An Algebra for Higher-Order Terms

Lars Hupel

December 14, 2021

Abstract

In this formalization, I introduce a higher-order term algebra, generalizing the notions of free variables, matching, and substitution. The need arose from the work on a verified compiler from Isabelle to CakeML [3]. Terms can be thought of as consisting of a *generic* (free variables, constants, application) and a *specific* part. As example applications, this entry provides instantiations for de-Bruijn terms, terms with named variables, and Blanchette's λ-free higher-order terms [1]. Furthermore, I implement translation functions between de-Bruijn terms and named terms and prove their correctness.

Contents

Chapter 1

Names as a unique datatype

theory *Name*
imports *Main*
begin

I would like to model names as *string*s. Unfortunately, there is no default order on lists, as there could be multiple reasonable implementations: e.g. lexicographic and point-wise. For both choices, users can import the corresponding instantiation.

In Isabelle, only at most one implementation of a given type class for a given type may be present in the same theory. Consequently, I avoided importing a list ordering from the library, because it may cause conflicts with users who use another ordering. The general approach for these situations is to introduce a type copy.

The full flexibility of strings (i.e. string manipulations) is only required where fresh names are being produced. Otherwise, only a linear order on terms is needed. Conveniently, Sternagel and Thiemann [5] provide tooling to automatically generate such a lexicographic order.

datatype *name* = *Name* (*as-string*: *string*)

— Mostly copied from *List-Lexorder*

instantiation *name* :: *ord*
begin

definition *less-name* **where**
$xs < ys \longleftrightarrow (as\text{-}string\ xs,\ as\text{-}string\ ys) \in lexord\ \{(u,\ v).\ (of\text{-}char\ u :: nat) < of\text{-}char\ v\}$

definition *less-eq-name* **where**
$(xs :: name) \leq ys \longleftrightarrow xs < ys \vee xs = ys$

instance ..

end

instance *name* :: *order*
proof
  fix *xs* :: *name*
  show *xs* ≤ *xs* **by** (*simp add*: *less-eq-name-def*)
next
  fix *xs ys zs* :: *name*
  assume *xs* ≤ *ys* **and** *ys* ≤ *zs*
  then show *xs* ≤ *zs*
    apply (*auto simp add*: *less-eq-name-def less-name-def*)
    apply (*rule lexord-trans*)
    apply (*auto intro*: *transI*)
    done
next
  fix *xs ys* :: *name*
  assume *xs* ≤ *ys* **and** *ys* ≤ *xs*
  then show *xs* = *ys*
    apply (*auto simp add*: *less-eq-name-def less-name-def*)
    apply (*rule lexord-irreflexive* [*THEN notE*])
    defer
    apply (*rule lexord-trans*)
    apply (*auto intro*: *transI*)
    done
next
  fix *xs ys* :: *name*
  show *xs* < *ys* ⟷ *xs* ≤ *ys* ∧ ¬ *ys* ≤ *xs*
    apply (*auto simp add*: *less-name-def less-eq-name-def*)
    defer
    apply (*rule lexord-irreflexive* [*THEN notE*])
    apply *auto*
    apply (*rule lexord-irreflexive* [*THEN notE*])
    defer
    apply (*rule lexord-trans*)
    apply (*auto intro*: *transI*)
    done
qed

instance *name* :: *linorder*
proof
  fix *xs ys* :: *name*
  have (*as-string xs*, *as-string ys*) ∈ *lexord* {(*u*, *v*). (*of-char u*::*nat*) < *of-char v*} ∨ *xs* = *ys* ∨ (*as-string ys*, *as-string xs*) ∈ *lexord* {(*u*, *v*). (*of-char u*::*nat*) < *of-char v*}
   by (*metis* (*no-types*, *lifting*) *case-prodI lexord-linear linorder-neqE-nat mem-Collect-eq name.expand of-char-eq-iff*)
  then show *xs* ≤ *ys* ∨ *ys* ≤ *xs*
    by (*auto simp add*: *less-eq-name-def less-name-def*)

qed

lemma *less-name-code*[*code*]:
Name xs < *Name* [] ⟷ *False*
Name [] < *Name* (*x* # *xs*) ⟷ *True*
Name (*x* # *xs*) < *Name* (*y* # *ys*) ⟷ (*of-char x*::*nat*) < *of-char y* ∨ *x* = *y* ∧ *Name xs* < *Name ys*
unfolding *less-name-def* **by** *auto*

lemma *le-name-code*[*code*]:
Name (*x* # *xs*) ≤ *Name* [] ⟷ *False*
Name [] ≤ *Name* (*x* # *xs*) ⟷ *True*
Name (*x* # *xs*) ≤ *Name* (*y* # *ys*) ⟷ (*of-char x*::*nat*) < *of-char y* ∨ *x* = *y* ∧ *Name xs* ≤ *Name ys*
unfolding *less-eq-name-def less-name-def* **by** *auto*

context begin

qualified definition *append* :: *name* ⇒ *name* ⇒ *name* **where**
append v1 v2 = *Name* (*as-string v1* @ *as-string v2*)

lemma *name-append-less*:
assumes *xs* ≠ *Name* []
shows *append ys xs* > *ys*
proof −
have *Name* (*ys* @ *xs*) > *Name ys* **if** *xs* ≠ [] **for** *xs ys*
using *that*
proof (*induction ys*)
case *Nil*
thus *?case*
unfolding *less-name-def*
by (*cases xs*) *auto*
next
case (*Cons y ys*)
thus *?case*
unfolding *less-name-def*
by *auto*
qed
with *assms* **show** *?thesis*
unfolding *append-def*
by (*cases xs*, *cases ys*) *auto*
qed

end

end

Chapter 2

A monad for generating fresh names

theory *Fresh-Monad*
imports
HOL−Library.State-Monad
Term-Utils
begin

Generation of fresh names in general can be thought of as picking a string that is not an element of a (finite) set of already existing names. For Isabelle, the *Nominal* framework [7, 8] provides support for reasoning over fresh names, but unfortunately, its definitions are not executable.

Instead, I chose to model generation of fresh names as a monad based on *state*. With this, it becomes possible to write programs using *do*-notation. This is implemented abstractly as a **locale** that expects two operations:

- *next* expects a value and generates a larger value, according to *linorder*
- *arb* produces any value, similarly to *undefined*, but executable

locale *fresh* =
fixes *next* :: *'a*::*linorder* ⇒ *'a* **and** *arb* :: *'a*
assumes *next-ge*: *next x* > *x*
begin

abbreviation *update-next* :: (*'a*, *unit*) *state* **where**
update-next ≡ *State-Monad.update next*

lemma *update-next-strict-mono*[*simp*, *intro*]: *strict-mono-state update-next*
using *next-ge* **by** (*auto intro*: *update-strict-mono*)

lemma *update-next-mono*[*simp*, *intro*]: *mono-state update-next*
by (*rule strict-mono-implies-mono*) (*rule update-next-strict-mono*)

definition *create* :: ('*a*, '*a*) *state* **where**
create = *update-next* ⤜ (λ-. *State-Monad.get*)

lemma *create-alt-def*[*code*]: *create* = *State* (λ*a*. (*next a*, *next a*))
unfolding *create-def State-Monad.update-def State-Monad.get-def State-Monad.set-def State-Monad.bind-def*
by *simp*

abbreviation *fresh-in* :: '*a set* ⇒ '*a* ⇒ *bool* **where**
fresh-in S s ≡ *Ball S* ((≥) *s*)

lemma *next-ge-all*: *finite S* ⟹ *fresh-in S s* ⟹ *next s* ∉ *S*
by (*metis antisym less-imp-le less-irrefl next-ge*)

definition *Next* :: '*a set* ⇒ '*a* **where**
Next S = (*if S* = {} *then arb else next* (*Max S*))

lemma *Next-ge-max*: *finite S* ⟹ *S* ≠ {} ⟹ *Next S* > *Max S*
unfolding *Next-def* **using** *next-ge* **by** *simp*

lemma *Next-not-member-subset*: *finite S'* ⟹ *S* ⊆ *S'* ⟹ *Next S'* ∉ *S*
unfolding *Next-def* **using** *next-ge*
by (*metis Max-ge Max-mono empty-iff finite-subset leD less-le-trans subset-empty*)

lemma *Next-not-member*: *finite S* ⟹ *Next S* ∉ *S*
by (*rule Next-not-member-subset*) *auto*

lemma *Next-geq-not-member*: *finite S* ⟹ *s* ≥ *Next S* ⟹ *s* ∉ *S*
unfolding *Next-def* **using** *next-ge*
by (*metis* (*full-types*) *Max-ge all-not-in-conv leD le-less-trans*)

lemma *next-not-member*: *finite S* ⟹ *s* ≥ *Next S* ⟹ *next s* ∉ *S*
by (*meson Next-geq-not-member less-imp-le next-ge order-trans*)

lemma *create-mono*[*simp*, *intro*]: *mono-state create*
unfolding *create-def*
by (*auto intro*: *bind-mono-strong*)

lemma *create-strict-mono*[*simp*, *intro*]: *strict-mono-state create*
unfolding *create-def*
by (*rule bind-strict-mono-strong2*) *auto*

abbreviation *run-fresh* **where**
run-fresh m S ≡ *fst* (*run-state m* (*Next S*))

abbreviation *fresh-fin* :: '*a fset* ⇒ '*a* ⇒ *bool* **where**
fresh-fin S s ≡ *fBall S* ((≥) *s*)

context includes *fset.lifting* **begin**

lemma *next-ge-fall*: *fresh-fin S s* $\Longrightarrow$ *next s* $|\notin|$ *S*
by (*transfer fixing*: *next*) (*rule next-ge-all*)

lift-definition *fNext* :: *'a fset* $\Rightarrow$ *'a* **is** *Next* **.**

lemma *fNext-ge-max*: *S* $\neq$ {||} $\Longrightarrow$ *fNext S* $>$ *fMax S*
by *transfer* (*rule Next-ge-max*)

lemma *next-not-fmember*: *s* $\geq$ *fNext S* $\Longrightarrow$ *next s* $|\notin|$ *S*
by *transfer* (*rule next-not-member*)

lemma *fNext-geq-not-member*: *s* $\geq$ *fNext S* $\Longrightarrow$ *s* $|\notin|$ *S*
by *transfer* (*rule Next-geq-not-member*)

lemma *fNext-not-member*: *fNext S* $|\notin|$ *S*
by *transfer* (*rule Next-not-member*)

lemma *fNext-not-member-subset*: *S* $|\subseteq|$ *S'* $\Longrightarrow$ *fNext S'* $|\notin|$ *S*
by *transfer* (*rule Next-not-member-subset*)

abbreviation *frun-fresh* **where**
frun-fresh m S $\equiv$ *fst* (*run-state m* (*fNext S*))

end

end

end

2.1 Fresh monad operations as class operations

theory *Fresh-Class*
imports
Fresh-Monad
Name
begin

The *fresh* locale allows arbitrary instantiations. However, this may be inconvenient to use. The following class serves as a global instantiation that can be used without interpretation. The *arb* parameter of the locale redirects to *default*.

Some instantiations are provided. For *name*s, underscores are appended to generate a fresh name.

class *fresh* = *linorder* + *default* +
fixes *next* :: *'a* $\Rightarrow$ *'a*
assumes *next-ge*: *next x* $>$ *x*

global-interpretation *Fresh-Monad.fresh next default*
  defines *fresh-create* = *create*
    and *fresh-Next* = *Next*
    and *fresh-fNext* = *fNext*
    and *fresh-frun* = *frun-fresh*
    and *fresh-run* = *run-fresh*
proof
  show $x <$ *next* x **for** x **by** (*rule next-ge*)
qed

lemma [*code*]: *fresh-frun m S* = *fst* (*run-state m* (*fresh-fNext S*))
by (*simp add*: *fresh-fNext-def fresh-frun-def*)

lemma [*code*]: *fresh-run m S* = *fst* (*run-state m* (*fresh-Next S*))
by (*simp add*: *fresh-Next-def fresh-run-def*)

instantiation *nat* :: *fresh* **begin**

definition *default-nat* :: *nat* **where**
default-nat = *0*

definition *next-nat* **where**
next-nat = *Suc*

instance
by *intro-classes* (*auto simp*: *next-nat-def*)

end

instantiation *char* :: *default*
begin

definition *default-char* :: *char* **where**
default-char = *CHR* ''-''

instance ..

end

instantiation *name* :: *fresh* **begin**

definition *default-name* **where**
default-name = *Name* ''-''

definition *next-name* **where**
next-name xs = *Name.append xs default*

instance proof

fix v :: *name*
show $v <$ *next* v
unfolding *next-name-def default-name-def*
by (*rule name-append-less*) *simp*
qed

end

export-code
fresh-create fresh-Next fresh-fNext fresh-frun fresh-run
checking *Scala? SML?*

end

Chapter 3

Terms

theory *Term-Class*
imports
Datatype-Order-Generator.Order-Generator
Name
Term-Utils
HOL−Library.Disjoint-FSets
begin

hide-type (**open**) *term*

3.1 A simple term type, modelled after Pure's *term* type

datatype *term* =
Const name |
Free name |
Abs term (Λ - [*71*] *71*) |
Bound nat |
App term term (**infixl** $ *70*)

derive *linorder term*

3.2 A type class describing terms

The type class is split into two parts, *pre-terms* and *terms*. The only difference is that terms assume more axioms about substitution (see below).

A term must provide the following generic constructors that behave like regular free constructors:

- *const* :: *name* ⇒ τ
- *free* :: *name* ⇒ τ

- *app* :: $\tau \Rightarrow \tau \Rightarrow \tau$

Conversely, there are also three corresponding destructors that could be defined in terms of Hilbert's choice operator. However, I have instead opted to let instances define destructors directly, which is simpler for execution purposes.

Besides the generic constructors, terms may also contain other constructors. Those are abstractly called *abstractions*, even though that name is not entirely accurate (bound variables may also fall under this).

Additionally, there must be operations that compute the list of all free variables (*frees*), constants (*consts*), and substitutions (*subst*). Pre-terms only assume some basic properties of substitution on the generic constructors.

