom_empty* : $\forall$ *fvars x y*,
  *dsub x fvars* = *empty* $\rightarrow x \neq y \rightarrow$ *y* \in *dom fvars* $\rightarrow$ *False*.
By case analysis on *fvars*.

If *E* binds *x* and $x \neq y$, then $(\triangleright_y E)$ binds *x*.
Lemma *binds_dsub* : $\forall$ *E x y T*,
  *binds x T E* $\rightarrow x \neq y \rightarrow$ *binds x T* (*dsub y E*).

By induction on $E$.

Inverse property of *binds_dsub_inv*.
Lemma *binds_dsub_inv* : $\forall$ *E x y T*,
  *binds x T* (*dsub y E*) $\rightarrow x \neq y \rightarrow$ *binds x T E*.
By induction on $E$.

If $x$ is in the domain of $E$ and $x \neq y$, then $x$ is in the domain of *dsub y E*.
Lemma *in_dom_dsub* : $\forall$ *E x y*,
  *x* \in *dom E* $\rightarrow x \neq y \rightarrow$ *x* \in *dom* (*dsub y E*).
By induction on $E$.

Inverse property of *in_dom_dsub*.
Lemma *in_dom_dsub_inv* : $\forall$ *E x y*,
  *x* \in *dom* (*dsub y E*) $\rightarrow$ *x* \in *dom E*.
By induction on $E$.

If $x$ is in the domain of $E$ and $\triangleright_y E$ is the empty environment, then $x$ must be $y$.
Lemma *in_dom_dsub_empty* : $\forall$ *E x y*,
  *x* \in *dom E* $\rightarrow$ *dsub y E = empty* $\rightarrow x = y$.
By induction on $E$.

If an environment is *ok*, it will still be *ok* if we remove a variable from its domain.
Lemma *ok_dsub* : $\forall$ *E x*,
  *ok E* $\rightarrow$ *ok* (*dsub x E*).
By induction on *ok E*.

If an environment is *ok*, it will still be *ok* if we add a single assumption about $x$ to the environment, provided that $x$ wasn't already in the domain of $E$.
Lemma *ok_dsub_inv* : $\forall$ *E x*,
  *ok* (*dsub x E*) $\rightarrow$ *x # dsub x E* $\rightarrow$ *ok E*.
By induction on $E$.

If removing $x$ from an environment yields the empty environment, then either the environment was empty to start with, or it is the singleton environment binding $x$.
Lemma *dsub_empty* : $\forall$ *E x*,
  *dsub x E = empty* $\rightarrow$ *E = empty* $\vee \exists$ *t*, *E = x* $\neg$ *t*.
By induction on $E$.

## 5.5 Kinding properties

An attributed type consists of a base type and an attribute.
Lemma *kinding_star_inv* : $\forall$ *t u*, *kinding* (*t ' u*) *kind_star* $\rightarrow$
  *kinding t kind_T* $\wedge$ *kinding u kind_U*.
By inversion on *kinding* (*t ' u*) *kind_star*.

The domain and codomain of functions must have kind $*$, and the attribute on the arrow must have kind $U$.
Lemma *kinding_fun_inv* : $\forall$ *a u b*, *kinding* (*a* $\langle$ *u* $\rangle$ *b*) *kind_star* $\rightarrow$
  *kinding a kind_star* $\wedge$ *kinding u kind_U* $\wedge$ *kinding b kind_star*.
By inversion on *kinding* (*a* $\langle$ *u* $\rangle$ *b*) *kind_star*.

Every type has at most one kind.
Lemma *kind_unique* : $\forall$ *t k1*, *kinding t k1* $\rightarrow$
  $\forall$ *k2*, *kinding t k2* $\rightarrow$ *k1 = k2*.
By induction on *kinding t k1*.

Equivalent types must have the same kind.

Lemma *typ_equiv_same_kind* : ∀ *t s*, *typ_equiv t s* →
  ∀ *k*, *kinding t k* ↔ *kinding s k*.
By induction on *typ_equiv t s*; uses *kind_unique*.

*or a b* has kind *U* if *a* and *b* have kind *u*.
Lemma *kinding_or* : ∀ *a b*, *kinding a kind_U* → *kinding b kind_U* →
  *kinding* (*or a b*) *kind_U*.
Trivial.

## 5.6 Well-formedness of environments

If an environment is well-formed, it must be *ok*.
Lemma *env_wf_ok* : ∀ *E k*, *env_wf E k* → *ok E*.
Trivial.

The empty environment is well-formed.
Lemma *env_wf_empty* : ∀ *k*, *env_wf empty k*.
Trivial.

The singleton environment is well-formed.
Lemma *env_wf_singleton* : ∀ *x t k*, *kinding t k* →
  *env_wf* (*x* ¬ *t*) *k*.
Trivial.

An environment can be extended with (*x* ¬ *t*) if *x* is not already in *E* and *t* has the right kind.
Lemma *env_wf_extend* : ∀ *E k x t*, *x* # *E* → *kinding t k* →
  *env_wf E k* → *env_wf* (*E* & *x* ¬ *t*) *k*.
Trivial.

The tail of a well-formed environment is also well-formed.
Lemma *env_wf_tail* : ∀ *E x t k*,
  *env_wf* (*E* & *x* ¬ *t*) *k* → *env_wf E k*.
Trivial.

Well-formedness of an environment is unaffected if we remove a variable.
Lemma *env_wf_dsub* : ∀ *E k x*,
  *env_wf E k* → *env_wf* (*dsub x E*) *k*.
Follows from *binds_dsub_inv*.

Well-formedness of an environment is unaffected when we add a fresh variable of the right kind.
Lemma *env_wf_dsub_inv* : ∀ *E k x*,
  *env_wf* (*dsub x E*) *k* →
  (∀ *t*, *binds x t E* → *kinding t k*) → *x* # *dsub x E* →
  *env_wf E k*.
Follows from *ok_dsub_inv* and *binds_dsub*.

Well-formed is unaffected if we replace a type by an equivalent one.
Lemma *env_wf_typ_equiv* : ∀ *E k x t s*, *typ_equiv t s* →
  *env_wf* (*E* & *x* ¬ *t*) *k* → *env_wf* (*E* & *x* ¬ *s*) *k*.
Follows from *typ_equiv_same_kind*.

Well-formedness of an environment is independent of the order of the assumptions.
Lemma *env_wf_exch* : ∀ *E1 E2 k*,
  *env_wf* (*E1* & *E2*) *k* → *env_wf* (*E2* & *E1*) *k*.
Trivial (uses *binds_exch*).

Generalization of *env_wf_exch*.

Lemma *env_wf_exch_3* : ∀ *E1 E2 E3 k*,
*env_wf* (*E1* & *E2* & *E3*) *k* → *env_wf* (*E1* & *E3* & *E2*) *k*.
Trivial (uses *binds_exch_3*).

Every type in a well-formed environment has the same kind.
Lemma *env_wf_binds_kind* : ∀ *E x t k*, *env_wf E k* →
*binds x t E* → *kinding t k*.
Trivial.

Every part of a well-formed environment must be well-formed.
Lemma *env_wf_concat_inv* : ∀ *E1 E2 k*, *env_wf* (*E1* & *E2*) *k* →
*env_wf E1 k* ∧ *env_wf E2 k*.
Trivial.

## 5.7 Regularity

A typing relation only holds when the environment is well-formed and the term is locally closed.
Lemma *typing_regular* : ∀ *E e T fvars*,
*typing E e T fvars* →
*env_wf E kind_star* ∧ *env_wf fvars kind_U* ∧ *term e*.
By induction on *typing E e T fvars*.

The answer predicate only holds for locally closed terms.
Lemma *answer_regular* : ∀ *e*,
*answer e* → *term e*.
Trivial induction on *answer e*.

The reduction relation only holds for pairs of locally closed terms.
Lemma *body_app* : ∀ *e e'*, *term e'* →
*body e* → *body* (*trm_app e e'*).
Trivial.

The reduction relation only applies to locally closed terms.
Lemma *red_regular* : ∀ *e e'*,
*red e e'* → *term e* ∧ *term e'*.
By induction on *red e e'*; uses *open_rec_term_open*.

## 5.8 Well-founded induction on subterms

Subterm relation on locally-closed terms.
Inductive *subterm* : *trm* → *trm* → Prop :=
| *sub_abs* : ∀ *x t*, *subterm* (*t* ^ *x*) (*trm_abs t*)
| *sub_abs_trans* : ∀ *x t t'*, *subterm t'* (*t* ^ *x*) → *subterm t'* (*trm_abs t*)
| *sub_app1* : ∀ *t1 t2*, *subterm t1* (*trm_app t1 t2*)
| *sub_app2* : ∀ *t1 t2*, *subterm t2* (*trm_app t1 t2*)
| *sub_app1_trans* : ∀ *t' t1 t2*, *subterm t' t1* → *subterm t'* (*trm_app t1 t2*)
| *sub_app2_trans* : ∀ *t' t1 t2*, *subterm t' t2* → *subterm t'* (*trm_app t1 t2*).

Size is defined to be the number of constructors used to build up a term.
Fixpoint *size* (*t*:*trm*) : *nat* :=
match *t* with
| *trm_fvar x* ⇒ 1
| *trm_bvar i* ⇒ 1
| *trm_abs t1* ⇒ 1 + *size t1*

| *trm_app t1 t2* ⇒ 1 + *size t1* + *size t2*
end.

Size is unaffected by substituting free variables for bound variables.
Lemma *size_subst_free* : ∀ *t i x*,
*size t* = *size* ({*i* ⇝ *trm_fvar x*} *t*).
By induction on *t*.

Special case of *size_subst_free*.
Lemma *size_open* : ∀ *t x*,
*size t* = *size* (*t* ^ *x*).
Follows directly from *size_subst_free*.

The subterm relation is well-founded[9].
Lemma *subterm_well_founded* : *well_founded subterm*.
We prove the more general property ∀ (*n*:*nat*) (*t*:*trm*), *size t* < *n* → *Acc subterm t* by induction on *n*.

## 5.9 Iterated domain subtraction

Removing a list of variables from the empty environment yields the empty environment.
Lemma *dsub_list_nil* : ∀ *xs*, *dsub_list xs nil* = *nil*.
Trivial.

Like *dsub_list_nil* but using *dsub_vars* instead of *dsub_list*.
Lemma *dsub_vars_nil* : ∀ *xs*, *dsub_vars xs nil* = *nil*.
Follows directly from *dsub_list_nil*.

Auxiliary lemma used to prove *dsub_list_inv*, below.
Lemma *dsub_list_inv_aux1* : ∀ *xs E v t*, *ok* ((*v*, *t*) :: *E*) →
*In v xs* → *dsub_list xs* ((*v*, *t*) :: *E*) = *dsub_list xs E*.
By induction on *xs*; uses *in_dom_dsub_inv*.

Auxiliary lemma used to prove *dsub_list_inv*, below.
Lemma *dsub_list_inv_aux2* : ∀ *xs E v t*,
¬ *In v xs* → *dsub_list xs* ((*v*, *t*) :: *E*) = (*v*, *t*) :: *dsub_list xs E*.
By induction on *xs*.

The following lemma is useful in proofs involving *dsub_list*. When we apply *dsub_list xs* to an environment with head (*v*, *t*), then either *v* is in the list *xs* and the head of the list will be removed, or *v* is not in the list *xs* and the head of the list will be left alone.
Lemma *dsub_list_inv* : ∀ *xs E v t*, *ok* ((*v*, *t*) :: *E*) →
( *In v xs* ∧ *dsub_list xs* ((*v*, *t*) :: *E*) = *dsub_list xs E*) ∨
(˜ *In v xs* ∧ *dsub_list xs* ((*v*, *t*) :: *E*) = (*v*, *t*) :: *dsub_list xs E*).
Follows from *dsub_list_inv_aux_1* and *dsub_list_inv_aux_2*.

Like *dsub_list_inv* but using *dsub_vars* instead of *dsub_list*.
Lemma *dsub_vars_inv* : ∀ *xs E v t*, *ok* ((*v*, *t*) :: *E*) →
(*v* \in *xs* ∧ *dsub_vars xs* ((*v*, *t*) :: *E*) = *dsub_vars xs E*) ∨
(*v* \notin *xs* ∧ *dsub_vars xs* ((*v*, *t*) :: *E*) = (*v*, *t*) :: *dsub_vars xs E*).
Follows from *dsub_list_inv*.

The order in which we remove variables from the domain of an environment is irrelevant.
Lemma *dsust_list_permut* : ∀ *E xs ys*, *ok E* →
(∀ *x*, *In x xs* → *In x ys*) →
(∀ *y*, *In y ys* → *In y xs*) →

[9] Proof suggested by Arthur Charguéraud.

dsub_list xs E = dsub_list ys E.
By induction on E; uses dsub_list_inv twice in the induction step (once for xs and once for ys).

Like dsust_list_permut, but using dsub_vars instead of dsub_list.
Lemma dsust_vars_permut : ∀ E xs ys, ok E →
(∀ x, x \in xs → x \in ys) →
(∀ y, y \in ys → y \in xs) →
dsub_vars xs E = dsub_vars ys E.
Proof analogous to dsust_list_permut but using dsub_vars_inv instead.

Special case of dsub_vars_inv.
Lemma dsub_vars_concat_assoc : ∀ E xs x t, ok (E & x ¬ t) →
x \notin xs → dsub_vars xs (E & x ¬ t) = (dsub_vars xs E) & x ¬ t.
Follows from dsub_vars_inv.

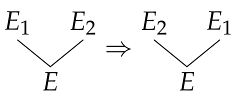
Special case of dsub_vars_inv.
Lemma dsub_vars_cons : ∀ E xs x t, ok (E & x ¬ t) → x \in xs →
dsub_vars xs (E & x ¬ t) = dsub_vars xs E.
Follows from dsub_vars_inv.

To remove ({{x}} \u xs) from the domain of an environment, we first remove x and then xs.
Lemma dsub_vars_to_dsub : ∀ E x xs, ok E →
dsub_vars ({{x}} \u xs) E = dsub_vars xs (dsub x E).
Follows from dsust_list_permut.

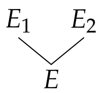
If x is in the domain of (E with xs removed), then x must be in the set (domain of E) with xs removed.
Lemma in_dom_dsub_vars : ∀ E x xs, ok E →
x \in dom ((dsub_vars xs) E) → x \in (S.diff (dom E) xs).
By induction on E; uses dsub_vars_inv in the induction step.

If x is not in the domain of E to start with, then it certainly will not be in the domain of E after we have removed some variables from the domain of E.
Lemma notin_dom_dsub_vars : ∀ E x xs, ok E →
x # E → x # (dsub_vars xs E).
Trivial.

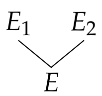
## 5.10 Context split

We can swap the two branches of a context split: 


Lemma split_exch : ∀ E E1 E2,
split_context E as (E1 ; E2) → split_context E as (E2 ; E1).
Trivial induction on split_context E as (E1 ; E2).

If 

and x is in the domain of $E_1$, then x must be in the domain of E.
Lemma in_dom_split_1 : ∀ E E1 E2 x,
split_context E as (E1 ; E2) → x \in dom E1 → x \in dom E.
By induction on split_context E as (E1 ; E2).

If 

and x is in the domain of $E_2$, then x must be in the domain of E.
Lemma in_dom_split_2 : ∀ E E1 E2 x,

*split_context E* as (*E1* ; *E2*) → *x* \in *dom E2* → *x* \in *dom E*.
Follows from *in_dom_split_1* and *split_exch*.

If $\underset{E}{\overset{E_1 \quad E_2}{\vee}}$ and *x* is in the domain of *E*, then *x* must either be in the domain of $E_1$ or in the domain of $E_2$ (or both).
Lemma *in_dom_split_inv* : ∀ *E E1 E2 x*,
*split_context E* as (*E1* ; *E2*) → *x* \in *dom E* → *x* \in *dom E1* ∨ *x* \in *dom E2*.
By induction on *split_context E* as (*E1* ; *E2*).

If $\underset{E}{\overset{E_1 \quad E_2}{\vee}}$ and $E_1$ binds *x*, then *E* must bind *x*. Note that unlike *in_dom_split_1*, we require *E* to be *ok*.
Lemma *binds_split_1* : ∀ *E E1 E2 x t*, *ok E* →
*split_context E* as (*E1* ; *E2*) → *binds x t E1* → *binds x t E*.
By induction on *split_context E* as (*E1* ; *E2*).

If $\underset{E}{\overset{E_1 \quad E_2}{\vee}}$ and $E_2$ binds *x*, then *E* must bind *x*. Note that unlike *in_dom_split_1*, we require *E* to be *ok*.
Lemma *binds_split_2* : ∀ *E E1 E2 x t*, *ok E* →
*split_context E* as (*E1* ; *E2*) → *binds x t E2* → *binds x t E*.
Follows from *binds_split_1* and *split_exch*.

If $\underset{E}{\overset{E_1 \quad E_2}{\vee}}$ and *E* binds *x*, then either $E_1$ or $E_2$ (or both) must bind *x*. Note that unlike *in_dom_split_inv*, we require *E* to be *ok*.
Lemma *binds_split_inv* : ∀ *E E1 E2 x t*,
*split_context E* as (*E1* ; *E2*) → *binds x t E* → *binds x t E1* ∨ *binds x t E2*.
By induction on *split_context E* as (*E1* ; *E2*).

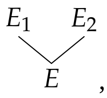
If $\underset{E}{\overset{E_1 \quad E_2}{\vee}}$ then $\underset{\triangleright_x E}{\overset{\triangleright_x E_1 \quad \triangleright_x E_2}{\vee}}$.
Lemma *split_dsub* : ∀ *E E1 E2 x*,
*split_context E* as (*E1* ; *E2*) → *ok E* →
*split_context* (*dsub x E*) as (*dsub x E1* ; *dsub x E2*).
By induction on split_context E as (E1 ; E2)].

We can always split an environment *E* as $\underset{E}{\overset{E \quad \emptyset}{\vee}}$.
Lemma *split_empty* : ∀ *E*,
*split_context E* as (*E* ; *empty*).
Trivial induction on *E*.

If $\underset{E}{\overset{E' \quad \emptyset}{\vee}}$ then *E* must be *E'*.
Lemma *split_empty_inv* : ∀ *E E'*,
*split_context E* as (*E'* ; *empty*) → *E* = *E'*.
We prove ∀ *E E' E''*, *split_context E* as (*E'* ; *E''*) → *E''* = *empty* → *E* = *E'* by induction on *split_context E* as (*E'* ; *E''*).

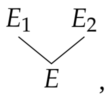
We can always split $E, x : t$ as $\underset{E, x : t}{\overset{E \quad x : t}{\vee}}$.

Lemma *split_tail* : ∀ *E x t*,
*split_context* (*E* & *x* ¬ *t*) as (*E* ; *x* ¬ *t*).
Follows from *split_empty*.

If 

, $E$ binds $x$ and $x$ is in the domain of $E_1$, then $E_1$ must bind $x$.