Most importantly, substitution is not specified for environments containing terms with free variables. Term types are not required to implement α-renaming to prevent capturing of variables.

class *pre-term* = *size* +
  fixes
    frees :: $'a \Rightarrow$ *name fset* **and**
    subst :: $'a \Rightarrow$ (*name*, $'a$) *fmap* $\Rightarrow$ $'a$ **and**
    consts :: $'a \Rightarrow$ *name fset*
  fixes
    app :: $'a \Rightarrow 'a \Rightarrow 'a$ **and** *unapp* :: $'a \Rightarrow ('a \times 'a)$ *option*
  fixes
    const :: *name* $\Rightarrow$ $'a$ **and** *unconst* :: $'a \Rightarrow$ *name option*
  fixes
    free :: *name* $\Rightarrow$ $'a$ **and** *unfree* :: $'a \Rightarrow$ *name option*
  assumes *unapp-app*[*simp*]: *unapp* (*app* u_1 u_2) = *Some* (u_1, u_2)
  assumes *app-unapp*[*dest*]: *unapp* u = *Some* (u_1, u_2) $\Longrightarrow$ u = *app* u_1 u_2
  assumes *app-size*[*simp*]: *size* (*app* u_1 u_2) = *size* u_1 + *size* u_2 + *1*
  assumes *unconst-const*[*simp*]: *unconst* (*const name*) = *Some name*
  assumes *const-unconst*[*dest*]: *unconst* u = *Some name* $\Longrightarrow$ u = *const name*
  assumes *unfree-free*[*simp*]: *unfree* (*free name*) = *Some name*
  assumes *free-unfree*[*dest*]: *unfree* u = *Some name* $\Longrightarrow$ u = *free name*
  assumes *app-const-distinct*: *app* u_1 u_2 $\neq$ *const name*
  assumes *app-free-distinct*: *app* u_1 u_2 $\neq$ *free name*
  assumes *free-const-distinct*: *free* $name_1$ $\neq$ *const* $name_2$
  assumes *frees-const*[*simp*]: *frees* (*const name*) = *fempty*
  assumes *frees-free*[*simp*]: *frees* (*free name*) = {| *name* |}
  assumes *frees-app*[*simp*]: *frees* (*app* u_1 u_2) = *frees* u_1 |∪| *frees* u_2
  assumes *consts-free*[*simp*]: *consts* (*free name*) = *fempty*
  assumes *consts-const*[*simp*]: *consts* (*const name*) = {| *name* |}
  assumes *consts-app*[*simp*]: *consts* (*app* u_1 u_2) = *consts* u_1 |∪| *consts* u_2
  assumes *subst-app*[*simp*]: *subst* (*app* u_1 u_2) *env* = *app* (*subst* u_1 *env*) (*subst* u_2 *env*)
  assumes *subst-const*[*simp*]: *subst* (*const name*) *env* = *const name*
  assumes *subst-free*[*simp*]: *subst* (*free name*) *env* = (*case fmlookup env name of Some t* $\Rightarrow$ *t* | - $\Rightarrow$ *free name*)

assumes *free-inject*: $free\ name_1 = free\ name_2 \Longrightarrow name_1 = name_2$
assumes *const-inject*: $const\ name_1 = const\ name_2 \Longrightarrow name_1 = name_2$
assumes *app-inject*: $app\ u_1\ u_2 = app\ u_3\ u_4 \Longrightarrow u_1 = u_3 \wedge u_2 = u_4$

instantiation *term* :: *pre-term* **begin**

definition *app-term* **where**
app-term t u = *t* $ *u*

fun *unapp-term* **where**
unapp-term (*t* $ *u*) = *Some* (*t*, *u*) |
unapp-term - = *None*

definition *const-term* **where**
const-term = *Const*

fun *unconst-term* **where**
unconst-term (*Const name*) = *Some name* |
unconst-term - = *None*

definition *free-term* **where**
free-term = *Free*

fun *unfree-term* **where**
unfree-term (*Free name*) = *Some name* |
unfree-term - = *None*

fun *frees-term* :: *term* ⇒ *name fset* **where**
frees-term (*Free x*) = {| *x* |} |
frees-term (t_1 $ t_2) = *frees-term* t_1 |∪| *frees-term* t_2 |
frees-term (Λ *t*) = *frees-term t* |
frees-term - = {||}

fun *subst-term* :: *term* ⇒ (*name*, *term*) *fmap* ⇒ *term* **where**
subst-term (*Free s*) *env* = (*case fmlookup env s of Some t* ⇒ *t* | *None* ⇒ *Free s*) |
subst-term (t_1 $ t_2) *env* = *subst-term* t_1 *env* $ *subst-term* t_2 *env* |
subst-term (Λ *t*) *env* = Λ *subst-term t env* |
subst-term t env = *t*

fun *consts-term* :: *term* ⇒ *name fset* **where**
consts-term (*Const x*) = {| *x* |} |
consts-term (t_1 $ t_2) = *consts-term* t_1 |∪| *consts-term* t_2 |
consts-term (Λ *t*) = *consts-term t* |
consts-term - = {||}

instance
by *standard*
(*auto*
simp: *app-term-def const-term-def free-term-def*

elim: *unapp-term.elims unconst-term.elims unfree-term.elims*
split: *option.splits*)

end

context *pre-term* **begin**

definition *freess* :: *'a list* ⇒ *name fset* **where**
freess = *ffUnion* ∘ *fset-of-list* ∘ *map frees*

lemma *freess-cons*[*simp*]: *freess* (*x* # *xs*) = *frees x* |∪| *freess xs*
unfolding *freess-def* **by** *simp*

lemma *freess-single*: *freess* [*x*] = *frees x*
unfolding *freess-def* **by** *simp*

lemma *freess-empty*[*simp*]: *freess* [] = {||}
unfolding *freess-def* **by** *simp*

lemma *freess-app*[*simp*]: *freess* (*xs* @ *ys*) = *freess xs* |∪| *freess ys*
unfolding *freess-def* **by** *simp*

lemma *freess-subset*: *set xs* ⊆ *set ys* ⟹ *freess xs* |⊆| *freess ys*
unfolding *freess-def comp-apply*
by (*intro ffunion-mono fset-of-list-subset*) *auto*

abbreviation *id-env* :: (*name*, *'a*) *fmap* ⇒ *bool* **where**
id-env ≡ *fmpred* (λ*x y*. *y* = *free x*)

definition *closed-except* :: *'a* ⇒ *name fset* ⇒ *bool* **where**
closed-except t S ⟷ *frees t* |⊆| *S*

abbreviation *closed* :: *'a* ⇒ *bool* **where**
closed t ≡ *closed-except t* {||}

lemmas *term-inject* = *free-inject const-inject app-inject*

lemmas *term-distinct*[*simp*] =
app-const-distinct app-const-distinct[*symmetric*]
app-free-distinct app-free-distinct[*symmetric*]
free-const-distinct free-const-distinct[*symmetric*]

lemma *app-size1*: *size* u_1 < *size* (*app* u_1 u_2)
by *simp*

lemma *app-size2*: *size* u_2 < *size* (*app* u_1 u_2)
by *simp*

lemma *unx-some-lemmas*:

unapp u = Some x ⟹ unconst u = None
unapp u = Some x ⟹ unfree u = None
unconst u = Some y ⟹ unapp u = None
unconst u = Some y ⟹ unfree u = None
unfree u = Some z ⟹ unconst u = None
unfree u = Some z ⟹ unapp u = None
subgoal by (*metis app-unapp const-unconst app-const-distinct not-None-eq surj-pair*)
subgoal by (*metis app-free-distinct app-unapp free-unfree option.exhaust surj-pair*)
subgoal by (*metis app-unapp const-unconst app-const-distinct old.prod.exhaust option.distinct*(*1*) *option.expand option.sel*)
subgoal by (*metis const-unconst free-const-distinct free-unfree option.exhaust*)
subgoal by (*metis const-unconst free-const-distinct free-unfree option.exhaust*)
subgoal by (*metis app-free-distinct app-unapp free-unfree not-Some-eq surj-pair*)
done

lemma *unx-none-simps*[*simp*]:
unapp (*const name*) = *None*
unapp (*free name*) = *None*
unconst (*app t u*) = *None*
unconst (*free name*) = *None*
unfree (*const name*) = *None*
unfree (*app t u*) = *None*
subgoal by (*metis app-unapp app-const-distinct not-None-eq surj-pair*)
subgoal by (*metis app-free-distinct app-unapp option.exhaust surj-pair*)
subgoal by (*metis const-unconst app-const-distinct option.distinct*(*1*) *option.expand option.sel*)
subgoal by (*metis const-unconst free-const-distinct option.exhaust*)
subgoal by (*metis free-const-distinct free-unfree option.exhaust*)
subgoal by (*metis app-free-distinct free-unfree not-Some-eq*)
done

lemma *term-cases*:
obtains (*free*) *name* **where** *t = free name*
| (*const*) *name* **where** *t = const name*
| (*app*) u_1 u_2 **where** *t = app* u_1 u_2
| (*other*) *unfree t = None unapp t = None unconst t = None*
apply (*cases unfree t*)
apply (*cases unconst t*)
apply (*cases unapp t*)
subgoal by *auto*
subgoal for *x* **by** (*cases x*) *auto*
subgoal by *auto*
subgoal by *auto*
done

definition *is-const* **where**
is-const t ⟷ (*unconst t* ≠ *None*)

definition *const-name* **where**

const-name t = (*case unconst t of Some name* ⇒ *name*)

lemma *is-const-simps*[*simp*]:
is-const (*const name*)
¬ *is-const* (*app t u*)
¬ *is-const* (*free name*)
unfolding *is-const-def* **by** *simp+*

lemma *const-name-simps*[*simp*]:
const-name (*const name*) = *name*
is-const t ⟹ *const* (*const-name t*) = *t*
unfolding *const-name-def is-const-def* **by** *auto*

definition *is-free* **where**
is-free t ⟷ (*unfree t* ≠ *None*)

definition *free-name* **where**
free-name t = (*case unfree t of Some name* ⇒ *name*)

lemma *is-free-simps*[*simp*]:
is-free (*free name*)
¬ *is-free* (*const name*)
¬ *is-free* (*app t u*)
unfolding *is-free-def* **by** *simp+*

lemma *free-name-simps*[*simp*]:
free-name (*free name*) = *name*
is-free t ⟹ *free* (*free-name t*) = *t*
unfolding *free-name-def is-free-def* **by** *auto*

definition *is-app* **where**
is-app t ⟷ (*unapp t* ≠ *None*)

definition *left* **where**
left t = (*case unapp t of Some* (*l*, -) ⇒ *l*)

definition *right* **where**
right t = (*case unapp t of Some* (-, *r*) ⇒ *r*)

lemma *app-simps*[*simp*]:
¬ *is-app* (*const name*)
¬ *is-app* (*free name*)
is-app (*app t u*)
unfolding *is-app-def* **by** *simp+*

lemma *left-right-simps*[*simp*]:
left (*app l r*) = *l*
right (*app l r*) = *r*
is-app t ⟹ *app* (*left t*) (*right t*) = *t*

unfolding *is-app-def left-def right-def* **by** *auto*

definition *ids* :: *'a* ⇒ *name fset* **where**
ids t = *frees t* |∪| *consts t*

lemma *closed-except-const*[*simp*]: *closed-except* (*const name*) *S*
unfolding *closed-except-def* **by** *auto*

abbreviation *closed-env* :: (*name*, *'a*) *fmap* ⇒ *bool* **where**
closed-env ≡ *fmpred* (λ-. *closed*)

lemma *closed-except-self*: *closed-except t* (*frees t*)
unfolding *closed-except-def* **by** *simp*

end

class *term* = *pre-term* + *size* +
fixes
abs-pred :: (*'a* ⇒ *bool*) ⇒ *'a* ⇒ *bool*
assumes
raw-induct[*case-names const free app abs*]:
(⋀*name*. *P* (*const name*)) ⟹
(⋀*name*. *P* (*free name*)) ⟹
(⋀*t*$_1$ *t*$_2$. *P t*$_1$ ⟹ *P t*$_2$ ⟹ *P* (*app t*$_1$ *t*$_2$)) ⟹
(⋀*t*. *abs-pred P t*) ⟹
P t
assumes
raw-subst-id: *abs-pred* (λ*t*. ∀ *env*. *id-env env* ⟶ *subst t env* = *t*) *t* **and**
raw-subst-drop: *abs-pred* (λ*t*. *x* |∉| *frees t* ⟶ (∀ *env*. *subst t* (*fmdrop x env*) = *subst t env*)) *t* **and**
raw-subst-indep: *abs-pred* (λ*t*. ∀ *env*$_1$ *env*$_2$. *closed-env env*$_2$ ⟶ *fdisjnt* (*fmdom env*$_1$) (*fmdom env*$_2$) ⟶ *subst t* (*env*$_1$ ++$_f$ *env*$_2$) = *subst* (*subst t env*$_2$) *env*$_1$) *t* **and**
raw-subst-frees: *abs-pred* (λ*t*. ∀ *env*. *closed-env env* ⟶ *frees* (*subst t env*) = *frees t* |−| *fmdom env*) *t* **and**
raw-subst-consts': *abs-pred* (λ*a*. ∀ *x*. *consts* (*subst a x*) = *consts a* |∪| *ffUnion* (*consts* |‘| *fmimage x* (*frees a*))) *t* **and**
abs-pred-trivI: *P t* ⟹ *abs-pred P t*
begin

lemma *subst-id*: *id-env env* ⟹ *subst t env* = *t*
proof (*induction t arbitrary*: *env rule*: *raw-induct*)
case (*abs t*)
show *?case*
by (*rule raw-subst-id*)
qed (*auto split*: *option.splits*)