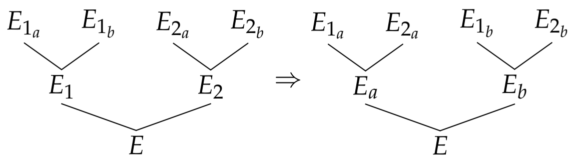
Lemma *split_binds_in_dom_1* : ∀ *E E1 E2*,
*split_context E* as (*E1* ; *E2*) → *ok E* →
∀ *x t*, *binds x t E* → *x* \in *dom E1* → *binds x t E1*.
By induction on *split_context E* as (*E1* ; *E2*).

If 

, $E$ binds $x$ and $x$ is in the domain of $E_2$, then $E_2$ must bind $x$.

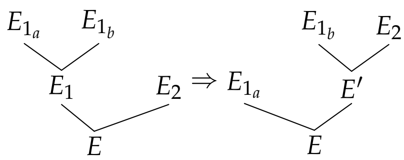
Lemma *split_binds_in_dom_2* : ∀ *E E1 E2*,
*split_context E* as (*E1* ; *E2*) → *ok E* →
∀ *x t*, *binds x t E* → *x* \in *dom E2* → *binds x t E2*.
Follows from *split_binds_in_dom_1* and *split_exch*.

We prove a series of four reordering lemmas, with the first the most general and the basis for the other three.



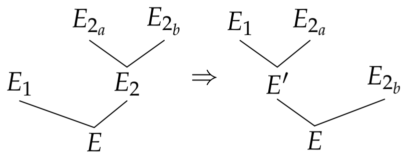

Lemma *reorder_ab'cd_ac'bd* : ∀ *E E1 E2*,
*split_context E* as (*E1* ; *E2*) → ∀ *E1a E1b E2a E2b*,
*split_context E1* as (*E1a* ; *E1b*) →
*split_context E2* as (*E2a* ; *E2b*) →
∃ *Ea*, ∃ *Eb*,
*split_context E* as (*Ea* ; *Eb*) ∧
*split_context Ea* as (*E1a* ; *E2a*) ∧
*split_context Eb* as (*E1b* ; *E2b*).
By induction on *split_context E* as (*E1* ; *E2*) followed by inversion on *split_context E1* as (*E1a* ; *E1b*) and *split_context E2* as (*E2a* ; *E2b*). There are 34 cases to consider but they are all trivial.

Restructure a three-way split $(E_a, E_b, E_c)$.



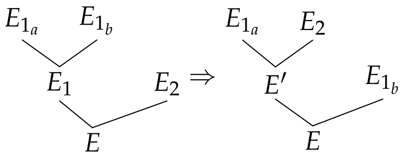

*Corollary reorder_ab'c_a'bc* : ∀ *E E1 E2 E1a E1b*,
*split_context E* as (*E1* ; *E2*) →
*split_context E1* as (*E1a* ; *E1b*) →
∃ *E'*,
*split_context E* as (*E1a* ; *E'*) ∧
*split_context E'* as (*E1b* ; *E2*).
Follows from *reorder_ab'cd_ac'bd*.

Inverse of *reorder_ab'c_a'bc*:

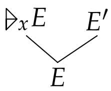



*Corollary reorder_a'bc_ab'c* : ∀ *E E1 E2 E2a E2b*,
*split_context E* as (*E1* ; *E2*) →
*split_context E2* as (*E2a* ; *E2b*) →
∃ *E'*,
*split_context E* as (*E'* ; *E2b*) ∧
*split_context E'* as (*E1* ; *E2a*).
Follows from *reorder_ab'cd_ac'bd*.

The final reordering lemma is its own inverse:

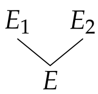



*Corollary reorder_ab'c_ac'b* : ∀ *E E1 E2 E1a E1b*,
*split_context E* as (*E1* ; *E2*) →
*split_context E1* as (*E1a* ; *E1b*) →
∃ *E'*,
*split_context E* as (*E'* ; *E1b*) ∧
*split_context E'* as (*E1a* ; *E2*).
Follows from *reorder_ab'cd_ac'bd*.

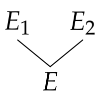
Remove the assumption about *x* from *E*: 

. We use this lemma in *split_dom_inv*, below.
Lemma *split_dom* : ∀ *E x*,
∃ *E'*, *split_context E* as (*dsub x E* ; *E'*) ∧ *dsub x E' = empty*.
By induction on *E*.

Inverse property of *split_dom*.
Lemma *split_dom_inv* : ∀ *E E' x*,
*E' = dsub x E* →
∃ *E_x*, *split_context E* as (*E'* ; *E_x*) ∧ *dsub x E_x = empty*.
By induction on *E*.

If 

and *E* is *ok*, then both *E1* and *E2* must be *ok*.
Lemma *split_context_ok* : ∀ *E E1 E2*,
*split_context E* as (*E1* ; *E2*) → *ok E* → *ok E1* ∧ *ok E2*.
By induction on *split_context E* as (*E1* ; *E2*).

If 

and *E* is well-formed, then both *E1* and *E2* must be well-formed.
Lemma *split_context_wf* : ∀ *E E1 E2 k*,
*split_context E* as (*E1* ; *E2*) → *env_wf E k* → *env_wf E1 k* ∧ *env_wf E2 k*.
Follows from *split_context_ok*, *binds_split_1* and *binds_split_2*.

We can always split the concatenation of two environments into its two constituents: $\begin{array}{cc} E_1 & E_2 \\ \multicolumn{2}{c}{E_1 \& E_2} \end{array}$ .
Lemma *split_concat* : ∀ *E2 E1*,
*split_context* (*E1* & *E2*) as (*E1* ; *E2*).
By induction on *E2*.

Split a domain *E* into two domains *E1* and *E2* so that all assumptions about variables in *xs* go into *E1* and the rest goes into *E2*: $\begin{array}{cc} \triangleright_{\mathrm{dom}\ E \setminus xs} E & \triangleright_{xs} E \\ \multicolumn{2}{c}{E} \end{array}$ .
Lemma *split_dom_set* : ∀ *E xs*, *ok E* →
*split_context E* as (*dsub_vars* (*S.diff* (*dom E*) *xs*) *E*; *dsub_vars xs E*).
By induction on *E*. The proof is slightly tricky, and relies on *dsust_vars_permut*, *dsub_vars_concat_assoc*, *dsub_vars_cons* and *dsub_vars_to_dsub*.

## 5.11 Type equivalence

If *t1 t2* is equivalent to *s*, then *s* must be of the form *s1 s2* where *t1* and *s1*, and *t2* and *s2*, are equivalent. This however only holds for types of kind other than *U* (counterexample: *typ_equiv* (*or a* (*not a*)) *a*).
Lemma *typ_equiv_app_inv_ex* : ∀ *t s*, ¬ *kinding t kind_U* →
*typ_equiv t s* →
(∀ *t1 t2* , *t* = *typ_app t1 t2* → ∃ *s1*, ∃ *s2*,
*s* = *typ_app s1 s2* ∧ *typ_equiv t1 s1* ∧ *typ_equiv t2 s2*) ∧
(∀ *s1 s2* , *s* = *typ_app s1 s2* → ∃ *t1*, ∃ *t2*,
*t* = *typ_app t1 t2* ∧ *typ_equiv t1 s1* ∧ *typ_equiv t2 s2*).
By induction on *typ_equiv t s*. This is a slightly tricky proof, and we do need to prove it in both directions (as stated in the lemma). If we try to prove it in one direction only, we get stuck in the case for *typ_equiv_sym*.

If *t1 s1* is equivalent to *t2 s2*, then the components must be equivalent.
Lemma *typ_equiv_app_inv* : ∀ *t1 t2 s1 s2*,
¬ *kinding* (*typ_app t1 s1*) *kind_U* →
*typ_equiv* (*typ_app t1 s1*) (*typ_app t2 s2*) →
*typ_equiv t1 t2* ∧ *typ_equiv s1 s2*.
Follows from *typ_equiv_app_inv*.

If *t* is equivalent to *ATTR*, it must be *ATTR*.
Lemma *typ_equiv_ATTR_inv* : ∀ *t s*, *typ_equiv t s* →
(*t* = *ATTR* → *s* = *ATTR*) ∧ (*s* = *ATTR* → *t* = *ATTR*).
By induction on *typ_equiv t s*.

Special case of *typ_equiv_app_inv_ex* for attributed types.
Lemma *typ_equiv_attr_inv_ex* : ∀ *t u s*,
*typ_equiv s* (*t* ' *u*) → ∃ *t'*, ∃ *u'*,
*s* = *t'* ' *u'* ∧ *typ_equiv t t'* ∧ *typ_equiv u u'*.
Follows from *typ_equiv_app_inv_ex*.

Special case of *typ_equiv_app_inv* for attributed types.
Lemma *typ_equiv_attr_inv* : ∀ *t u s v*,
*typ_equiv* (*t* ' *u*) (*s* ' *v*) → *typ_equiv t s* ∧ *typ_equiv u v*.
Follows from *typ_equiv_app_inv*.

Special case of *typ_equiv_app_inv* for function types.
Lemma *typ_equiv_fun_inv* : ∀ *a u b a' u' b'*,
*typ_equiv* (*a* ⟨ *u* ⟩ *b*) (*a'* ⟨ *u'* ⟩ *b'*) →

*typ_equiv a a'* ∧
*typ_equiv u u'* ∧
*typ_equiv b b'*.
Follows from *typ_equiv_attr_inv*.

Replace an attribute on an attributed type.
Lemma *typ_equiv_new_attr* : ∀ *t u v*, *typ_equiv u v* ⟶
*typ_equiv* (*t* ' *u*) (*t* ' *v*).
Trivial.