lemma *subst-drop*: *x* |∉| *frees t* ⟹ *subst t* (*fmdrop x env*) = *subst t env*
proof (*induction t arbitrary*: *env rule*: *raw-induct*)

```
  case (abs t)
  show ?case
    by (rule raw-subst-drop)
qed (auto split: option.splits)

lemma subst-frees: fmpred (λ-. closed) env ⟹ frees (subst t env) = frees t |−| fmdom env
proof (induction t arbitrary: env rule: raw-induct)
  case (abs t)
  show ?case
    by (rule raw-subst-frees)
qed (auto split: option.splits simp: closed-except-def)

lemma subst-consts′: consts (subst t env) = consts t |∪| ffUnion (consts |‘| fmimage env (frees t))
proof (induction t arbitrary: env rule: raw-induct)
  case (free name)
  then show ?case
    by (auto
        split: option.splits
        simp: ffUnion-alt-def fmlookup-ran-iff fmlookup-image-iff fmlookup-dom-iff
        intro!: fBexI)
next
  case (abs t)
  show ?case
    by (rule raw-subst-consts′)
qed (auto simp: funion-image-bind-eq finter-funion-distrib fbind-funion)

fun match :: term ⇒ 'a ⇒ (name, 'a) fmap option where
match (t1 $ t2) u = do {
  (u1, u2) ← unapp u;
  env1 ← match t1 u1;
  env2 ← match t2 u2;
  Some (env1 ++f env2)
} |
match (Const name) u =
  (case unconst u of
    None ⇒ None
  | Some name′ ⇒ if name = name′ then Some fmempty else None) |
match (Free name) u = Some (fmap-of-list [(name, u)]) |
match (Bound n) u = None |
match (Abs t) u = None

lemma match-simps[simp]:
  match (t1 $ t2) (app u1 u2) = do {
    env1 ← match t1 u1;
    env2 ← match t2 u2;
    Some (env1 ++f env2)
  }
```

match (*Const name*) (*const name'*) = (*if name = name' then Some fmempty else None*)
by *auto*

lemma *match-some-induct*[*consumes 1*, *case-names app const free*]:
assumes *match t u = Some env*
assumes $\bigwedge t_1\ t_2\ u_1\ u_2\ env_1\ env_2$. *P* t_1 u_1 $env_1 \Longrightarrow$ *match* t_1 u_1 = *Some* $env_1 \Longrightarrow$ *P* t_2 u_2 $env_2 \Longrightarrow$ *match* t_2 u_2 = *Some* $env_2 \Longrightarrow$ *P* (t_1 $ t_2) (*app* u_1 u_2) (env_1 $++_f$ env_2)
assumes $\bigwedge$*name*. *P* (*Const name*) (*const name*) *fmempty*
assumes $\bigwedge$*name u*. *P* (*Free name*) *u* (*fmupd name u fmempty*)
shows *P t u env*
using *assms*
by (*induction t u arbitrary*: *env rule*: *match.induct*)
(*auto split*: *option.splits if-splits elim*!: *option-bindE*)

lemma *match-dom*: *match p t = Some env* $\Longrightarrow$ *fmdom env = frees p*
by (*induction p arbitrary*: *t env*)
(*fastforce split*: *option.splits if-splits elim*: *option-bindE*)+

lemma *match-vars*: *match p t = Some env* $\Longrightarrow$ *fmpred* (λ- *u*. *frees u* $|\subseteq|$ *frees t*) *env*
proof (*induction p t env rule*: *match-some-induct*)
case (*app* t_1 t_2 u_1 u_2 env_1 env_2)
show *?case*
apply *rule*
using *app*
by (*fastforce intro*: *fmpred-mono-strong*)+
qed *auto*

lemma *match-appE-split*:
assumes *match* (t_1 $ t_2) *u = Some env*
obtains u_1 u_2 env_1 env_2 **where**
u = app u_1 u_2 *match* t_1 u_1 = *Some* env_1 *match* t_2 u_2 = *Some* env_2 *env* = env_1 $++_f$ env_2
using *assms*
by (*auto split*: *option.splits elim*!: *option-bindE*)

lemma *subst-consts*:
assumes *consts t* $|\subseteq|$ *S fmpred* (λ- *u*. *consts u* $|\subseteq|$ *S*) *env*
shows *consts* (*subst t env*) $|\subseteq|$ *S*
apply (*subst subst-consts'*)
using *assms* **by** (*auto intro*!: *ffUnion-least*)

lemma *subst-empty*[*simp*]: *subst t fmempty = t*
by (*auto simp*: *subst-id*)

lemma *subst-drop-fset*: *fdisjnt S* (*frees t*) $\Longrightarrow$ *subst t* (*fmdrop-fset S env*) = *subst t env*

by (*induct S*) (*auto simp*: *subst-drop fdisjnt-alt-def*)

lemma *subst-restrict*:
  assumes *frees t* |⊆| *M*
  shows *subst t* (*fmrestrict-fset M env*) = *subst t env*
proof −
  have ∗: *fmrestrict-fset M env* = *fmdrop-fset* (*fmdom env* − *M*) *env*
    by (*rule fmap-ext*) *auto*

  show *?thesis*
    apply (*subst* ∗)
    apply (*subst subst-drop-fset*)
    unfolding *fdisjnt-alt-def*
    using *assms* **by** *auto*
qed

corollary *subst-restrict′*[*simp*]: *subst t* (*fmrestrict-fset* (*frees t*) *env*) = *subst t env*
by (*simp add*: *subst-restrict*)

corollary *subst-cong*:
  assumes ⋀*x*. *x* |∈| *frees t* ⟹ *fmlookup* Γ$_1$ *x* = *fmlookup* Γ$_2$ *x*
  shows *subst t* Γ$_1$ = *subst t* Γ$_2$
proof −
  have *fmrestrict-fset* (*frees t*) Γ$_1$ = *fmrestrict-fset* (*frees t*) Γ$_2$
    apply (*rule fmap-ext*)
    using *assms* **by** *simp*
  thus *?thesis*
    by (*metis subst-restrict′*)
qed

corollary *subst-add-disjnt*:
  assumes *fdisjnt* (*frees t*) (*fmdom env*$_1$)
  shows *subst t* (*env*$_1$ ++$_f$ *env*$_2$) = *subst t env*$_2$
proof −
  have *subst t* (*env*$_1$ ++$_f$ *env*$_2$) = *subst t* (*fmrestrict-fset* (*frees t*) (*env*$_1$ ++$_f$ *env*$_2$))
    by (*metis subst-restrict′*)
  also have ... = *subst t* (*fmrestrict-fset* (*frees t*) *env*$_1$ ++$_f$ *fmrestrict-fset* (*frees t*) *env*$_2$)
    by *simp*
  also have ... = *subst t* (*fmempty* ++$_f$ *fmrestrict-fset* (*frees t*) *env*$_2$)
    unfolding *fmfilter-alt-defs*
    apply (*subst fmfilter-false*)
    using *assms*
    by (*auto simp*: *fdisjnt-alt-def intro*: *fmdomI*)
  also have ... = *subst t* (*fmrestrict-fset* (*frees t*) *env*$_2$)
    by *simp*
  also have ... = *subst t env*$_2$
    by *simp*

finally show *?thesis* **.**
qed

corollary *subst-add-shadowed-env*:
assumes *frees t* |⊆| *fmdom* env_2
shows *subst t* (env_1 ++$_f$ env_2) = *subst t* env_2
proof −
have *subst t* (env_1 ++$_f$ env_2) = *subst t* (*fmdrop-fset* (*fmdom* env_2) env_1 ++$_f$ env_2)
by (*subst fmadd-drop-left-dom*) *rule*
also have ... = *subst t* (*fmrestrict-fset* (*frees t*) (*fmdrop-fset* (*fmdom* env_2) env_1 ++$_f$ env_2))
by (*metis subst-restrict′*)
also have ... = *subst t* (*fmrestrict-fset* (*frees t*) (*fmdrop-fset* (*fmdom* env_2) env_1) ++$_f$ *fmrestrict-fset* (*frees t*) env_2)
by *simp*
also have ... = *subst t* (*fmempty* ++$_f$ *fmrestrict-fset* (*frees t*) env_2)
unfolding *fmfilter-alt-defs*
using *fsubsetD*[*OF assms*]
by *auto*
also have ... = *subst t* env_2
by *simp*
finally show *?thesis* **.**
qed

corollary *subst-restrict-closed*: *closed-except t S* ⟹ *subst t* (*fmrestrict-fset S env*) = *subst t env*
by (*metis subst-restrict closed-except-def*)

lemma *subst-closed-except-id*:
assumes *closed-except t S fdisjnt* (*fmdom env*) *S*
shows *subst t env* = *t*
using *assms*
by (*metis fdisjnt-subset-right fmdom-drop-fset fminus-cancel fmrestrict-fset-dom fmrestrict-fset-null closed-except-def subst-drop-fset subst-empty*)

lemma *subst-closed-except-preserved*:
assumes *closed-except t S fdisjnt* (*fmdom env*) *S*
shows *closed-except* (*subst t env*) *S*
using *assms*
by (*metis subst-closed-except-id*)

corollary *subst-closed-id*: *closed t* ⟹ *subst t env* = *t*
by (*simp add*: *subst-closed-except-id fdisjnt-alt-def*)

corollary *subst-closed-preserved*: *closed t* ⟹ *closed* (*subst t env*)
by (*simp add*: *subst-closed-except-preserved fdisjnt-alt-def*)

context begin

private lemma *subst-indep0*:
assumes *closed-env* env_2 *fdisjnt* (*fmdom* env_1) (*fmdom* env_2)
shows *subst* t (env_1 $+\!\!+_f$ env_2) = *subst* (*subst* t env_2) env_1
using *assms* **proof** (*induction* t *arbitrary*: env_1 env_2 *rule*: *raw-induct*)
case (*free name*)
show *?case*
using ‹*closed-env* env_2›
by (*cases rule*: *fmpred-cases*[**where** x = *name*]) (*auto simp*: *subst-closed-id*)
next
case (*abs* t)
show *?case*
by (*rule raw-subst-indep*)
qed *auto*

lemma *subst-indep*:
assumes *closed-env* Γ'
shows *subst* t (Γ $+\!\!+_f$ Γ') = *subst* (*subst* t Γ') Γ
proof −
have *subst* t (Γ $+\!\!+_f$ Γ') = *subst* t (*fmrestrict-fset* (*frees* t) (Γ $+\!\!+_f$ Γ'))
by (*metis subst-restrict'*)
also have ... = *subst* t (*fmrestrict-fset* (*frees* t) Γ $+\!\!+_f$ Γ')
by (*smt fmlookup-add fmlookup-restrict-fset subst-cong*)
also have ... = *subst* t (*fmrestrict-fset* (*frees* t |−| *fmdom* Γ') Γ $+\!\!+_f$ Γ')
by (*rule subst-cong*) (*simp add*: *fmfilter-alt-defs*(*5*))
also have ... = *subst* (*subst* t Γ') (*fmrestrict-fset* (*frees* t |−| *fmdom* Γ') Γ)
apply (*rule subst-indep0*[*OF assms*])
using *fmdom-restrict-fset*
unfolding *fdisjnt-alt-def*
by *auto*
also have ... = *subst* (*subst* t Γ') (*fmrestrict-fset* (*frees* (*subst* t Γ')) Γ)
using *assms* **by** (*auto simp*: *subst-frees*)
also have ... = *subst* (*subst* t Γ') Γ
by *simp*
finally show *?thesis* **.**
qed

lemma *subst-indep'*:
assumes *closed-env* Γ' *fdisjnt* (*fmdom* Γ') (*fmdom* Γ)
shows *subst* t (Γ' $+\!\!+_f$ Γ) = *subst* (*subst* t Γ') Γ
using *assms* **by** (*metis subst-indep fmadd-disjnt*)

lemma *subst-twice*:
assumes $\Gamma' \subseteq_f \Gamma$ *closed-env* Γ'
shows *subst* (*subst* t Γ') Γ = *subst* t Γ
proof −
have *subst* (*subst* t Γ') Γ = *subst* t (Γ $+\!\!+_f$ Γ')
apply (*rule subst-indep*[*symmetric*])

```
    apply fact
    done
  also have ... = subst t Γ
    apply (rule subst-cong)
    using ‹Γ' ⊆ₓ Γ› unfolding fmsubset-alt-def
    by fastforce
  finally show ?thesis .
qed

end

fun matchs :: term list ⇒ 'a list ⇒ (name, 'a) fmap option where
matchs [] [] = Some fmempty |
matchs (t # ts) (u # us) = do { env₁ ← match t u; env₂ ← matchs ts us; Some
(env₁ ++ₓ env₂) } |
matchs - - = None

lemmas matchs-induct = matchs.induct[case-names empty cons]

context begin

private lemma matchs-alt-def0:
  assumes length ps = length vs
 shows map-option (λenv. m ++ₓ env) (matchs ps vs) = map-option (foldl (++ₓ)
m) (those (map2 match ps vs))
using assms proof (induction arbitrary: m rule: list-induct2)
  case (Cons x xs y ys)
  show ?case
    proof (cases match x y)
      case x-y: Some
      show ?thesis
        proof (cases matchs xs ys)
          case None
          with x-y Cons show ?thesis
            by simp
        next
          case Some
          with x-y show ?thesis
            apply simp
            using Cons(2) apply simp
            apply (subst option.map-comp)
            by (auto cong: map-option-cong)
        qed
    qed simp
qed simp

lemma matchs-alt-def:
  assumes length ps = length vs
  shows matchs ps vs = map-option (foldl (++ₓ) fmempty) (those (map2 match
```

ps vs))
by (*subst matchs-alt-def0*[**where** *m* = *fmempty*, *simplified*, *symmetric*, *OF assms*])
(*simp add*: *option.map-ident*)

end

lemma *matchs-neq-length-none*[*simp*]: *length xs* ≠ *length ys* ⟹ *matchs xs ys* = *None*
by (*induct xs ys rule*: *matchs.induct*) *fastforce*+

corollary *matchs-some-eq-length*: *matchs xs ys* = *Some env* ⟹ *length xs* = *length ys*
by (*metis option.distinct*(*1*) *matchs-neq-length-none*)

lemma *matchs-app*[*simp*]:
assumes *length* xs_2 = *length* ys_2
shows *matchs* (xs_1 @ xs_2) (ys_1 @ ys_2) =
matchs xs_1 ys_1 ⟜ (λenv_1. *matchs* xs_2 ys_2 ⟜ (λenv_2. *Some* (env_1 ++$_f$ env_2)))
using *assms*
by (*induct* xs_1 ys_1 *rule*: *matchs.induct*) *fastforce*+