Replace the domain of an arrow
Lemma *typ_equiv_fun_new_dom* : ∀ *a u b a'*, *typ_equiv a a'* ⟶
*typ_equiv* (*a* ⟨ *u* ⟩ *b*) (*a'* ⟨ *u* ⟩ *b*).
Trivial.

Replace the codomain of an arrow
Lemma *typ_equiv_fun_new_cod* : ∀ *a u b b'*, *typ_equiv b b'* ⟶
*typ_equiv* (*a* ⟨ *u* ⟩ *b*) (*a* ⟨ *u* ⟩ *b'*).
Trivial.

If *t* and *s* are equivalent and have kind *U*, then they must also be equivalent by the boolean equivalence relation.
Lemma *typ_equiv_BA_equiv* : ∀ *t s*,
*typ_equiv t s* ⟶ *kinding t kind_U* ⟶ *BA.equiv t s*.
By induction on *typ_equiv t s*.

Commutativity of *or*.
Lemma *typ_equiv_comm_or* : ∀ *a b*, *kinding* (*or a b*) *kind_U* ⟶
*typ_equiv* (*or a b*) (*or b a*).
Trivial.

## 5.12 Non-unique types

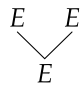
If *t* and *s* are equivalent and *t* is *non_unique*, *s* must be *non_unique*.
Lemma *non_unique_equiv* : ∀ *t s*, *typ_equiv t s* ⟶
*non_unique t* ⟶ *non_unique s*.
By inversion on *non_unique t*.

If $t^u$ is non-unique, then *u* must be equivalent to false.
Lemma *non_unique_star* : ∀ *t u*,
*non_unique* (*t* ' *u*) ⟶ *typ_equiv u NU*.
By inversion on *non_unique* (*t* ' *u*). There are two possibilities (see the definition of *non_unique*). For the first case, (*t* ' *u*) of kind ∗, the lemma follows immediately. For the second, we show that *kinding* (*t* ' *u*) *kind_U* leads to contradiction.

If *u* is *non_unique* and has kind *U*, it must be equivalent to false.
Lemma *non_unique_U* : ∀ *u*,
*non_unique u* ⟶ *kinding u kind_U* ⟶ *typ_equiv u NU*.
By inversion on *non_unique u*. Proof analogous to *non_unique_star*.

If $\begin{smallmatrix} E_1 & & E_2 \\ & \diagdown\diagup & \\ & E & \end{smallmatrix}$, *E* binds *x*, and both *E1* and *E2* bind *x*, then *x* must have a non-unique type. That is, only variables of non-unique type can be duplicated.
Lemma *split_both_inv* : ∀ *E E1 E2 x t*, *ok E* ⟶
*split_context E* as (*E1* ; *E2*) ⟶
*binds x t E* ⟶ *x* \in *dom E1* ⟶ *x* \in *dom E2* ⟶

*non_unique t*.
By induction on *split_context E* as (*E1* ; *E2*).

If every type in *E* is non-unique, then 

.
Lemma *split_non_unique* : ∀ *E*, *ok E* →
 (∀ *x t*, *binds x t E* → *non_unique t*) →
 *split_context E* as (*E* ; *E*).
By induction on *E*.

## 5.13 Equivalence of environments.

We start with a number of trivial consequences of ≅. These lemmas enable us to work directly with the notion of an equivalence, rather than having to unfold the definition of ≅ every time we need one of its constituents.

Equivalence only holds between well-formed environments.
Lemma *env_equiv_regular* : ∀ *E1 E2 k*,
 (*E1* ≅ *E2*) *k* → *env_wf E1 k* ∧ *env_wf E2 k*.
Trivial.

If *E1* ≅ *E2* and *E1* binds *x*, then *E2* must bind *x*.
Lemma *env_equiv_binds_1* : ∀ *E1 E2 k*, (*E1* ≅ *E2*) *k* →
 ∀ *x t*, *binds x t E1* → ∃ *t'*, *binds x t' E2* ∧ *typ_equiv t t'*.
Trivial.

If *E1* ≅ *E2* and *E2* binds *x*, then *E1* must bind *x*.
Lemma *env_equiv_binds_2* : ∀ *E1 E2 k*, (*E1* ≅ *E2*) *k* →
 ∀ *x t*, *binds x t E2* → ∃ *t'*, *binds x t' E1* ∧ *typ_equiv t t'*.
Trivial.

If *E1* ≅ *E2* and *x* is in the domain of *E1*, *x* must be in the domain of *E2*.
Lemma *env_equiv_in_dom_1* : ∀ *E1 E2 k*, (*E1* ≅ *E2*) *k* →
 ∀ *x*, *x* \in *dom E1* → *x* \in *dom E2*.
Follows directly from *binds_in_dom* and *env_equiv_binds_1*.

If *E1* ≅ *E2* and *x* is in the domain of *E2*, *x* must be in the domain of *E1*.
Lemma *env_equiv_in_dom_2* : ∀ *E1 E2 k*, (*E1* ≅ *E2*) *k* →
 ∀ *x*, *x* \in *dom E2* → *x* \in *dom E1*.
Follows directly from *binds_in_dom* and *env_equiv_binds_2*.

The equivalence relation is reflexive.
Lemma *env_equiv_refl* : ∀ *E k*, *env_wf E k* → (*E* ≅ *E*) *k*.
Trivial.

The equivalence relation is commutative.
Lemma *env_equiv_comm* : ∀ *E1 E2 k*, (*E1* ≅ *E2*) *k* → (*E2* ≅ *E1*) *k*.
Trivial.

The equivalence relation is transitive.
Lemma *env_equiv_trans* : ∀ *E1 E2 E3 k*,
 (*E1* ≅ *E2*) *k* → (*E2* ≅ *E3*) *k* → (*E1* ≅ *E3*) *k*.
Trivial.

If *E* is equivalent to the empty environment, it must be the empty environment.
Lemma *env_equiv_empty* : ∀ *E k*, (*E* ≅ *empty*) *k* → *E* = *empty*.
By case analysis on *E*.

If *E* is equivalent to a singleton environment, it must be that singleton environment.
Lemma *env_equiv_singleton* : ∀ *E k y s*, (*E* ≅ (*y* ¬ *s*)) *k* →
∃ *s'*, *E* = *y* ¬ *s'* ∧ *typ_equiv s s'*.
By case analysis on *E*; distinguishing between the empty environment, the singleton environment, and the environment with more than one element. We show contradiction for all cases except the singleton case.

Equivalence between environments is unaffected if we remove a variable from both sides.
Lemma *env_equiv_dsub* : ∀ *E1 E2 k x*,
(*E1* ≅ *E2*) *k* → (*dsub x E1* ≅ *dsub x E2*) *k*.
Follows from *binds_in_dom*, *binds_dsub* and *binds_dsub_inv*.

Equivalence between environments is unaffected if we add a variable on both sides, provided that that variable wasn't already in the domain of the environments to start with and has the right kind.
Lemma *env_equiv_extend* : ∀ *E E' k x t s*, *x* # *E* →
*kinding t k* → *typ_equiv t s* →
(*E* ≅ *E'*) *k* → (*E* & *x* ¬ *t* ≅ *E'* & *x* ¬ *s*) *k*.
Trivial.

Special case of *env_equiv_extend*.
Lemma *env_equiv_typ_equiv* : ∀ *E k x t s*, *env_wf* (*E* & *x* ¬ *t*) *k* →
*typ_equiv t s* →
(*E* & *x* ¬ *t* ≅ *E* & *x* ¬ *s*) *k*.
Follows from *env_equiv_extend* and *env_wf_binds_kind*.

Special case of *env_equiv_dsub*.
Lemma *env_equiv_cons* : ∀ *E E' k x t*,
(*E* & *x* ¬ *t* ≅ *E'*) *k* → (*E* ≅ *dsub x E'*) *k*.
Follows from *env_equiv_dsub* and *dsub_not_in_dom*.

Inverse property of *env_equiv_cons*.
Lemma *env_equiv_cons_inv* : ∀ *E E' k x t*,
(*dsub x E* ≅ *E'*) *k* →
*binds x t E* → *env_wf E k* →
(*E* ≅ *E'* & *x* ¬ *t*) *k*.
Follows from *binds_dsub* and *binds_dsub_inv*.

We can take an environment *E*, remove its assumption about *x*, and then re-insert that assumption at the start of the environment; the result will be equivalent to the original environment.
Lemma *env_equiv_reorder* : ∀ *E k x t*,
*env_wf E k* → *binds x t E* → (*E* ≅ *dsub x E* & *x* ¬ *t*) *k*.
Follows from *binds_dsub* and *binds_dsub_inv*.

If $\overset{E_1 \quad E_2}{\diagdown\diagup}\atop E$ and *E'* is equivalent to *E*, then there exist two environments $E'_1$ and $E'_2$ such that $\overset{E'_1 \quad E'_2}{\diagdown\diagup}\atop E'$ and $E_1$ and $E_2$ are equivalent to $E'_1$ and $E'_2$.
Lemma *env_equiv_split* : ∀ *E' E E1 E2 k*,
*split_context E* as (*E1* ; *E2*) → (*E* ≅ *E'*) *k* →
∃ *E1'*, ∃ *E2'*,
*split_context E'* as (*E1'* ; *E2'*) ∧ (*E1* ≅ *E1'*) *k* ∧ (*E2* ≅ *E2'*) *k*.
By induction on E'. For the case (*v*, *t*) :: *E'*, we recurse on *dsub v E*, then add (*v*, *t*) to the partially constructed *E1'* or *E2'* depending on whether *v* \in *dom E1* or *v* \in *dom E2*.

Equivalence is unaffected by order.
Lemma *env_equiv_exch* : ∀ *E1 E2 k*, *env_wf* (*E1* & *E2*) *k* →
(*E1* & *E2* ≅ *E2* & *E1*) *k*.

Follows trivially from *env_wf_exch* and *binds_exch*.

Generalization of *env_equiv_exch*.
Lemma *env_equiv_exch_3* : ∀ *E1 E2 E3 k*, *env_wf* (*E1* & *E2* & *E3*) *k* →
(*E1* & *E2* & *E3* ≅ *E1* & *E3* & *E2*) *k*.
Follows trivially from *env_wf_exch_3* and *binds_exch_3*.

## 5.14 Range

The range of an environment containing only types of kind *U* is *U*.
Lemma *rng_kind_U* : ∀ *E*, *env_wf E kind_U* → *kinding* (*rng E*) *kind_U*.
By induction on *E*.

Auxiliary lemma used to prove *rng_non_unique*.
Lemma *rng_non_unique_BA* : ∀ *fvars*,
*BA.equiv* (*rng fvars*) *NU* →
(∀ *x u*, *binds x u fvars* → *BA.equiv u NU*).
By induction on *fvars*, using lemma *or_false_both* from the Boolean Algebra formalization.

If the range of an environment is equivalent to false, then every attribute in that environment must be equivalent to false.
Lemma *rng_non_unique* : ∀ *fvars*, *env_wf fvars kind_U* →
*typ_equiv* (*rng fvars*) *NU* →
(∀ *x u*, *binds x u fvars* → *typ_equiv u NU*).
Follows from *rng_non_unique_BA*, *env_wf_binds_kind* and *typ_equiv_BA_equiv*.

Auxiliary lemma used to prove *split_rng*.
Lemma *split_rng_BA* : ∀ *fvars fvars1 fvars2*,
*split_context fvars* as (*fvars1* ; *fvars2*) →
*BA.equiv* (*rng fvars*) (*or* (*rng fvars1*) (*rng fvars2*)).
By induction on *split_context fvars* as (*fvars1* ; *fvars2*), using properties of the boolean equivalence relation and *rng_concat*.

If $\underset{fvars}{fvars_1 \quad fvars_2}$ then the range of *fvars* is equivalent to the range of the concatenation of *fvars1* and *fvars2*. This holds because if there is an assumption about *x* in both *fvars1* and *fvars2*, then that must be the same assumption, and we know that *t* is equivalent to *or t t* for any *t* (disjunction is idempotent).
Lemma *split_rng* : ∀ *fvars fvars1 fvars2*, *env_wf fvars kind_U* →
*split_context fvars* as (*fvars1* ; *fvars2*) →
*typ_equiv* (*rng fvars*) (*or* (*rng fvars1*) (*rng fvars2*)).
Follows from *split_rng_BA*.

Auxiliary lemma needed to prove *env_equiv_rng*.
Lemma *rng_reorder* : ∀ (*E* : *env*) *x t*, *binds x t E* →
*BA.equiv* (*rng E*) (*or* (*rng* (*dsub x E*)) *t*).
By induction on *E*.

If two environments are equivalent, then their ranges must be equivalent.
Lemma *env_equiv_rng* : ∀ *E E'*,
(*E* ≅ *E'*) *kind_U* → *typ_equiv* (*rng E*) (*rng E'*).
By induction on *E*, using properties of the boolean equivalence relation, *rng_reorder*, *rng_concat* and *typ_equiv_BA_equiv*.

# 6 Properties of the typing relation

## 6.1 Kinding properties

Every assumption in $E$ must have kind $*$.
Lemma $kinding\_env$ : $\forall$ $E$ $e$ $t$ $fvars$,
  $E \vdash e : t \mid fvars \rightarrow \forall$ $x$ $s$,
  $binds$ $x$ $s$ $E \rightarrow kinding$ $s$ $kind\_star$.
Follows trivially from regularity and $env\_wf\_binds\_kind$.

Every assumption in $fvars$ must have kind $\mathcal{U}$.
Lemma $kinding\_fvars$ : $\forall$ $E$ $e$ $t$ $fvars$,
  $E \vdash e : t \mid fvars \rightarrow \forall$ $x$ $u$,
  $binds$ $x$ $u$ $fvars \rightarrow kinding$ $u$ $kind\_U$.
Follows trivially from regularity and $env\_wf\_binds\_kind$.

If $e$ has type $t$, then $t$ must have kind $*$.
Lemma $typing\_kind\_star$ : $\forall$ $E$ $e$ $t$ $fvars$,
  $E \vdash e : t \mid fvars \rightarrow kinding$ $t$ $kind\_star$.
By induction on $E \vdash e : t \mid fvars$.

## 6.2 Free variables

If $E \vdash e : T \mid fvars$, then if $x$ is free in $e$ it must be in the domain of $E$ and in the domain of $fvars$.
Lemma $typing\_fv$ : $\forall$ $E$ $e$ $T$ $fvars$,
  $E \vdash e : T \mid fvars \rightarrow \forall$ $x$, $x$ \in $fv$ $e \rightarrow x$ \in $dom$ $E \wedge x$ \in $dom$ $fvars$.
By induction on $E \vdash e : T \mid fvars$.

If there is an evaluation context $E$ such that $t = E[x]$, then $x$ must be free in $t$.
Lemma $eval\_fv$ : $\forall$ $t$ $x$,
  $evals$ $t$ $x \rightarrow x$ \in $fv$ $t$.
By induction on $evals$ $t$ $x$.

## 6.3 Consistency of *E* and *fvars*

Every assumption in $fvars$ must have a corresponding assumption in $E$.
Lemma $fvars\_and\_env\_consistent$ : $\forall$ $E$ $e$ $S$ $fvars$ $x$ $u$,
  $E \vdash e : S \mid fvars \rightarrow binds$ $x$ $u$ $fvars \rightarrow$
  $\exists$ $t$, $\exists$ $v$, $binds$ $x$ ($t$ ’ $v$) $E \wedge typ\_equiv$ $u$ $v$.
By induction on $E \vdash e : S \mid fvars$.

Every assumption in $E$ must have a corresponding assumption in $fvars$.
Lemma $env\_and\_fvars\_consistent$ : $\forall$ $E$ $e$ $S$ $fvars$ $x$ $t$ $u$,
  $E \vdash e : S \mid fvars \rightarrow binds$ $x$ ($t$ ’ $u$) $E \rightarrow x$ \in $fv$ $e \rightarrow$
  $\exists$ $v$, $binds$ $x$ $v$ $fvars \wedge typ\_equiv$ $u$ $v$.
By induction on $E \vdash e$ :˜ $S \mid fvars$.

## 6.4 Weakening

Auxiliary lemma used to prove $unused\_assumptions$.
Lemma $unused\_assumption\_env$ : $\forall$ $E$ $e$ $T$ $fvars$ $x$,
  $E \vdash e : T \mid fvars \rightarrow x$ \notin $fv$ $e \rightarrow dsub$ $x$ $E \vdash e : T \mid fvars$.
By induction on $E \vdash e : T \mid fvars$.

Auxiliary lemma used to prove $unused\_assumptions$.
Lemma $unused\_assumptions\_list$ : $\forall$ $xs$ $E$ $e$ $T$ $fvars$,

E ⊢ e : T | fvars → (∀ x, In x xs → x \notin fv e) →
dsub_list xs E ⊢ e : T | fvars.
By induction on *xs*, using *unused_assumption_env*.

We can remove all assumptions in *E* about variables that are not free in *e*.
Lemma *unused_assumptions* : ∀ xs E e T fvars,
E ⊢ e : T | fvars → (∀ x, x \in xs → x \notin fv e) →
dsub_vars xs E ⊢ e : T | fvars.
Follows trivially from *unused_assumptions_list*.

We can append unused assumptions to the typing environment.
Lemma *weakening_1* : ∀ E1 e T fvars,
E1 ⊢ e : T | fvars → ∀ E E2, env_wf E kind_star →
split_context E as (E1 ; E2) → E ⊢ e : T | fvars.
By induction on *E1 ⊢ e : T | fvars*.

We can prepend unused assumptions to the typing environment.
Lemma *weakening_2* : ∀ E2 e T fvars,
E2 ⊢ e : T | fvars → ∀ E E1, env_wf E kind_star →
split_context E as (E1 ; E2) → E ⊢ e : T | fvars.
Follows trivially from *weakening_1* and *split_exch*.

Every assumption in *fvars* must be used.
Lemma *no_fvars_weakening* : ∀ E e T fvars,
E ⊢ e : T | fvars → ∀ x, x \notin fv e → x # fvars.
By induction on *E ⊢ e : T | fvars*.

Since every assumption in *fvars* must be used, if *x* is not free in *e* then removing *x* from *fvars* has no effect (since it wasn't in *fvars* to start with).
Lemma *unused_assumption_fvars* : ∀ E e T fvars x,
E ⊢ e : T | fvars → x \notin fv e → E ⊢ e : T | dsub x fvars.
Follows trivially from *no_fvars_weakening* and *dsub_not_in_dom*.

Combination of *unused_assumption_env* and *unused_assumption_fvars*.
Lemma *unused_assumption* : ∀ E e T fvars x,
E ⊢ e : T | fvars → x \notin fv e → dsub x E ⊢ e : T | dsub x fvars.
Follows directly from *unused_assumption_fvars* and *unused_assumption_env*.