corollary *matchs-appI*:
assumes *matchs xs ys* = *Some* env_1 *matchs xs' ys'* = *Some* env_2
shows *matchs* (*xs* @ *xs'*) (*ys* @ *ys'*) = *Some* (env_1 ++$_f$ env_2)
using *assms*
by (*metis* (*no-types*, *lifting*) *Option.bind-lunit matchs-app matchs-some-eq-length*)

corollary *matchs-dom*:
assumes *matchs ps ts* = *Some env*
shows *fmdom env* = *freess ps*
using *assms*
by (*induction ps ts arbitrary*: *env rule*: *matchs-induct*)
(*auto simp*: *match-dom elim*!: *option-bindE*)

fun *find-match* :: (*term* × *'a*) *list* ⇒ *'a* ⇒ ((*name*, *'a*) *fmap* × *term* × *'a*) *option*
where
find-match [] - = *None* |
find-match ((*pat*, *rhs*) # *cs*) *t* =
(*case match pat t of*
Some env ⇒ *Some* (*env*, *pat*, *rhs*)
| *None* ⇒ *find-match cs t*)

lemma *find-match-map*:
find-match (*map* (λ(*pat*, *t*). (*pat*, *f pat t*)) *cs*) *t* =
map-option (λ(*env*, *pat*, *rhs*). (*env*, *pat*, *f pat rhs*)) (*find-match cs t*)
by (*induct cs*) (*auto split*: *option.splits*)

lemma *find-match-elem*:

assumes *find-match cs t* = *Some* (*env*, *pat*, *rhs*)
shows (*pat*, *rhs*) ∈ *set cs match pat t* = *Some env*
using *assms*
by (*induct cs*) (*auto split*: *option.splits*)

lemma *match-subst-closed*:
assumes *match pat t* = *Some env closed-except rhs* (*frees pat*) *closed t*
shows *closed* (*subst rhs env*)
using *assms*
by (*smt fminusE fmpred-iff fset-mp fsubsetI closed-except-def match-vars match-dom subst-frees*)

fun *rewrite-step* :: (*term* × *'a*) ⇒ *'a* ⇒ *'a option* **where**
rewrite-step (*t*$_1$, *t*$_2$) *u* = *map-option* (*subst t*$_2$) (*match t*$_1$ *u*)

abbreviation *rewrite-step'* :: (*term* × *'a*) ⇒ *'a* ⇒ *'a* ⇒ *bool* (-/ ⊢/ - →/ - [*50*,*0*,*50*] *50*) **where**
r ⊢ *t* → *u* ≡ *rewrite-step r t* = *Some u*

lemma *rewrite-step-closed*:
assumes *frees t*$_2$ |⊆| *frees t*$_1$ (*t*$_1$, *t*$_2$) ⊢ *u* → *u'* *closed u*
shows *closed u'*
proof −
from *assms* **obtain** *env* **where** ∗: *match t*$_1$ *u* = *Some env*
by *auto*
then have *closed* (*subst t*$_2$ *env*)
apply (*rule match-subst-closed*[**where** *pat* = *t*$_1$ **and** *t* = *u*])
using *assms* **unfolding** *closed-except-def* **by** *auto*
with ∗ **show** *?thesis*
using *assms* **by** *auto*
qed

definition *matches* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ≲ *50*) **where**
t ≲ *u* ⟷ (∃ *env*. *subst t env* = *u*)

lemma *matchesI*[*intro*]: *subst t env* = *u* ⟹ *t* ≲ *u*
unfolding *matches-def* **by** *auto*

lemma *matchesE*[*elim*]:
assumes *t* ≲ *u*
obtains *env* **where** *subst t env* = *u*
using *assms* **unfolding** *matches-def* **by** *blast*

definition *overlapping* :: *'a* ⇒ *'a* ⇒ *bool* **where**
overlapping s t ⟷ (∃ *u*. *s* ≲ *u* ∧ *t* ≲ *u*)

lemma *overlapping-refl*: *overlapping t t*
unfolding *overlapping-def matches-def* **by** *blast*

lemma *overlapping-sym*: *overlapping t u* $\Longrightarrow$ *overlapping u t*
unfolding *overlapping-def* **by** *auto*

lemma *overlappingI*[*intro*]: $s \lesssim u \Longrightarrow t \lesssim u \Longrightarrow$ *overlapping s t*
unfolding *overlapping-def* **by** *auto*

lemma *overlappingE*[*elim*]:
assumes *overlapping s t*
obtains *u* **where** $s \lesssim u$ $t \lesssim u$
using *assms* **unfolding** *overlapping-def* **by** *blast*

abbreviation *non-overlapping s t* $\equiv \neg$ *overlapping s t*

corollary *non-overlapping-implies-neq*: *non-overlapping t u* $\Longrightarrow t \neq u$
by (*metis overlapping-refl*)

end

inductive *rewrite-first* :: (*term* $\times$ *'a*::*term*) *list* $\Rightarrow$ *'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool* **where**
match: *match pat t* = *Some env* $\Longrightarrow$ *rewrite-first* ((*pat*, *rhs*) # -) *t* (*subst rhs env*)
|
nomatch: *match pat t* = *None* $\Longrightarrow$ *rewrite-first cs t t'* $\Longrightarrow$ *rewrite-first* ((*pat*, -) # *cs*) *t t'*

code-pred (*modes*: *i* $\Rightarrow$ *i* $\Rightarrow$ *o* $\Rightarrow$ *bool*) *rewrite-first* **.**

lemma *rewrite-firstE*:
assumes *rewrite-first cs t t'*
obtains *pat rhs env* **where** (*pat*, *rhs*) $\in$ *set cs match pat t* = *Some env t'* = *subst rhs env*
using *assms* **by** *induction auto*

This doesn't follow from *find-match-elem*, because *rewrite-first* requires the first match, not just any.

lemma *find-match-rewrite-first*:
assumes *find-match cs t* = *Some* (*env*, *pat*, *rhs*)
shows *rewrite-first cs t* (*subst rhs env*)
using *assms* **proof** (*induction cs*)
case (*Cons c cs*)
obtain *pat0 rhs0* **where** *c* = (*pat0*, *rhs0*)
by *fastforce*
thus *?case*
using *Cons*
by (*cases match pat0 t*) (*auto intro*: *rewrite-first.intros*)
qed *simp*

definition *term-cases* :: (*name* $\Rightarrow$ *'b*) $\Rightarrow$ (*name* $\Rightarrow$ *'b*) $\Rightarrow$ (*'a* $\Rightarrow$ *'a* $\Rightarrow$ *'b*) $\Rightarrow$ *'b* $\Rightarrow$ *'a*::*term* $\Rightarrow$ *'b* **where**
term-cases if-const if-free if-app otherwise t =

```
  (case unconst t of
    Some name ⇒ if-const name |
    None ⇒ (case unfree t of
      Some name ⇒ if-free name |
      None ⇒
        (case unapp t of
          Some (t, u) ⇒ if-app t u
        | None ⇒ otherwise)))

lemma term-cases-cong[fundef-cong]:
  assumes t = u otherwise1 = otherwise2
  assumes (⋀name. t = const name ⟹ if-const1 name = if-const2 name)
  assumes (⋀name. t = free name ⟹ if-free1 name = if-free2 name)
  assumes (⋀u1 u2. t = app u1 u2 ⟹ if-app1 u1 u2 = if-app2 u1 u2)
  shows term-cases if-const1 if-free1 if-app1 otherwise1 t = term-cases if-const2
if-free2 if-app2 otherwise2 u
using assms
unfolding term-cases-def
by (auto split: option.splits)

lemma term-cases[simp]:
  term-cases if-const if-free if-app otherwise (const name) = if-const name
  term-cases if-const if-free if-app otherwise (free name) = if-free name
  term-cases if-const if-free if-app otherwise (app t u) = if-app t u
unfolding term-cases-def
by (auto split: option.splits)

lemma term-cases-template:
  assumes ⋀x. f x = term-cases if-const if-free if-app otherwise x
  shows f (const name) = if-const name
    and f (free name) = if-free name
    and f (app t u) = if-app t u
unfolding assms by (rule term-cases)+

context term begin

function (sequential) strip-comb :: 'a ⇒ 'a × 'a list where
[simp del]: strip-comb t =
  (case unapp t of
    Some (t, u) ⇒
      (let (f, args) = strip-comb t in (f, args @ [u]))
  | None ⇒ (t, []))
by pat-completeness auto


termination
  apply (relation measure size)
  apply rule
  apply auto
```

done

lemma *strip-comb-simps*[*simp*]:
strip-comb (*app t u*) = (*let* (*f*, *args*) = *strip-comb t in* (*f*, *args* @ [*u*]))
unapp t = *None* ⟹ *strip-comb t* = (*t*, [])
by (*subst strip-comb.simps*; *auto*)+

lemma *strip-comb-induct*[*case-names app no-app*]:
assumes ⋀*x y*. *P x* ⟹ *P* (*app x y*)
assumes ⋀*t*. *unapp t* = *None* ⟹ *P t*
shows *P t*
proof (*rule strip-comb.induct*, *goal-cases*)
case (*1 t*)
show *?case*
proof (*cases unapp t*)
case *None*
with *assms* **show** *?thesis* **by** *metis*
next
case (*Some a*)
then show *?thesis*
apply (*cases a*)
using *1 assms* **by** *auto*
qed
qed

lemma *strip-comb-size*: *t'* ∈ *set* (*snd* (*strip-comb t*)) ⟹ *size t'* < *size t*
by (*induction t rule*: *strip-comb-induct*) (*auto split*: *prod.splits*)

lemma *sstrip-comb-termination*[*termination-simp*]:
(*f*, *ts*) = *strip-comb t* ⟹ *t'* ∈ *set ts* ⟹ *size t'* < *size t*
by (*metis snd-conv strip-comb-size*)

lemma *strip-comb-empty*: *snd* (*strip-comb t*) = [] ⟹ *fst* (*strip-comb t*) = *t*
by (*induction t rule*: *strip-comb-induct*) (*auto split*: *prod.splits*)

lemma *strip-comb-app*: *fst* (*strip-comb* (*app t u*)) = *fst* (*strip-comb t*)
by (*simp split*: *prod.splits*)

primrec *list-comb* :: *'a* ⇒ *'a list* ⇒ *'a* **where**
list-comb f [] = *f* |
list-comb f (*t* # *ts*) = *list-comb* (*app f t*) *ts*

lemma *list-comb-app*[*simp*]: *list-comb f* (*xs* @ *ys*) = *list-comb* (*list-comb f xs*) *ys*
by (*induct xs arbitrary*: *f*) *auto*

corollary *list-comb-snoc*: *app* (*list-comb f xs*) *y* = *list-comb f* (*xs* @ [*y*])
by *simp*

lemma *list-comb-size*[*simp*]: *size* (*list-comb f xs*) = *size f* + *size-list size xs*

by (*induct xs arbitrary*: *f*) *auto*

lemma *subst-list-comb*: *subst* (*list-comb f xs*) *env* = *list-comb* (*subst f env*) (*map* (λ*t*. *subst t env*) *xs*)
by (*induct xs arbitrary*: *f*) *auto*

abbreviation *const-list-comb* :: *name* ⇒ *'a list* ⇒ *'a* (**infixl** $$ *70*) **where**
const-list-comb name ≡ *list-comb* (*const name*)

lemma *list-strip-comb*[*simp*]: *list-comb* (*fst* (*strip-comb t*)) (*snd* (*strip-comb t*)) = *t*
by (*induction t rule*: *strip-comb-induct*) (*auto split*: *prod.splits*)

lemma *strip-list-comb*: *strip-comb* (*list-comb f ys*) = (*fst* (*strip-comb f*), *snd* (*strip-comb f*) @ *ys*)
by (*induct ys arbitrary*: *f*) (*auto simp*: *split-beta*)

lemma *strip-list-comb-const*: *strip-comb* (*name* $$ *xs*) = (*const name*, *xs*)
by (*simp add*: *strip-list-comb*)

lemma *frees-list-comb*[*simp*]: *frees* (*list-comb t xs*) = *frees t* |∪| *freess xs*
by (*induct xs arbitrary*: *t*) (*auto simp*: *freess-def*)

lemma *consts-list-comb*: *consts* (*list-comb f xs*) = *consts f* |∪| *ffUnion* (*fset-of-list* (*map consts xs*))
by (*induct xs arbitrary*: *f*) *auto*

lemma *ids-list-comb*: *ids* (*list-comb f xs*) = *ids f* |∪| *ffUnion* (*fset-of-list* (*map ids xs*))
unfolding *ids-def frees-list-comb consts-list-comb freess-def*
apply *auto*
apply (*smt fbind-iff finsert-absorb finsert-fsubset funion-image-bind-eq inf-sup-ord*(*3*))
apply (*metis fbind-iff funionCI funion-image-bind-eq*)
by (*smt fbind-iff funionE funion-image-bind-eq*)

lemma *frees-strip-comb*: *frees t* = *frees* (*fst* (*strip-comb t*)) |∪| *freess* (*snd* (*strip-comb t*))
by (*metis list-strip-comb frees-list-comb*)

lemma *list-comb-cases'*:
  obtains (*app*) *is-app* (*list-comb f xs*)
      | (*empty*) *list-comb f xs* = *f xs* = []
by (*induction xs arbitrary*: *f*) *auto*

lemma *list-comb-cases*[*consumes 1*]:
  assumes *t* = *list-comb f xs*
  obtains (*head*) *t* = *f xs* = []
      | (*app*) *u v* **where** *t* = *app u v*

using *assms* **by** (*metis list-comb-cases' left-right-simps*(*3*))

end

fun *left-nesting* :: *'a*::*term* ⇒ *nat* **where**
[*simp del*]: *left-nesting t* = *term-cases* (λ-. *0*) (λ-. *0*) (λ*t u*. *Suc* (*left-nesting t*)) *0 t*

lemmas *left-nesting-simps*[*simp*] = *term-cases-template*[*OF left-nesting.simps*]

lemma *list-comb-nesting*[*simp*]: *left-nesting* (*list-comb f xs*) = *left-nesting f* + *length xs*
by (*induct xs arbitrary*: *f*) *auto*