If *e* can be typed in environment *E*, we can split *E* into two environments *E1* and *E2* such that every assumption about variables in *e* will be in *E1*; then *e* can also be typed in environment *E1*.
Lemma *split_env* : ∀ E e t u fvars,
E ⊢ e : t ' u | fvars →
(∃ E1, ∃ E2,
split_context E as (E1 ; E2) ∧
E1 ⊢ e : t ' u | fvars ∧
(∀ x, x \in dom E1 → x \in fv e)).
Follows from *split_dom_set*.

## 6.5 Exchange

We can replace both *E* and *fvars* by equivalent environments. This is a powerful lemma, because the definition of equivalence for environment is very general (in particular, it allows to replace a type by an equivalent type).
Lemma *env_equiv_typing* : ∀ E e T fvars,
E ⊢ e : T | fvars → ∀ E' fvars',
(E ≅ E') kind_star → (fvars ≅ fvars') kind_U →

*E' ⊢ e : T | fvars'*.
By induction on *E ⊢ e : T | fvars*. This proof is slightly tricky. The case of variables relies on *env_equiv_singleton*. In the case for abstraction, we need *env_equiv_rng*, *env_equiv_extend* and *env_equiv_cons_inv*, and in the case for application we need *env_equiv_split*.

Change the order of the assumptions in the environment.
Lemma *exchange* : ∀ *E1 E2 E3 e T fvars*,
  *E1* & *E2* & *E3* ⊢ *e* : *T* | *fvars* →
  *E1* & *E3* & *E2* ⊢ *e* : *T* | *fvars*.
Follows trivially from *env_equiv_typing* and *env_equiv_exch_3*.

Replace an assumption in the environment by an equivalent one.
Lemma *typ_equiv_env* : ∀ *E x s s' e t fvars*,
  *E* & *x* ¬ *s* ⊢ *e* : *t* | *fvars* → *typ_equiv s s'* →
  *E* & *x* ¬ *s'* ⊢ *e* : *t* | *fvars*.
Follows trivially from *env_equiv_typing* and *env_equiv_typ_equiv*.

## 6.6 Inversion lemmas

Inversion lemma for variables.
Lemma *typing_var_inv* : ∀ *E x s fvars*,
  *E* ⊢ *trm_fvar x* : *s* | *fvars* →
    ∃ *t*, ∃ *u*, ∃ *v*,
      *typ_equiv s* (*t* ' *u*) ∧
      *fvars* = *x* ¬ *v* ∧
      *env_wf E kind_star* ∧
      *binds x* (*t* ' *u*) *E* ∧
      *typ_equiv u v*.
We prove the more general lemma ∀ *E e s fvars*, *E* ⊢ *e* : *s* | *fvars* → ∀ *x*, *e* = *trm_fvar x* → ∃ *t*, ∃ *u*, ∃ *v*, *typ_equiv s* (*t* ' *u*) ∧ *fvars* = *x* ¬ *v* ∧ *env_wf E kind_star* ∧ *binds x* (*t* ' *u*) *E* ∧ *typ_equiv u v*) by induction on *E* ⊢ *e* : *s* | *fvars*. The case for variables is trivial, the cases for application and abstraction can be dismissed, and the case for *typing_equiv* is a straightforward application of the induction hypothesis.

Inversion lemma for application.
Lemma *typing_app_inv* : ∀ *E e1 e2 s fvars*,
  *E* ⊢ *trm_app e1 e2* : *s* | *fvars* →
    ∃ *E1*, ∃ *E2*, ∃ *fvars1*, ∃ *fvars2*,
    ∃ *a*, ∃ *b*, ∃ *u*,
      *typ_equiv s b* ∧
      *E1* ⊢ *e1* : *a* ⟨ *u* ⟩ *b* | *fvars1* ∧
      *E2* ⊢ *e2* : *a* | *fvars2* ∧
      *split_context E* as (*E1* ; *E2*) ∧ *env_wf E kind_star* ∧
      *split_context fvars* as (*fvars1* ; *fvars2*) ∧ *env_wf fvars kind_U*.
Analogous to the proof of the inversion lemma for variables.

Inversion lemma for abstraction.
Lemma *typing_abs_inv* : ∀ *E e s fvars'*,
  *E* ⊢ *trm_abs e* : *s* | *fvars'* →
    ∃ *L*, ∃ *a*, ∃ *b*,
      *typ_equiv s* (*a* ⟨ *rng fvars'* ⟩ *b*) ∧
      (∀ *x fvars*, *x* \notin *L* → *fvars'* = *dsub x fvars* →
        (*E* & *x* ¬ *a*) ⊢ *e* ˆ *x* : *b* | *fvars*).
Analogous to the proof of the inversion lemma for variables.

Tactic *typing_inversion* can be used instead of a call to the standard Coq tactic *inversion* to do inversion on the typing relation using the inversion lemmas we just proved.

```
Ltac typing_inversion H :=
  match type of H with
  | ?E ⊢ trm_fvar ?x : ?T | ?fvars ⇒
      let t := fresh "t" in
      let u := fresh "u" in
      let v := fresh "v" in
      elim3 (typing_var_inv H) t u v (?, (?, (?, (?, ?))))
  | ?E ⊢ trm_app ?e1 ?e2 : ?T | ?fvars ⇒
      let E1 := fresh "E1" in
      let E2 := fresh "E2" in
      let fvars1 := fresh "fvars1" in
      let fvars2 := fresh "fvars2" in
      let a := fresh "a" in
      let b := fresh "b" in
      let u := fresh "u" in
      elim7 (typing_app_inv H) E1 E2 fvars1 fvars2 a b u (?, (?, (?, (?, (?, (?, ?))))))
  | ?E ⊢ trm_abs ?e : ?T | ?fvars ⇒
      let L := fresh "L" in
      let a := fresh "a" in
      let b := fresh "b" in
      elim3 (typing_abs_inv H) L a b (?, ?)
  end.
```

# 7 Subject reduction

## 7.1 Progress

If *e* is locally-closed, then either it is an answer, it reduces to some other term *e'*, or there exists an evaluation context *E* such that *e* = *E*[*x*] for some free variable *x* in *e*.

```
Lemma weak_progress : ∀ e, term e →
  answer e ∨
  (∃ e':trm, red e e') ∨
  (∃ x, x \in fv e ∧ evals e x).
```

By complete structural induction on *term e* (using *subterm_well_founded*).

If *e* can be typed in the empty environment, then either *e* is an answer or it reduces to some other term *e'*.

```
Theorem progress : ∀ e T fvars,
  empty ⊢ e : T | fvars → answer e ∨ ∃ e', red e e'.
```

Follows from *weak_progress* and *typing_fv*.

## 7.2 Preservation

When a function is non-unique, then all of the elements in its closure must be non-unique. In other words, all assumptions about the free variables of the function must be non-unique. That means that we can type the function in an environment *E'* (which is *E* stripped from all unnecessary assumptions) so that we can duplicate *E'* (split it into *E'* twice). We will need this lemma in the substitution lemma, when we have to substitute a function for a free variable in both terms of an application (i.e., when we have to duplicate the function, or in other words, apply it twice).

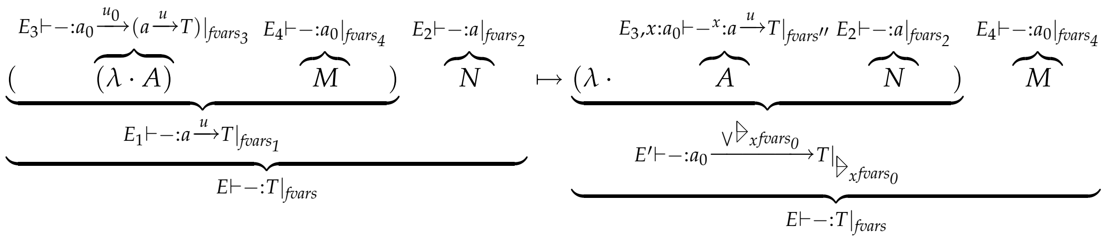
Lemma *shared_function* : ∀ *E e a b u_f fvars*,
  *E* ⊢ *trm_abs e* : *a* ⟨ *u_f* ⟩ *b* | *fvars* →
  *typ_equiv* (*rng fvars*) *NU* →
  ∃ *E'*, ∃ *E"*,
    *E'* ⊢ *trm_abs e* : *a* ⟨ *u_f* ⟩ *b* | *fvars* ∧
    *split_context E* as (*E'* ; *E"*) ∧
    *split_context E'* as (*E'* ; *E'*) ∧
    *split_context fvars* as (*fvars* ; *fvars*).
Follows from *split_env*, *rng_non_unique* and *fvars_and_env_consistent*.