lemma *list-comb-cond-inj*:
assumes *list-comb f xs* = *list-comb g ys left-nesting f* = *left-nesting g*
shows *xs* = *ys f* = *g*
using *assms* **proof** (*induction xs arbitrary*: *f g ys*)
case *Nil*
fix *f g* :: *'a*
fix *ys*
assume *prems*: *list-comb f* [] = *list-comb g ys left-nesting f* = *left-nesting g*

hence *left-nesting f* = *left-nesting g* + *length ys*
by *simp*
with *prems* **show** [] = *ys f* = *g*
by *simp*+
next
case (*Cons x xs*)
fix *f g ys*
assume *prems*: *list-comb f* (*x* # *xs*) = *list-comb g ys left-nesting f* = *left-nesting g*

hence *left-nesting* (*list-comb f* (*x* # *xs*)) = *left-nesting* (*list-comb g ys*)
by *simp*
hence *Suc* (*left-nesting f* + *length xs*) = *left-nesting g* + *length ys*
by *simp*
with *prems* **have** *length ys* = *Suc* (*length xs*)
by *linarith*
then obtain *z zs* **where** *ys* = *z* # *zs*
by (*metis length-Suc-conv*)

thus *x* # *xs* = *ys f* = *g*
using *prems Cons*[**where** *ys* = *zs* **and** *f* = *app f x* **and** *g* = *app g z*]
by (*auto dest*: *app-inject*)
qed

lemma *list-comb-inj-second*: *inj* (*list-comb f*)
by (*metis injI list-comb-cond-inj*)

lemma *list-comb-semi-inj*:
assumes *length xs = length ys*
assumes *list-comb f xs = list-comb g ys*
shows *xs = ys f = g*
proof −
from *assms* **have** *left-nesting* (*list-comb f xs*) = *left-nesting* (*list-comb g ys*)
by *simp*
with *assms* **have** *left-nesting f = left-nesting g*
unfolding *list-comb-nesting* **by** *simp*
with *assms* **show** *xs = ys f = g*
by (*metis list-comb-cond-inj*)+
qed

fun *no-abs* :: *'a*::*term* ⇒ *bool* **where**
[*simp del*]: *no-abs t* = *term-cases* (λ-. *True*) (λ-. *True*) (λ*t u*. *no-abs t* ∧ *no-abs u*)
False t

lemmas *no-abs-simps*[*simp*] = *term-cases-template*[*OF no-abs.simps*]

lemma *no-abs-induct*[*consumes 1*, *case-names free const app*, *induct pred*: *no-abs*]:
assumes *no-abs t*
assumes ⋀*name*. *P* (*free name*)
assumes ⋀*name*. *P* (*const name*)
assumes ⋀*t*$_1$ *t*$_2$. *P t*$_1$ ⟹ *no-abs t*$_1$ ⟹ *P t*$_2$ ⟹ *no-abs t*$_2$ ⟹ *P* (*app t*$_1$ *t*$_2$)
shows *P t*
using *assms*(*1*) **proof** (*induction rule*: *no-abs.induct*)
case (*1 t*)
show *?case*
proof (*cases rule*: *pre-term-class.term-cases*[**where** *t* = *t*])
case (*free name*)
then show *?thesis*
using *assms* **by** *auto*
next
case (*const name*)
then show *?thesis*
using *assms* **by** *auto*
next
case (*app u*$_1$ *u*$_2$)
with *assms* **have** *P u*$_1$ *P u*$_2$
using *1* **by** *auto*
with *assms* ‹*no-abs t*› **show** *?thesis*
unfolding ‹*t* = -› **by** *auto*
next
case *other*
then show *?thesis*
using ‹*no-abs t*›
apply (*subst* (*asm*) *no-abs.simps*)
apply (*subst* (*asm*) *term-cases-def*)

```
      by simp
    qed
qed

lemma no-abs-cases[consumes 1, cases pred: no-abs]:
  assumes no-abs t
  obtains (free) name where t = free name
        | (const) name where t = const name
        | (app) t1 t2 where t = app t1 t2 no-abs t1 no-abs t2
proof (cases rule: pre-term-class.term-cases[where t = t])
  case (app u1 u2)
  show ?thesis
    apply (rule that(3))
    apply fact
    using ‹no-abs t› unfolding ‹t = -› by auto
next
  case other
  then have False
    using ‹no-abs t›
    apply (subst (asm) no-abs.simps)
    by (auto simp: term-cases-def)
  then show ?thesis ..
qed

definition is-abs :: 'a::term ⇒ bool where
is-abs t = term-cases (λ-. False) (λ-. False) (λ- -. False) True t

lemmas is-abs-simps[simp] = term-cases-template[OF is-abs-def]

definition abs-ish :: term list ⇒ 'a::term ⇒ bool where
abs-ish pats rhs ⟷ pats ≠ [] ∨ is-abs rhs

locale simple-syntactic-and =
  fixes P :: 'a::term ⇒ bool
  assumes app: P (app t u) ⟷ P t ∧ P u
begin

context
  notes app[simp]
begin

lemma list-comb: P (list-comb f xs) ⟷ P f ∧ list-all P xs
by (induction xs arbitrary: f) auto

corollary list-combE:
  assumes P (list-comb f xs)
  shows P f x ∈ set xs ⟹ P x
using assms
by (auto simp: list-comb list-all-iff)
```

lemma *match*:
assumes *match pat t* = *Some env P t*
shows *fmpred* (λ-. *P*) *env*
using *assms*
by (*induction pat t env rule*: *match-some-induct*) *auto*

lemma *matchs*:
assumes *matchs pats ts* = *Some env list-all P ts*
shows *fmpred* (λ-. *P*) *env*
using *assms*
by (*induction pats ts arbitrary*: *env rule*: *matchs.induct*) (*auto elim*!: *option-bindE intro*: *match*)

end

end

locale *subst-syntactic-and* = *simple-syntactic-and* +
assumes *subst*: *P t* ⟹ *fmpred* (λ-. *P*) *env* ⟹ *P* (*subst t env*)
begin

lemma *rewrite-step*:
assumes (*lhs*, *rhs*) ⊢ *t* → *t*′ *P t P rhs*
shows *P t*′
using *assms* **by** (*auto intro*: *match subst*)

end

locale *simple-syntactic-or* =
fixes *P* :: ′*a*::*term* ⇒ *bool*
assumes *app*: *P* (*app t u*) ⟷ *P t* ∨ *P u*
begin

context
notes *app*[*simp*]
begin

lemma *list-comb*: *P* (*list-comb f xs*) ⟷ *P f* ∨ *list-ex P xs*
by (*induction xs arbitrary*: *f*) *auto*

lemma *match*:
assumes *match pat t* = *Some env* ¬ *P t*
shows *fmpred* (λ- *t*. ¬ *P t*) *env*
using *assms*
by (*induction pat t env rule*: *match-some-induct*) *auto*

end

sublocale *neg*: *simple-syntactic-and* $\lambda t.\ \neg\ P\ t$
by *standard* (*auto simp*: *app*)

end

global-interpretation *no-abs*: *simple-syntactic-and no-abs*
by *standard simp*

global-interpretation *closed*: *simple-syntactic-and* $\lambda t.$ *closed-except* $t\ S$ **for** S
by *standard* (*simp add*: *closed-except-def*)

global-interpretation *closed*: *subst-syntactic-and closed*
by *standard* (*rule subst-closed-preserved*)

corollary *closed-list-comb*: *closed* (*name* \$\$ *args*) $\longleftrightarrow$ *list-all closed args*
by (*simp add*: *closed.list-comb*)

locale *term-struct-rel* =
  fixes P :: $'a$::*term* $\Rightarrow$ $'b$::*term* $\Rightarrow$ *bool*
  assumes *P-t-const*: $P\ t$ (*const name*) $\Longrightarrow$ t = *const name*
  assumes *P-const-const*: P (*const name*) (*const name*)
  assumes *P-t-app*: $P\ t$ (*app* $u_1\ u_2$) $\Longrightarrow$ $\exists\, t_1\ t_2.\ t$ = *app* $t_1\ t_2 \wedge P\ t_1\ u_1 \wedge P\ t_2\ u_2$
  assumes *P-app-app*: $P\ t_1\ u_1 \Longrightarrow P\ t_2\ u_2 \Longrightarrow P$ (*app* $t_1\ t_2$) (*app* $u_1\ u_2$)
begin

abbreviation *P-env* :: ($'k$, $'a$) *fmap* $\Rightarrow$ ($'k$, $'b$) *fmap* $\Rightarrow$ *bool* **where**
P-env $\equiv$ *fmrel P*

lemma *related-match*:
  assumes *match x u* = *Some env P t u*
  obtains *env'* **where** *match x t* = *Some env' P-env env' env*
using *assms* **proof** (*induction x u env arbitrary*: *t thesis rule*: *match-some-induct*)
  case (*app* $v_1\ v_2\ w_1\ w_2\ env_1\ env_2$)
  obtain $u_1\ u_2$ **where** t = *app* $u_1\ u_2\ P\ u_1\ w_1\ P\ u_2\ w_2$
    using *P-t-app*[*OF* ‹*P t* (*app* $w_1\ w_2$)›] **by** *auto*
  with *app* **obtain** $env_1'\ env_2'$
    where *match* $v_1\ u_1$ = *Some* env_1' *P-env* $env_1'\ env_1$
      and *match* $v_2\ u_2$ = *Some* env_2' *P-env* $env_2'\ env_2$
    by *metis*
  hence *match* (v_1 \$ v_2) (*app* $u_1\ u_2$) = *Some* ($env_1' +\!\!+_f env_2'$)
    by *simp*

  show *?case*
    proof (*rule app.prems*)
      show *match* (v_1 \$ v_2) t = *Some* ($env_1' +\!\!+_f env_2'$)
        unfolding ‹t = -› **by** *fact*
    next
      show *P-env* ($env_1' +\!\!+_f env_2'$) ($env_1 +\!\!+_f env_2$)
        by *rule fact+*

```
    qed
qed (auto split: option.splits if-splits dest: P-t-const)

lemma list-combI:
  assumes list-all2 P us1 us2 P t1 t2
  shows P (list-comb t1 us1) (list-comb t2 us2)
using assms
by (induction arbitrary: t1 t2 rule: list.rel-induct) (auto intro: P-app-app)

lemma list-combE:
  assumes P t (name $$ args)
  obtains args' where t = name $$ args' list-all2 P args' args
using assms proof (induction args arbitrary: t thesis rule: rev-induct)
  case Nil
  hence P t (const name)
    by simp
  hence t = const name
    using P-t-const by auto
  with Nil show ?case
    by simp
next
  case (snoc x xs)
  hence P t (app (name $$ xs) x)
    by simp
  obtain t' y where t = app t' y P t' (name $$ xs) P y x
    using P-t-app[OF ‹P t (app (name $$ xs) x)›] by auto
  with snoc obtain ys where t' = name $$ ys list-all2 P ys xs
    by blast
  show ?case
    proof (rule snoc.prems)
      show t = name $$ (ys @ [y])
        unfolding ‹t = app t' y› ‹t' = name $$ ys›
        by simp
    next
      have list-all2 P [y] [x]
        using ‹P y x› by simp
      thus list-all2 P (ys @ [y]) (xs @ [x])
        using ‹list-all2 P ys xs›
        by (metis list-all2-appendI)
    qed
qed

end

locale term-struct-rel-strong = term-struct-rel +
  assumes P-const-t: P (const name) t ⟹ t = const name
  assumes P-app-t: P (app u1 u2) t ⟹ ∃ t1 t2. t = app t1 t2 ∧ P u1 t1 ∧ P u2
t2
begin
```

lemma *unconst-rel*: *P t u* ⟹ *unconst t* = *unconst u*
by (*metis P-const-t P-t-const const-name-simps*(*2*) *is-const-def unconst-const*)

lemma *unapp-rel*: *P t u* ⟹ *rel-option* (*rel-prod P P*) (*unapp t*) (*unapp u*)
by (*smt P-app-t P-t-app is-app-def left-right-simps*(*3*) *option.rel-sel option.sel option.simps*(*3*) *rel-prod-inject unapp-app*)

lemma *match-rel*:
assumes *P t u*
shows *rel-option P-env* (*match p t*) (*match p u*)
using *assms* **proof** (*induction p arbitrary*: *t u*)
case (*Const name*)
thus *?case*
by (*auto split*: *option.splits simp*: *unconst-rel*)
next
case (*App p1 p2*)
hence *rel-option* (*rel-prod P P*) (*unapp t*) (*unapp u*)
by (*metis unapp-rel*)
thus *?case*
using *App*
by *cases* (*auto split*: *option.splits intro*!: *rel-option-bind*)
qed *auto*

lemma *find-match-rel*:
assumes *list-all2* (*rel-prod* (=) *P*) *cs cs*′ *P t t*′
shows *rel-option* (*rel-prod P-env* (*rel-prod* (=) *P*)) (*find-match cs t*) (*find-match cs*′ *t*′)
using *assms* **proof** *induction*
case (*Cons x xs y ys*)
moreover obtain *px tx py ty* **where** *x* = (*px*, *tx*) *y* = (*py*, *ty*)
by (*cases x*, *cases y*) *auto*
moreover note *match-rel*[*OF Cons*(*4*), **where** *p* = *px*]
ultimately show *?case*
by (*auto elim*: *option.rel-cases*)
qed *auto*

end

fun *convert-term* :: *'a*::*term* ⇒ *'b*::*term* **where**
[*simp del*]: *convert-term t* = *term-cases const free* (λ*t u*. *app* (*convert-term t*) (*convert-term u*)) *undefined t*

lemmas *convert-term-simps*[*simp*] = *term-cases-template*[*OF convert-term.simps*]

lemma *convert-term-id*:
assumes *no-abs t*
shows *convert-term t* = *t*
using *assms* **by** *induction auto*

lemma *convert-term-no-abs*:
assumes *no-abs t*
shows *no-abs* (*convert-term t*)
using *assms* **by** *induction auto*

lemma *convert-term-inj*:
assumes *no-abs t no-abs t′ convert-term t* = *convert-term t′*
shows *t* = *t′*
using *assms* **proof** (*induction t arbitrary*: *t′*)
case (*free name*)
then show *?case*
by *cases* (*auto dest*: *term-inject*)
next
case (*const name*)
then show *?case*
by *cases* (*auto dest*: *term-inject*)
next
case (*app* t_1 t_2)
from ‹*no-abs t′*› **show** *?case*
apply *cases*
using *app* **by** (*auto dest*: *term-inject*)
qed

lemma *convert-term-idem*:
assumes *no-abs t*
shows *convert-term* (*convert-term t*) = *convert-term t*
using *assms* **by** (*induction t*) *auto*

lemma *convert-term-frees*[*simp*]:
assumes *no-abs t*
shows *frees* (*convert-term t*) = *frees t*
using *assms* **by** *induction auto*

lemma *convert-term-consts*[*simp*]:
assumes *no-abs t*
shows *consts* (*convert-term t*) = *consts t*
using *assms* **by** *induction auto*

The following lemma does not generalize to when *match t u* = *None*. Assume matching return *None*, because the pattern is an application and the object is a term satisfying *is-abs*. Now, *convert-term* applied to the object will produce *undefined*. Of course we don't know anything about that and whether or not that matches. A workaround would be to require implementations of *term* to prove ∃ *t*. *is-abs t*, such that *convert-term* could use that instead of *undefined*. This seems to be too much of a special case in order to be useful.

lemma *convert-term-match*:

assumes *match t u* = *Some env*
shows *match t* (*convert-term u*) = *Some* (*fmmap convert-term env*)
using *assms* **by** (*induction t u env rule*: *match-some-induct*) *auto*

3.3 Related work

Schmidt-Schauß and Siekmann [4] discuss the concept of *unification algebras*. They generalize terms to *objects* and substitutions to *mappings*. A unification problem can be rephrased to finding a mapping such that a set of objects are mapped to the same object. The advantage of this generalization is that other – superficially unrelated – problems like solving algebraic equations or querying logic programs can be seen as unification problems.

In particular, the authors note that among the similarities of such problems are that "objects [have] variables" whose "names do not matter" and "there exists an operation like substituting objects into variables". The major difference between this formalization and their work is that I use concrete types for variables and mappings. Otherwise, some similarities to here can be found.

Eder [2] discusses properties of substitutions with a special focus on a partial ordering between substitutions. However, Eder constructs and uses a concrete type of first-order terms, similarly to Sternagel and Thiemann [6].

Williams [9] defines substitutions as elements in a monoid. In this setting, instantiations can be represented as *monoid actions*. Williams then proceeds to define – for arbitrary sets of terms and variables – the notion of *instantiation systems,* heavily drawing on notation from Schmidt-Schauß and Siekmann. Some of the presented axioms are also present in this formalization, as are some theorems that have a direct correspondence.

end

3.4 Instantiation of class *term* for type *term*

theory *Term*
imports *Term-Class*
begin

instantiation *term* :: *term* **begin**

All of these definitions need to be marked as *code del*; otherwise the code generator will attempt to generate these, which will fail because they are not executable.

definition *abs-pred-term* :: (*term* ⇒ *bool*) ⇒ *term* ⇒ *bool* **where**
[*code del*]: *abs-pred P t* ⟷
(∀ *x*. *t* = *Bound x* ⟶ *P t*) ∧
(∀ *t'*. *t* = Λ *t'* ⟶ *P t'* ⟶ *P t*)

instance proof (*standard*, *goal-cases*)
  case (*1 P t*)
  then show *?case*
    by (*induction t*) (*auto simp*: *abs-pred-term-def const-term-def free-term-def app-term-def*)
qed (*auto simp*: *abs-pred-term-def*)

end

lemma *is-const-free*[*simp*]: ¬ *is-const* (*Free name*)
unfolding *is-const-def* **by** *simp*

lemma *is-free-app*[*simp*]: ¬ *is-free* (*t* $ *u*)
unfolding *is-free-def* **by** *simp*

lemma *is-free-free*[*simp*]: *is-free* (*Free name*)
unfolding *is-free-def* **by** *simp*

lemma *is-const-const*[*simp*]: *is-const* (*Const name*)
unfolding *is-const-def* **by** *simp*

lemma *list-comb-free*: *is-free* (*list-comb f xs*) ⟹ *is-free f*
apply (*induction xs arbitrary*: *f*)
apply *auto*
subgoal premises *prems*
  apply (*insert prems*(*1*)[*OF prems*(*2*)])
  unfolding *app-term-def*
  by *simp*
done

lemma *const-list-comb-free*[*simp*]: ¬ *is-free* (*name* $$ *args*)
by (*fastforce dest*: *list-comb-free simp*: *const-term-def*)

corollary *const-list-comb-neq-free*[*simp*]: *name* $$ *args* ≠ *free name'*
by (*metis const-list-comb-free is-free-simps*(*1*))

declare *const-list-comb-neq-free*[*symmetric*, *simp*]

lemma *match-list-comb-list-comb-eq-lengths*[*simp*]:
  assumes *length ps* = *length vs*
  shows *match* (*list-comb f ps*) (*list-comb g vs*) =
    (*case match f g of*
      Some env ⇒
        (*case those* (*map2 match ps vs*) *of*
          Some envs ⇒ *Some* (*foldl* (+$+_f$) *env envs*)
        | *None* ⇒ *None*)
    | *None* ⇒ *None*)
using *assms*

by (*induction ps vs arbitrary*: *f g rule*: *list-induct2*) (*auto split*: *option.splits simp*: *app-term-def*)

lemma *matchs-match-list-comb*[*simp*]: *match* (*name* $$ *xs*) (*name* $$ *ys*) = *matchs xs ys*
proof (*induction xs arbitrary*: *ys rule*: *rev-induct*)
  case *Nil*
  show *?case*
    by (*cases ys rule*: *rev-cases*) (*auto simp*: *const-term-def*)
next
  case (*snoc x xs*)
  note *snoc0* = *snoc*
  have *match* (*name* $$ *xs* $ *x*) (*name* $$ *ys*) = *matchs* (*xs* @ [*x*]) *ys*
    proof (*cases ys rule*: *rev-cases*)
      case (*snoc zs z*)
      show *?thesis*
        unfolding *snoc* **using** *snoc0*
        by *simp*
    qed *auto*
  thus *?case*
    by (*simp add*: *app-term-def*)
qed

fun *bounds* :: *term* ⇒ *nat fset* **where**
bounds (*Bound i*) = {| *i* |} |
bounds (*t*$_1$ $ *t*$_2$) = *bounds t*$_1$ |∪| *bounds t*$_2$ |
bounds (Λ *t*) = (λ*i*. *i* − *1*) |‘| (*bounds t* − {| *0* |}) |
bounds - = {||}

definition *shift-nat* :: *nat* ⇒ *int* ⇒ *nat* **where**
[*simp*]: *shift-nat n k* = (*if k* ≥ *0 then n* + *nat k else n* − *nat* |*k*|)

fun *incr-bounds* :: *int* ⇒ *nat* ⇒ *term* ⇒ *term* **where**
incr-bounds inc lev (*Bound i*) = (*if i* ≥ *lev then Bound* (*shift-nat i inc*) *else Bound i*) |
incr-bounds inc lev (Λ *u*) = Λ *incr-bounds inc* (*lev* + *1*) *u* |
incr-bounds inc lev (*t*$_1$ $ *t*$_2$) = *incr-bounds inc lev t*$_1$ $ *incr-bounds inc lev t*$_2$ |
incr-bounds - - *t* = *t*

lemma *incr-bounds-frees*[*simp*]: *frees* (*incr-bounds n k t*) = *frees t*
by (*induction n k t rule*: *incr-bounds.induct*) *auto*

lemma *incr-bounds-zero*[*simp*]: *incr-bounds 0 i t* = *t*
by (*induct t arbitrary*: *i*) *auto*

fun *replace-bound* :: *nat* ⇒ *term* ⇒ *term* ⇒ *term* **where**
replace-bound lev (*Bound i*) *t* = (*if i* < *lev then Bound i else if i* = *lev then incr-bounds* (*int lev*) *0 t else Bound* (*i* − *1*)) |
replace-bound lev (*t*$_1$ $ *t*$_2$) *t* = *replace-bound lev t*$_1$ *t* $ *replace-bound lev t*$_2$ *t* |

replace-bound lev (Λ *u*) *t* = Λ *replace-bound* (*lev* + *1*) *u t* |
replace-bound - t - = t

abbreviation *β-reduce* :: *term* ⇒ *term* ⇒ *term* (- [-]$_\beta$) **where**
t [*u*]$_\beta$ ≡ *replace-bound 0 t u*

lemma *replace-bound-frees*: *frees* (*replace-bound n t t'*) |⊆| *frees t* |∪| *frees t'*
by (*induction n t t' rule*: *replace-bound.induct*) *auto*

lemma *replace-bound-eq*:
assumes *i* |∉| *bounds t*
shows *replace-bound i t t'* = *incr-bounds* (−*1*) (*i* + *1*) *t*
using *assms*
by (*induct t arbitrary*: *i*) *force+*

fun *wellformed'* :: *nat* ⇒ *term* ⇒ *bool* **where**
wellformed' n (*t*$_1$ $ *t*$_2$) ⟷ *wellformed' n t*$_1$ ∧ *wellformed' n t*$_2$ |
wellformed' n (*Bound n'*) ⟷ *n'* < *n* |
wellformed' n (Λ *t*) ⟷ *wellformed'* (*n* + *1*) *t* |
wellformed' - - ⟷ *True*

lemma *wellformed-inc*:
assumes *wellformed' k t k* ≤ *n*
shows *wellformed' n t*
using *assms*
by (*induct t arbitrary*: *k n*) *auto*

abbreviation *wellformed* :: *term* ⇒ *bool* **where**
wellformed ≡ *wellformed' 0*

lemma *wellformed'-replace-bound-eq*:
assumes *wellformed' n t k* ≥ *n*
shows *replace-bound k t u = t*
using *assms*
by (*induction t arbitrary*: *n k*) *auto*

lemma *wellformed-replace-bound-eq*: *wellformed t* ⟹ *replace-bound k t u = t*
by (*rule wellformed'-replace-bound-eq*) *simp+*

lemma *incr-bounds-eq*: *n* ≥ *k* ⟹ *wellformed' k t* ⟹ *incr-bounds i n t = t*
by (*induct t arbitrary*: *k n*) *force+*

lemma *incr-bounds-subst*:
assumes ⋀*t*. *t* ∈ *fmran' env* ⟹ *wellformed t*
shows *incr-bounds i n* (*subst t env*) = *subst* (*incr-bounds i n t*) *env*
proof (*induction t arbitrary*: *n*)
case (*Free name*)
show *?case*
proof (*cases fmlookup env name*)

```
      case (Some t)
      hence wellformed t
        using assms by (auto intro: fmran'I)
      hence incr-bounds i n t = t
        by (subst incr-bounds-eq) auto
      with Some show ?thesis
        by simp
    qed auto
qed auto

lemma incr-bounds-wellformed:
  assumes wellformed' m u
  shows wellformed' (k + m) (incr-bounds (int k) n u)
using assms
by (induct u arbitrary: n m) force+

lemma replace-bound-wellformed:
  assumes wellformed u wellformed' (Suc k) t i ≤ k
  shows wellformed' k (replace-bound i t u)
using assms
apply (induction t arbitrary: i k)
apply auto
using incr-bounds-wellformed[where m = 0, simplified]
using wellformed-inc by blast

lemma subst-wellformed:
  assumes wellformed' n t fmpred (λ-. wellformed) env
  shows wellformed' n (subst t env)
using assms
by (induction t arbitrary: n) (auto split: option.splits intro: wellformed-inc)

global-interpretation wellformed: simple-syntactic-and wellformed' n for n
by standard (auto simp: app-term-def)

global-interpretation wellformed: subst-syntactic-and wellformed
by standard (auto intro: subst-wellformed)

lemma match-list-combE:
  assumes match (name $$ xs) t = Some env
  obtains ys where t = name $$ ys matchs xs ys = Some env
proof −
  from assms that show thesis
    proof (induction xs arbitrary: t env thesis rule: rev-induct)
      case Nil
      from Nil(1) show ?case
        apply (auto simp: const-term-def split: option.splits if-splits)
        using Nil(2)[where ys = []]
        by auto
    next
```

```
      case (snoc x xs)
      obtain t' y where t = app t' y
        using ‹match - t = Some env›
        by (auto simp: app-term-def elim!: option-bindE)
      from snoc(2) obtain env₁ env₂
        where match (name $$ xs) t' = Some env₁ match x y = Some env₂ env =
env₁ ++_f env₂
        unfolding ‹t = -› by (fastforce simp: app-term-def elim: option-bindE)

      with snoc obtain ys where t' = name $$ ys matchs xs ys = Some env₁
        by blast

      show ?case
        proof (rule snoc(3))
          show t = name $$ (ys @ [y])
            unfolding ‹t = -› ‹t' = -›
            by simp
        next
          have matchs [x] [y] = Some env₂
            using ‹match x y = -› by simp
          thus matchs (xs @ [x]) (ys @ [y]) = Some env
            unfolding ‹env = -› using ‹matchs xs ys = -›
            by simp
        qed
    qed
qed

lemma left-nesting-neq-match:
  left-nesting f ≠ left-nesting g ⟹ is-const (fst (strip-comb f)) ⟹ match f g =
None
proof (induction f arbitrary: g)
  case (Const x)
  then show ?case
    apply (auto split: option.splits)
    apply (fold const-term-def)
    apply auto
    done
next
  case (App f1 f2)
   then have f1-g: Suc (left-nesting f1) ≠ left-nesting g and f1: is-const (fst
(strip-comb f1))
    apply (fold app-term-def)
    by (auto split: prod.splits)
  show ?case
    proof (cases unapp g)
      case (Some g')
      obtain g1 g2 where g' = (g1, g2)
        by (cases g') auto
      with Some have g = app g1 g2
```

```
      by auto
    with f1-g have left-nesting f1 ≠ left-nesting g1
      by simp
    with f1 App have match f1 g1 = None
      by simp
    then show ?thesis
      unfolding ‹g' = -› ‹g = -›
      by simp
  qed simp
qed auto

context begin

private lemma match-list-comb-list-comb-none-structure:
  assumes length ps = length vs left-nesting f ≠ left-nesting g
  assumes is-const (fst (strip-comb f))
  shows match (list-comb f ps) (list-comb g vs) = None
using assms
by (induction ps vs arbitrary: f g rule: list-induct2) (auto simp: split-beta left-nesting-neq-match)

lemma match-list-comb-list-comb-some:
  assumes match (list-comb f ps) (list-comb g vs) = Some env left-nesting f =
left-nesting g
  assumes is-const (fst (strip-comb f))
  shows match f g ≠ None length ps = length vs
proof −
  have match f g ≠ None ∧ length ps = length vs
    proof (cases rule: linorder-cases[where y = length vs and x = length ps])
      assume length ps = length vs
      thus ?thesis
        using assms
        proof (induction ps vs arbitrary: f g env rule: list-induct2)
          case (Cons p ps v vs)
          have match (app f p) (app g v) ≠ None ∧ length ps = length vs
            proof (rule Cons)
              show is-const (fst (strip-comb (app f p)))
                using Cons by (simp add: split-beta)
            next
              show left-nesting (app f p) = left-nesting (app g v)
                using Cons by simp
            next
              show match (list-comb (app f p) ps) (list-comb (app g v) vs) = Some
env
                using Cons by simp
            qed
          thus ?case
            unfolding app-term-def
            by (auto elim: match.elims option-bindE)
        qed auto
```

```
    next
      assume length ps < length vs
     then obtain vs₁ vs₂ where vs = vs₁ @ vs₂ length ps = length vs₂ 0 < length
vs₁
       by (auto elim: list-split)

      have match (list-comb f ps) (list-comb (list-comb g vs₁) vs₂) = None
        proof (rule match-list-comb-list-comb-none-structure)
          show left-nesting f ≠ left-nesting (list-comb g vs₁)
            using assms(2) ‹0 < length vs₁› by simp
        qed fact+
      hence match (list-comb f ps) (list-comb g vs) = None
        unfolding ‹vs = -› by simp
      hence False
        using assms by auto
      thus ?thesis ..
    next
      assume length vs < length ps
     then obtain ps₁ ps₂ where ps = ps₁ @ ps₂ length ps₂ = length vs 0 < length
ps₁
       by (auto elim: list-split)

      have match (list-comb (list-comb f ps₁) ps₂) (list-comb g vs) = None
        proof (rule match-list-comb-list-comb-none-structure)
          show left-nesting (list-comb f ps₁) ≠ left-nesting g
            using assms ‹0 < length ps₁› by simp
        next
          show is-const (fst (strip-comb (list-comb f ps₁)))
            using assms by (simp add: strip-list-comb)
        qed fact
      hence match (list-comb f ps) (list-comb g vs) = None
        unfolding ‹ps = -› by simp
      hence False
        using assms by auto
      thus ?thesis ..
    qed
  thus match f g ≠ None length ps = length vs
    by simp+
qed

end

lemma match-list-comb-list-comb-none-name[simp]:
  assumes name ≠ name'
  shows match (name $$ ps) (name' $$ vs) = None
proof (rule ccontr)
  assume match (name $$ ps) (name' $$ vs) ≠ None
  then obtain env where *: match (name $$ ps) (name' $$ vs) = Some env
    by blast
```