The substitution lemma is probably the most difficult lemma in the subject reduction proof. This is not surprising, because when we substitute a term *e2* for *x* in *e1*, *e2* may be duplicated (when there is more than one use for *x* in *e1*). That is not necessarily a problem, because when there is more than one use of *x* in *e1*, then *x* must have a non-unique type and therefore it should be okay to duplicate *e2*. However, for the result of the substitution to be well-typed, if *e2* is duplicated, we must also duplicate all the assumptions that are needed to type *e2*, and that is not possible in the general case (we may need a unique assumption even when the result is non-unique). However, in the specific case that *e2* is an abstraction, we know that if *e2* is non-unique, that all of the elements in its closure must be non-unique, and so we can actually duplicate all assumptions required to type *e2* (this is what we proved in the previous lemma).
Lemma *substitution* : ∀ *e1*, *term e1* →
  ∀ *E E1 E2 fvars fvars1 fvars2 x a b e2 T*,
    *split_context E* as (*E1* ; *E2*) → *env_wf E kind_star* →
    *split_context fvars* as (*fvars1* ; *fvars2*) → *env_wf fvars kind_U* →
    *E1* & *x* ¬ (*a* ⟨ *rng fvars2* ⟩ *b*) ⊢ *e1* : *T* | *fvars1* & *x* ¬ *rng fvars2* →
    *E2* ⊢ *trm_abs e2* : *a* ⟨ *rng fvars2* ⟩ *b* | *fvars2* →
    *x* \notin (*dom E1* \u *dom E2* \u *dom fvars1*) →
    *x* \in *fv e1* →
    *E* ⊢ [x ˜> trm_abs e2] *e1* : *T* | *fvars*.
By induction on *term e1*. For the case of variables, we know that *e1* must be *x* (it cannot be a different variable because of the requirement that *x* must be free in *e1*), and the lemma follows from *weakening_2*. In the case for an application *e1 e1'*, we do case analysis on *x* \in *fv e1* and *x* \in *fv e2* (again, it cannot be in neither because of the same requirement). If it is *e1* but not in *e1'*, or in *e1'* but not in *e1*, then it is a matter of reordering the environment so that the assumptions about *e2* are passed to the appropriate branch of the application. If it is in both, then we know that *e2* must be non-unique, and we can use *shared_function* to distribute the assumptions to type *e2* to both branches. Finally, the case for abstraction uses *split_dom_inv*, *exchange* and *simplify_rng* (and we make sure to include the assumption about the bound variable of the abstraction when using the induction hypothesis).

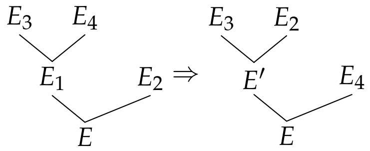
Preservation for evaluation rule *red_value*.
Lemma *preservation_value* : ∀ *L M N*,
  *term* (*lt trm_abs M* in *N*) →
  (∀ *x* : *S.elt*, *x* \notin *L* → *evals* (*N* ˆ *x*) *x*) →
  ∀ *E T fvars*,
    (*E* ⊢ *lt trm_abs M* in *N* : *T* | *fvars*) →
    (*E* ⊢ *N* ˆˆ *trm_abs M* : *T* | *fvars*).
Follows from *substitution* and *eval_fv*.

Preservation for evaluation rule *red_commute*.
Lemma *preservation_commute* : ∀ *L M A N*,
  *term* (*trm_app* (*lt M* in *A*) *N*) →
  (∀ *x* : *S.elt*, *x* \notin *L* → *answer* (*A* ˆ *x*)) →
  ∀ *E T fvars*,
    (*E* ⊢ *trm_app* (*lt M* in *A*) *N* : *T* | *fvars*) →
    (*E* ⊢ *lt M* in *trm_app A N* : *T* | *fvars*).

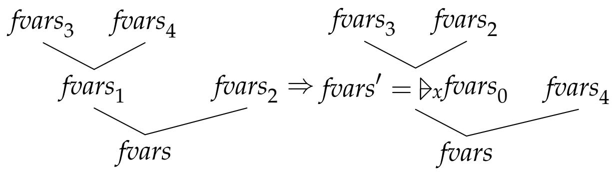
This and the next lemma are mainly a matter of re-ordering the assumptions in the environments $E$ and *fvars* in a useful way. Graphically, what we want is



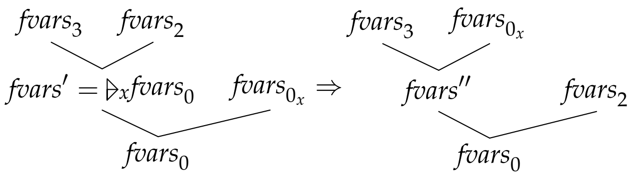
The ordering of $E$ is straightforward:




but the reordering of *fvars* is slightly more involved. We have




Here, the equality on *fvars'* comes from the premise of the abstraction rule. In addition, we can use *split_dom_inv* to get

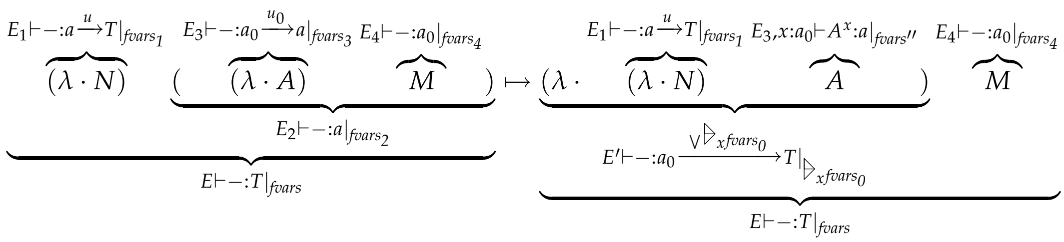



Together with *split_empty_inv*, that is sufficient to prove the lemma.

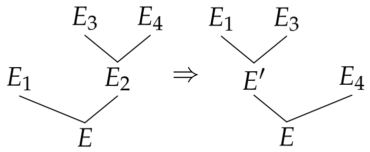
Preservation for evaluation rule *red_assoc*.

```
Lemma preservation_assoc : ∀ L M A N,
  term (lt lt M in A in N) →
  (∀ x : S.elt, x \notin L → answer (A ^ x)) →
  (∀ x : S.elt, x \notin L → evals (N ^ x) x) →
  ∀ E T fvars,
    (E ⊢ lt lt M in A in N : T | fvars) →
    (E ⊢ lt M in (lt A in N) : T | fvars).
```

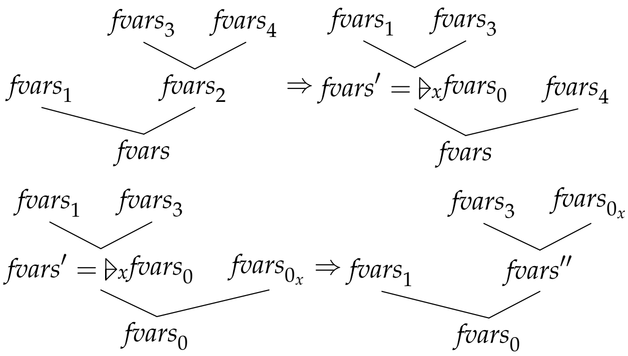
Like in the previous lemma, proving this lemma is mainly a matter of reordering the environments. The following diagram shows roughly what we're trying to achieve:



Also, as for the last lemma, the reordering on *E* is straightforward,




but the ordering on *fvars* is again slightly more involved:




Preservation for evaluation rule *red_closure_app*.
Lemma *preservation_closure_app* : ∀ *E E' M*,
  *term* (*trm_app E M*) →
  *red E E'* →
  (∀ (*E0* : *env*) (*T* : *typ*) (*fvars* : *env*),
      *E0* ⊢ *E* : *T* | *fvars* → *E0* ⊢ *E'* : *T* | *fvars*) →
  ∀ *E0 T fvars*,
    (*E0* ⊢ *trm_app E M* : *T* | *fvars*) →
    (*E0* ⊢ *trm_app E' M* : *T* | *fvars*).
Trivial.

Preservation for evaluation rule *red_closure_let*.
Lemma *preservation_closure_let* : ∀ *L E E' M*,
  *term* (*lt M* in *E*) →
  (∀ *x* : *S.elt*, *x* \notin *L* → *red* (*E* ˆ *x*) (*E'* ˆ *x*)) →
  (∀ *x* : *S.elt*,
    *x* \notin *L* →
    ∀ (*E0* : *env*) (*T* : *typ*) (*fvars* : *env*),
    *E0* ⊢ *E* ˆ *x* : *T* | *fvars* → *E0* ⊢ *E'* ˆ *x* : *T* | *fvars*) →
  ∀ *E0 T fvars*,
    (*E0* ⊢ *lt M* in *E* : *T* | *fvars*) →
    (*E0* ⊢ *lt M* in *E'* : *T* | *fvars*).
Trivial.

Preservation for evaluation rule *red_closure_dem*.
Lemma *preservation_closure_dem* : ∀ *L E0 E0' E1*,
  *term* (*lt E0* in *E1*) →
  *red E0 E0'* →
  (∀ (*E* : *env*) (*T* : *typ*) (*fvars* : *env*),
      *E* ⊢ *E0* : *T* | *fvars* → *E* ⊢ *E0'* : *T* | *fvars*) →
  (∀ *x* : *S.elt*, *x* \notin *L* → *evals* (*E1* ˆ *x*) *x*) →
  ∀ *E T fvars*,
    (*E* ⊢ *lt E0* in *E1* : *T* | *fvars*) →

($E \vdash lt\ E0'$ in $E1 : T \mid fvars$).
Trivial.

If $e$ has type $T$ and $e$ reduces to $e'$, then $e'$ will also have type $T$.
Theorem *preservation* : $\forall\ e\ e',\ red\ e\ e' \rightarrow$
$\forall\ E\ T\ fvars,\ E \vdash e : T \mid fvars \rightarrow E \vdash e' : T \mid fvars$.
Follows trivially by induction on $E \vdash e : T$ from the preceding preservation lemmas.

# A Boolean algebra

This formalization is based on the second chapter ("The self-dual system of axioms") in Goodstein's book "Boolean Algebra" [9].