hence *match* (*const name*) (*const name'* :: *'a*) ≠ *None*
by (*rule match-list-comb-list-comb-some*) (*simp add*: *is-const-def*)+
hence *name* = *name'*
unfolding *const-term-def*
by (*simp split*: *if-splits*)
thus *False*
using *assms* **by** *blast*
qed

lemma *match-list-comb-list-comb-none-length*[*simp*]:
assumes *length ps* ≠ *length vs*
shows *match* (*name* $$ *ps*) (*name'* $$ *vs*) = *None*
proof (*rule ccontr*)
assume *match* (*name* $$ *ps*) (*name'* $$ *vs*) ≠ *None*
then obtain *env* **where** *match* (*name* $$ *ps*) (*name'* $$ *vs*) = *Some env*
by *blast*
hence *length ps* = *length vs*
by (*rule match-list-comb-list-comb-some*) (*simp add*: *is-const-def*)+
thus *False*
using *assms* **by** *blast*
qed

context *term-struct-rel* **begin**

corollary *related-matchs*:
assumes *matchs ps* ts_2 = *Some* env_2 *list-all2 P* ts_1 ts_2
obtains env_1 **where** *matchs ps* ts_1 = *Some* env_1 *P-env* env_1 env_2
proof −
fix *name* — dummy

from *assms* **have** *match* (*name* $$ *ps*) (*name* $$ ts_2) = *Some* env_2
by *simp*
moreover have *P* (*name* $$ ts_1) (*name* $$ ts_2)
using *assms* **by** (*auto intro*: *P-const-const list-combI*)
ultimately obtain env_1 **where** *match* (*name* $$ *ps*) (*name* $$ ts_1) = *Some* env_1 *P-env* env_1 env_2
by (*metis related-match*)
hence *matchs ps* ts_1 = *Some* env_1
by *simp*

show *thesis*
by (*rule that*) *fact*+
qed

end

end

Chapter 4

Wellformedness of patterns

theory *Pats*
imports *Term*
begin

The *term* class already defines a generic definition of *matching* a *pattern* with a term. Importantly, the type of patterns is neither generic, nor a dedicated pattern type; instead, it is *term* itself.

Patterns are a proper subset of terms, with the restriction that no abstractions may occur and there must be at most a single occurrence of any variable (usually known as *linearity*). The first restriction can be modelled in a datatype, the second cannot. Consequently, I define a predicate that captures both properties.

Using linearity, many more generic properties can be proved, for example that substituting the environment produced by matching yields the matched term.

fun *linear* :: *term* ⇒ *bool* **where**
linear (*Free* -) ⟷ *True* |
linear (*Const* -) ⟷ *True* |
linear (t_1 $ t_2) ⟷ *linear* t_1 ∧ *linear* t_2 ∧ ¬ *is-free* t_1 ∧ *fdisjnt* (*frees* t_1) (*frees* t_2) |
linear - ⟷ *False*

lemmas *linear-simps*[*simp*] =
linear.simps(*2*)[*folded const-term-def*]
linear.simps(*3*)[*folded app-term-def*]

lemma *linear-implies-no-abs*: *linear t* ⟹ *no-abs t*
proof *induction*
case *Const*
then show *?case*
by (*fold const-term-def free-term-def app-term-def*) *auto*
next
case *Free*

```
  then show ?case
    by (fold const-term-def free-term-def app-term-def) auto
next
  case App
  then show ?case
    by (fold const-term-def free-term-def app-term-def) auto
qed auto

fun linears :: term list ⇒ bool where
linears [] ⟷ True |
linears (t # ts) ⟷ linear t ∧ fdisjnt (frees t) (freess ts) ∧ linears ts

lemma linears-butlastI[intro]: linears ts ⟹ linears (butlast ts)
proof (induction ts)
  case (Cons t ts)
  hence linear t linears (butlast ts)
    by simp+
  moreover have  fdisjnt (frees t) (freess (butlast ts))
    proof (rule fdisjnt-subset-right)
      show freess (butlast ts) |⊆| freess ts
        by (rule freess-subset) (auto dest: in-set-butlastD)
    next
      show fdisjnt (frees t) (freess ts)
        using Cons by simp
    qed
  ultimately show ?case
    by simp
qed simp

lemma linears-appI[intro]:
  assumes linears xs linears ys fdisjnt (freess xs) (freess ys)
  shows linears (xs @ ys)
using assms proof (induction xs)
  case (Cons z zs)
  hence linears zs
    by simp+

  have fdisjnt (frees z) (freess zs |∪| freess ys)
    proof (rule fdisjnt-union-right)
      show fdisjnt (frees z) (freess zs)
        using ‹linears (z # zs)› by simp
    next
      have frees z |⊆| freess (z # zs)
        unfolding freess-def by simp
      thus fdisjnt (frees z) (freess ys)
        by (rule fdisjnt-subset-left) fact
    qed

  moreover have linears (zs @ ys)
```

```
    proof (rule Cons)
      show fdisjnt (freess zs) (freess ys)
        using Cons
        by (auto intro: freess-subset fdisjnt-subset-left)
    qed fact+

  ultimately show ?case
    using Cons by auto
qed simp

lemma linears-linear: linears ts ⟹ t ∈ set ts ⟹ linear t
by (induct ts) auto

lemma linears-singleI[intro]: linear t ⟹ linears [t]
by (simp add: freess-def fdisjnt-alt-def)

lemma linear-strip-comb: linear t ⟹ linear (fst (strip-comb t))
by (induction t rule: strip-comb-induct) (auto simp: split-beta)

lemma linears-strip-comb: linear t ⟹ linears (snd (strip-comb t))
proof (induction t rule: strip-comb-induct)
  case (app t₁ t₂)
  have linears (snd (strip-comb t₁) @ [t₂])
    proof (intro linears-appI linears-singleI)
      have freess (snd (strip-comb t₁)) |⊆| frees t₁
        by (subst frees-strip-comb) auto
      moreover have fdisjnt (frees t₁) (frees t₂)
        using app by auto
      ultimately have fdisjnt (freess (snd (strip-comb t₁))) (frees t₂)
        by (rule fdisjnt-subset-left)
      thus fdisjnt (freess (snd (strip-comb t₁))) (freess [t₂])
        by simp
    next
      show linear t₂ linears (snd (strip-comb t₁))
        using app by auto
    qed
  thus ?case
    by (simp add: split-beta)
qed auto

lemma linears-appendD:
  assumes linears (xs @ ys)
  shows linears xs linears ys fdisjnt (freess xs) (freess ys)
using assms proof (induction xs)
  case (Cons x xs)
  assume linears ((x # xs) @ ys)

  hence linears (x # (xs @ ys))
    by simp
```

```
  hence linears (xs @ ys) linear x fdisjnt (frees x) (freess (xs @ ys))
    by auto
  hence linears xs
    using Cons by simp
  moreover have fdisjnt (frees x) (freess xs)
    proof (rule fdisjnt-subset-right)
      show freess xs |⊆| freess (xs @ ys) by simp
    qed fact
  ultimately show linears (x # xs)
    using ‹linear x› by auto

  have fdisjnt (freess xs) (freess ys)
    by (rule Cons) fact
  moreover have fdisjnt (frees x) (freess ys)
    proof (rule fdisjnt-subset-right)
      show freess ys |⊆| freess (xs @ ys) by simp
    qed fact
  ultimately show fdisjnt (freess (x # xs)) (freess ys)
    unfolding fdisjnt-alt-def
    by auto
qed (auto simp: fdisjnt-alt-def)

lemma linear-list-comb:
  assumes linear f linears xs fdisjnt (frees f) (freess xs) ¬ is-free f
  shows linear (list-comb f xs)
using assms
proof (induction xs arbitrary: f)
  case (Cons x xs)

  hence ∗: fdisjnt (frees f) (frees x |∪| freess xs)
    by simp

  have linear (list-comb (f $ x) xs)
    proof (rule Cons)
      have linear x
        using Cons by simp
      moreover have fdisjnt (frees f) (frees x)
        using ∗ by (auto intro: fdisjnt-subset-right)
      ultimately show linear (f $ x)
        using assms Cons by simp
    next
      show linears xs
        using Cons by simp
    next
      have fdisjnt (frees f) (freess xs)
        using ∗ by (auto intro: fdisjnt-subset-right)
      moreover have fdisjnt (frees x) (freess xs)
        using Cons by simp
      ultimately show fdisjnt (frees (f $ x)) (freess xs)
```

```
      by (auto intro: fdisjnt-union-left)
    qed auto
  thus ?case
    by (simp add: app-term-def)
qed auto

corollary linear-list-comb': linears xs ⟹ linear (name $$ xs)
by (auto intro: linear-list-comb simp: fdisjnt-alt-def)

lemma linear-strip-comb-cases[consumes 1]:
  assumes linear pat
  obtains (comb) s args where strip-comb pat = (Const s, args) pat = s $$ args
        | (free) s where strip-comb pat = (Free s, []) pat = Free s
using assms
proof (induction pat rule: strip-comb-induct)
  case (app t1 t2)
  show ?case
    proof (rule app.IH)
      show linear t1
        using app by simp
    next
      fix s
      assume strip-comb t1 = (Free s, [])
      hence t1 = Free s
        by (metis fst-conv snd-conv strip-comb-empty)
      hence False
        using app by simp
      thus thesis
        by simp
    next
      fix s args
      assume strip-comb t1 = (Const s, args)
      with app show thesis
        by (fastforce simp add: strip-comb-app)
    qed
next
  case (no-app t)
  thus ?case
    by (cases t) (auto simp: const-term-def)
qed

lemma wellformed-linearI: linear t ⟹ wellformed' n t
by (induct t) auto

lemma pat-cases:
  obtains (free) s where t = Free s
        | (comb) name args where linears args t = name $$ args
        | (nonlinear) ¬ linear t
proof (cases t)
```

```
  case Free
  thus thesis using free by simp
next
  case Bound
  thus thesis using nonlinear by simp
next
  case Abs
  thus thesis using nonlinear by simp
next
  case (Const name)
  have linears [] by simp
  moreover have t = name $$ [] unfolding Const by (simp add: const-term-def)
  ultimately show thesis
    by (rule comb)
next
  case (App u v)
  show thesis
    proof (cases linear t)
      case False
      thus thesis using nonlinear by simp
    next
      case True
      thus thesis
        proof (cases rule: linear-strip-comb-cases)
          case free
          thus thesis using that by simp
        next
          case (comb name args)
          moreover hence linears (snd (strip-comb t))
            using True by (blast intro: linears-strip-comb)
          ultimately have linears args
            by simp
          thus thesis using that comb by simp
        qed
    qed
qed

corollary linear-pat-cases[consumes 1]:
  assumes linear t
  obtains (free) s where t = Free s
        | (comb) name args where linears args t = name $$ args
using assms
by (metis pat-cases)

lemma linear-pat-induct[consumes 1, case-names free comb]:
  assumes linear t
  assumes ⋀s. P (Free s)
  assumes ⋀name args. linears args ⟹ (⋀arg. arg ∈ set args ⟹ P arg) ⟹ P
(name $$ args)
```

```
  shows P t
using wf-measure[of size] ‹linear t›
proof (induction t)
  case (less t)

  show ?case
    using ‹linear t›
    proof (cases rule: linear-pat-cases)
      case (free name)
      thus ?thesis
        using assms by simp
    next
      case (comb name args)
      show ?thesis
        proof (cases args = [])
          case True
          thus ?thesis
            using assms comb by fastforce
        next
          case False
          show ?thesis
            unfolding ‹t = -›
            proof (rule assms)
              fix arg
              assume arg ∈ set args

              then have (arg, t) ∈ measure size
                unfolding ‹t = -›
                by (induction args) auto

              moreover have linear arg
                using ‹arg ∈ set args› ‹linears args›
                by (auto dest: linears-linear)

              ultimately show P arg
                using less by auto
            qed fact
        qed
    qed
qed

context begin

private lemma match-subst-correctness0:
  assumes linear t
  shows case match t u of
        None ⇒ (∀ env. subst (convert-term t) env ≠ u) |
        Some env ⇒ subst (convert-term t) env = u
using assms proof (induction t arbitrary: u)
```

```
  case Free
  show ?case
    unfolding match.simps
    by (fold free-term-def) auto
next
  case Const
  show ?case
    unfolding match.simps
    by (fold const-term-def) (auto split: option.splits)
next
  case (App t₁ t₂)
  hence linear: linear t₁ linear t₂ fdisjnt (frees t₁) (frees t₂)
    by simp+

  show ?case
    proof (cases unapp u)
      case None
      then show ?thesis
        apply simp
        apply (fold app-term-def)
        apply simp
        using app-simps(3) is-app-def by blast
    next
      case (Some u')
      then obtain u₁ u₂ where u: unapp u = Some (u₁, u₂) by (cases u') auto
      hence u = app u₁ u₂ by auto

      note 1 = App(1)[OF ‹linear t₁›, of u₁]
      note 2 = App(2)[OF ‹linear t₂›, of u₂]

      show ?thesis
        proof (cases match t₁ u₁)
          case None
          then show ?thesis
            using u
            apply simp
            apply (fold app-term-def)
            using 1 by auto
        next
          case (Some env₁)
          with 1 have s1: subst (convert-term t₁) env₁ = u₁ by simp
          show ?thesis
            proof (cases match t₂ u₂)
              case None
              then show ?thesis
                using u
                apply simp
                apply (fold app-term-def)
                using 2 by auto
```

```
        next
          case (Some env₂)
          with 2 have s2: subst (convert-term t₂) env₂ = u₂ by simp

          note match = ‹match t₁ u₁ = Some env₁› ‹match t₂ u₂ = Some env₂›

          let ?env = env₁ ++ₓ env₂
          from match have frees t₁ = fmdom env₁ frees t₂ = fmdom env₂
            by (auto simp: match-dom)
              with linear have env₁ = fmrestrict-fset (frees t₁) ?env env₂ =
fmrestrict-fset (frees t₂) ?env
            apply auto
            apply (auto simp: fmfilter-alt-defs)
           apply (subst fmfilter-false; auto simp: fdisjnt-alt-def intro: fmdomI)+
            done
         with s1 s2 have subst (convert-term t₁) ?env = u₁ subst (convert-term
t₂) ?env = u₂
            using linear
            by (metis subst-restrict' convert-term-frees linear-implies-no-abs)+

          then show ?thesis
            using match unfolding ‹u = -›
            apply simp
            apply (fold app-term-def)
            by simp
        qed
      qed
    qed
qed auto

lemma match-subst-some[simp]:
  match t u = Some env ⟹ linear t ⟹ subst (convert-term t) env = u
by (metis (mono-tags) match-subst-correctness0 option.simps(5))

lemma match-subst-none:
  match t u = None ⟹ linear t ⟹ subst (convert-term t) env = u ⟹ False
by (metis (mono-tags, lifting) match-subst-correctness0 option.simps(4))

end



lemma match-matches: match t u = Some env ⟹ linear t ⟹ t ≲ u
by (metis match-subst-some linear-implies-no-abs convert-term-id matchesI)

lemma overlapping-var1I: overlapping (Free name) t
proof (intro overlappingI matchesI)
  show subst (Free name) (fmap-of-list [(name, t)]) = t
    by simp
```