## A.1 Abstraction over the structure of terms

```
Module Type BooleanAlgebraTerm.

Parameter trm : Set.
Parameter true : trm.
Parameter false : trm.
Parameter or : trm → trm → trm.
Parameter and : trm → trm → trm.
Parameter not : trm → trm.

End BooleanAlgebraTerm.
```

## A.2 Huntington's postulates

```
Module BooleanAlgebra (Term : BooleanAlgebraTerm).
Import Term.

Inductive equiv : trm → trm → Prop :=
   (** Commutativity *)
  | comm_or : ∀ (a b:trm), equiv (or a b) (or b a)
  | comm_and : ∀ (a b:trm), equiv (and a b) (and b a)
   (** Distributivity *)
  | distr_or : ∀ (a b c:trm), equiv (or a (and b c)) (and (or a b) (or a c))
  | distr_and : ∀ (a b c:trm), equiv (and a (or b c)) (or (and a b) (and a c))
   (** Identities *)
  | id_or : ∀ (a:trm), equiv (or a false) a
  | id_and : ∀ (a:trm), equiv (and a true) a
   (** Complements *)
  | compl_or : ∀ (a:trm), equiv (or a (not a)) true
  | compl_and : ∀ (a:trm), equiv (and a (not a)) false
   (** Closure *)
  | clos_not : ∀ (a b:trm), equiv a b → equiv (not a) (not b)
  | clos_or : ∀ (a b c:trm), equiv a b → equiv (or a c) (or b c)
  | clos_and : ∀ (a b c:trm), equiv a b → equiv (and a c) (and b c)
   (** Structural rules *)
  | refl : ∀ (a:trm), equiv a a
  | sym : ∀ (a b:trm), equiv a b → equiv b a
  | trans : ∀ (a b c:trm), equiv a b → equiv b c → equiv a c.
```

## A.3 Setup for Coq setoids

Thanks to Adam Megacz.

```
Add Relation trm equiv
  reflexivity proved by refl
  symmetry proved by sym
  transitivity proved by trans
```

as *equiv_relation*.

*Add Morphism or*
with *signature equiv* ==> *equiv* ==> *equiv*
as *or_morphism*.

*Add Morphism and*
with *signature equiv* ==> *equiv* ==> *equiv*
as *and_morphism*.

*Add Morphism not*
with *signature equiv* ==> *equiv*
as *not_morphism*.

## A.4 Derived Properties

Lemma *false_unique* : ∀ (*x*:*trm*), (∀ (*a*:*trm*), *equiv* (*or a x*) *a*) → *equiv false x*.

Lemma *true_unique* : ∀ (*y*:*trm*), (∀ (*a*:*trm*), *equiv* (*and a y*) *a*) → *equiv true y*.

Lemma *complement_unique* : ∀ (*a a' a"*:*trm*),
(** if *a' has the property of the complement* *)
*equiv* (*or a a'*) *true* → *equiv* (*and a a'*) *false* →
(** *and so does a"* *)
*equiv* (*or a a"*) *true* → *equiv* (*and a a"*) *false* →
(** then *a' and a" must be equivalent* *)
*equiv a' a"*.

Lemma *involution* : ∀ (*a*:*trm*), *equiv* (*not* (*not a*)) *a*.

Lemma *true_compl_false* : *equiv false* (*not true*).

Lemma *false_compl_true* : *equiv* (*not false*) *true*.

Lemma *zero_or* : ∀ (*a*:*trm*), *equiv* (*or a true*) *true*.

Lemma *zero_and* : ∀ (*a*:*trm*), *equiv* (*and a false*) *false*.

Lemma *idem_or* : ∀ (*a*:*trm*), *equiv a* (*or a a*).

Lemma *idem_and* : ∀ (*a*:*trm*), *equiv a* (*and a a*).

Lemma *abs_or* : ∀ (*a b*:*trm*), *equiv* (*or a* (*and a b*)) *a*.

Lemma *abs_and* : ∀ (*a b*:*trm*), *equiv* (*and a* (*or a b*)) *a*.

Lemma *equiv_or_and3* : ∀ (*a b c*:*trm*),
*equiv* (*or a b*) (*or a c*) → *equiv* (*and a b*) (*and a c*) → *equiv b c*.

Lemma *equiv_or_not* : ∀ (*a b c*:*trm*),
*equiv* (*or a b*) (*or a c*) → *equiv* (*or* (*not a*) *b*) (*or* (*not a*) *c*) → *equiv b c*.

Lemma *equiv_and_not* : ∀ (*a b c*:*trm*),
*equiv* (*and a b*) (*and a c*) → *equiv* (*and* (*not a*) *b*) (*and* (*not a*) *c*) → *equiv b c*.

Lemma *assoc_or* : ∀ (*a b c*:*trm*), *equiv* (*or a* (*or b c*)) (*or* (*or a b*) *c*).

Lemma *assoc_and* : ∀ (*a b c*:*trm*), *equiv* (*and a* (*and b c*)) (*and* (*and a b*) *c*).

Lemma *equiv_or_and2* : ∀ (*a b*:*trm*), *equiv* (*or a b*) (*and a b*) → *equiv a b*.

Lemma *DeMorgan_or* : ∀ (*a b*:*trm*), *equiv* (*not* (*or a b*)) (*and* (*not a*) (*not b*)).

Lemma *DeMorgan_and* : ∀ (*a b*:*trm*), *equiv* (*not* (*and a b*)) (*or* (*not a*) (*not b*)).

## A.5 "Non-standard" properties (not proven in Goodstein)

Lemma *abs_or_or* : ∀ (*a b*:*trm*), *equiv* (*or* (*or a b*) *a*) (*or a b*).

Lemma *abs_and_and* : ∀ (*a b*:*trm*), *equiv* (*and* (*and a b*) *a*) (*and a b*).

Lemma *distr_or_or* : ∀ *a b c*, *equiv* (*or a* (*or b c*)) (*or* (*or a b*) (*or a c*)).

Lemma *distr_and_and* : ∀ *a b c*, *equiv* (*and a* (*and b c*)) (*and* (*and a b*) (*and a c*)).

Lemma *or_false_left* : ∀ (*a b*:*trm*), *equiv* (*or a b*) *false* → *equiv a false*.

Lemma *or_false_right* : ∀ (*a b*:*trm*), *equiv* (*or a b*) *false* → *equiv b false*.

Lemma *or_false_both* : ∀ (*a b*:*trm*),
*equiv* (*or a b*) *false* → *equiv a false* ∧ *equiv b false*.

Lemma *and_true_left* : ∀ (*a b*:*trm*), *equiv* (*and a b*) *true* → *equiv a true*.

Lemma *and_true_right* : ∀ (*a b*:*trm*), *equiv* (*and a b*) *true* → *equiv b true*.

Lemma *and_true_both* : ∀ (*a b*:*trm*),
*equiv* (*and a b*) *true* → *equiv a true* ∧ *equiv b true*.

## A.6 Conditional

Definition *ifbool* (*b P Q*:*trm*) : *trm* := *or* (*and b P*) (*and* (*not b*) *Q*).

Lemma *if_ident_branch* : ∀ (*b P*:*trm*),
*equiv* (*ifbool b P P*) *P*.

Lemma *distr_or_if* : ∀ (*b P Q R*:*trm*),
*equiv* (*or* (*ifbool b P Q*) *R*) (*ifbool b* (*or P R*) (*or Q R*)).

Lemma *distr_or_if2* : ∀ (*b P Q*:*trm*),
*equiv* (*ifbool b P Q*) (*or* (*ifbool b P Q*) (*and P Q*)).

Lemma *distr_and_if* : ∀ (*b P Q R*:*trm*),
*equiv* (*and* (*ifbool b P Q*) *R*) (*ifbool b* (*and P R*) (*and Q R*)).

Lemma *distr_not_if* : ∀ (*b P Q*:*trm*),
*equiv* (*not* (*ifbool b P Q*)) (*ifbool b* (*not P*) (*not Q*)).

End *BooleanAlgebra*.

## References


[1] AYDEMIR, B., CHARGUÉRAUD, A., PIERCE, B. C., POLLACK, R., AND WEIRICH, S. Engineering formal metatheory. *SIGPLAN Not. 43*, 1 (2008), 3–15.

[2] BARENDREGT, H. P. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, 1984.

[3] BERTOT, Y., AND CASTERAN, P. *Interactive Theorem Proving and Program Development (Coq'Art: The Calculus of Inductive Constructions)*. Springer-Verlag, 2004.

[4] BIERNACKA, M., AND BIERNACKI, D. Formalizing constructions of abstract machines for functional languages in Coq. In *Informal Proceedings of the 7th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2007)* (June 2007).

[5] CERVESATO, I., AND PFENNING, F. A linear logical framework. *Inf. Comput. 179*, 1 (2002), 19–75.

[6] CHARGUÉRAUD, A. Formal PL metatheory: Locally nameless developments (Coq development), 2007. `http://www.chargueraud.org/arthur/research/2007/binders`.

[7] DE VRIES, E., PLASMEIJER, R., AND ABRAHAMSON, D. M. Uniqueness typing simplified. In *Implementation and Application of Functional Languages* (2008), vol. 5083/2008 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 181–198.

[8] DUBOIS, C. Proving ML type soundness within Coq. In *TPHOLs '00: Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics* (London, UK, 2000), Springer-Verlag, pp. 126–144. Published version is incorrect; corrected version available from the author's website.

[9] GOODSTEIN, R. L. *Boolean Algebra*. Dover Publications, Inc, 2007. Unabridged republicaton of the 1966 printing of the work originally published by Pergamon Press, London, in 1963.

[10] HUNTINGTON, E. V. Sets of independent postulates for the algebra of logic. *Transactions of the American Mathematical Society 5* (1904), 288–309.

[11] MARAIST, J., ODERSKY, M., AND WADLER, P. The call-by-need lambda calculus. *J. Funct. Program. 8*, 3 (1998), 275–317.

[12] PITTS, A. M. Nominal logic: A first order theory of names and binding. In *TACS '01: Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software* (London, UK, 2001), Springer-Verlag, pp. 219–242.

[13] WALKER, D. Substructural type systems. In *Advanced Topics in Types and Programming Languages*, B. Pierce, Ed. The MIT Press, 2005.