next
show *subst t fmempty* = *t*
by *simp*
qed

lemma *overlapping-var2I*: *overlapping t* (*Free name*)
proof (*intro overlappingI matchesI*)
show *subst* (*Free name*) (*fmap-of-list* [(*name*, *t*)]) = *t*
by *simp*
next
show *subst t fmempty* = *t*
by *simp*
qed

lemma *non-overlapping-appI1*: *non-overlapping* t_1 u_1 ⟹ *non-overlapping* (t_1 $ t_2) (u_1 $ u_2)
unfolding *overlapping-def matches-def* **by** *auto*

lemma *non-overlapping-appI2*: *non-overlapping* t_2 u_2 ⟹ *non-overlapping* (t_1 $ t_2) (u_1 $ u_2)
unfolding *overlapping-def matches-def* **by** *auto*

lemma *non-overlapping-app-constI*: *non-overlapping* (t_1 $ t_2) (*Const name*)
unfolding *overlapping-def matches-def* **by** *simp*

lemma *non-overlapping-const-appI*: *non-overlapping* (*Const name*) (t_1 $ t_2)
unfolding *overlapping-def matches-def* **by** *simp*

lemma *non-overlapping-const-constI*: $x \neq y$ ⟹ *non-overlapping* (*Const x*) (*Const y*)
unfolding *overlapping-def matches-def* **by** *simp*

lemma *match-overlapping*:
assumes *linear* t_1 *linear* t_2
assumes *match* t_1 *u* = *Some* env_1 *match* t_2 *u* = *Some* env_2
shows *overlapping* t_1 t_2
proof −
define env_1' **where** env_1' = (*fmmap convert-term* env_1 :: (*name*, *term*) *fmap*)
define env_2' **where** env_2' = (*fmmap convert-term* env_2 :: (*name*, *term*) *fmap*)
from *assms* **have** *match* t_1 (*convert-term u* :: *term*) = *Some* env_1' *match* t_2 (*convert-term u* :: *term*) = *Some* env_2'
unfolding env_1'*-def* env_2'*-def*
by (*metis convert-term-match*)+
with *assms* **show** *?thesis*
by (*metis overlappingI match-matches*)
qed

end

Chapter 5

Terms with explicit bound variable names

theory *Nterm*
imports *Term-Class*
begin

The *nterm* type is similar to *term*, but removes the distinction between bound and free variables. Instead, there are only named variables.

datatype *nterm* =
  Nconst name |
  Nvar name |
  Nabs name nterm (Λ_n -. - [*0*, *50*] *50*) |
  Napp nterm nterm (**infixl** $\$_n$ *70*)

derive *linorder nterm*

instantiation *nterm* :: *pre-term* **begin**

definition *app-nterm* **where**
app-nterm t u = *t* $\$_n$ *u*

fun *unapp-nterm* **where**
unapp-nterm (*t* $\$_n$ *u*) = *Some* (*t*, *u*) |
unapp-nterm - = *None*

definition *const-nterm* **where**
const-nterm = *Nconst*

fun *unconst-nterm* **where**
unconst-nterm (*Nconst name*) = *Some name* |
unconst-nterm - = *None*

definition *free-nterm* **where**
free-nterm = *Nvar*

fun *unfree-nterm* **where**
unfree-nterm (*Nvar name*) = *Some name* |
unfree-nterm - = *None*

fun *frees-nterm* :: *nterm* ⇒ *name fset* **where**
frees-nterm (*Nvar x*) = {| *x* |} |
frees-nterm (t_1 $\$_n$ t_2) = *frees-nterm* t_1 |∪| *frees-nterm* t_2 |
frees-nterm (Λ_n *x*. *t*) = *frees-nterm t* − {| *x* |} |
frees-nterm (*Nconst* -) = {||}

fun *subst-nterm* :: *nterm* ⇒ (*name*, *nterm*) *fmap* ⇒ *nterm* **where**
subst-nterm (*Nvar s*) *env* = (*case fmlookup env s of Some t* ⇒ *t* | *None* ⇒ *Nvar s*) |
subst-nterm (t_1 $\$_n$ t_2) *env* = *subst-nterm* t_1 *env* $\$_n$ *subst-nterm* t_2 *env* |
subst-nterm (Λ_n *x*. *t*) *env* = (Λ_n *x*. *subst-nterm t* (*fmdrop x env*)) |
subst-nterm t env = *t*

fun *consts-nterm* :: *nterm* ⇒ *name fset* **where**
consts-nterm (*Nconst x*) = {| *x* |} |
consts-nterm (t_1 $\$_n$ t_2) = *consts-nterm* t_1 |∪| *consts-nterm* t_2 |
consts-nterm (*Nabs* - *t*) = *consts-nterm t* |
consts-nterm (*Nvar* -) = {||}

instance
by *standard*
(*auto*
simp: *app-nterm-def const-nterm-def free-nterm-def*
elim: *unapp-nterm.elims unconst-nterm.elims unfree-nterm.elims*
split: *option.splits*)

end

instantiation *nterm* :: *term* **begin**

definition *abs-pred-nterm* :: (*nterm* ⇒ *bool*) ⇒ *nterm* ⇒ *bool* **where**
[*code del*]: *abs-pred P t* ⟷ (∀ t' *x*. *t* = (Λ_n *x*. t') ⟶ *P* t' ⟶ *P t*)

instance proof (*standard*, *goal-cases*)
case (*1 P t*)
then show *?case*
by (*induction t*) (*auto simp*: *abs-pred-nterm-def const-nterm-def free-nterm-def app-nterm-def*)
next
case *3*
show *?case*
unfolding *abs-pred-nterm-def*
apply *auto*
apply (*subst fmdrop-comm*)

```
    by auto
next
  case 4
  show ?case
    unfolding abs-pred-nterm-def
    apply auto
    apply (erule-tac x = fmdrop x env₁ in allE)
    apply (erule-tac x = fmdrop x env₂ in allE)
    by (auto simp: fdisjnt-alt-def)
next
  case 5
  show ?case
    unfolding abs-pred-nterm-def
    apply clarify
    subgoal for t' x env
      apply (erule allE[where x = fmdrop x env])
      by auto
    done
next
  case 6
  show ?case
    unfolding abs-pred-nterm-def
    apply clarify
    subgoal premises prems[rule-format] for t x env
      unfolding consts-nterm.simps subst-nterm.simps frees-nterm.simps
      apply (subst prems)
      unfolding fmimage-drop fmdom-drop
      apply (rule arg-cong[where f = (|∪|) (consts t)])
      apply (rule arg-cong[where f = ffUnion])
      apply (rule arg-cong[where f = λx. consts |`| fmimage env x])
      by auto
    done
qed (auto simp: abs-pred-nterm-def)

end

lemma no-abs-abs[simp]: ¬ no-abs (Λn x. t)
by (subst no-abs.simps) (auto simp: term-cases-def)

end
```

Chapter 6

Converting between *term*s and *nterm*s

theory *Term-to-Nterm*
imports
Fresh-Class
Find-First
Term
Nterm
begin

6.1 α-equivalence

inductive *alpha-equiv* :: (*name*, *name*) *fmap* ⇒ *nterm* ⇒ *nterm* ⇒ *bool* **where**
const: *alpha-equiv env* (*Nconst x*) (*Nconst x*) |
var1: *x* |∉| *fmdom env* ⟹ *x* |∉| *fmran env* ⟹ *alpha-equiv env* (*Nvar x*) (*Nvar x*) |
var2: *fmlookup env x* = *Some y* ⟹ *alpha-equiv env* (*Nvar x*) (*Nvar y*) |
abs: *alpha-equiv* (*fmupd x y env*) *n1 n2* ⟹ *alpha-equiv env* (Λ_n *x*. *n1*) (Λ_n *y*. *n2*) |
app: *alpha-equiv env n1 n2* ⟹ *alpha-equiv env m1 m2* ⟹ *alpha-equiv env* (*n1* $\$_n$ *m1*) (*n2* $\$_n$ *m2*)

code-pred *alpha-equiv* **.**

abbreviation *alpha-eq* :: *nterm* ⇒ *nterm* ⇒ *bool* (**infixl** $\approx_\alpha$ *50*) **where**
alpha-eq n1 n2 ≡ *alpha-equiv fmempty n1 n2*

lemma *alpha-equiv-refl*[*intro?*]:
assumes *fmpred* (=) Γ
shows *alpha-equiv* Γ *t t*
using *assms* **proof** (*induction t arbitrary*: Γ)
case *Napp*
show *?case*

apply (*rule alpha-equiv.app*; *rule Napp*)
using *Napp.prems* **unfolding** *fdisjnt-alt-def* **by** *auto*
qed (*auto simp*: *fdisjnt-alt-def intro*: *alpha-equiv.intros*)

corollary *alpha-eq-refl*: *alpha-eq t t*
by (*auto intro*: *alpha-equiv-refl*)

6.2 From *Term-Class.term* to *nterm*

fun *term-to-nterm* :: *name list* ⇒ *term* ⇒ (*name*, *nterm*) *state* **where**
term-to-nterm - (Const name) = *State-Monad.return* (*Nconst name*) |
term-to-nterm - (Free name) = *State-Monad.return* (*Nvar name*) |
term-to-nterm Γ (*Bound n*) = *State-Monad.return* (*Nvar* (Γ ! *n*)) |
term-to-nterm Γ (Λ *t*) = *do* {
  n ← *fresh-create*;
  e ← *term-to-nterm* (*n* # Γ) *t*;
  State-Monad.return (Λ$_n$ *n*. *e*)
} |
term-to-nterm Γ (*t*$_1$ $ *t*$_2$) = *do* {
  e$_1$ ← *term-to-nterm* Γ *t*$_1$;
  e$_2$ ← *term-to-nterm* Γ *t*$_2$;
  State-Monad.return (*e*$_1$ $$_n$ *e*$_2$)
}

lemmas *term-to-nterm-induct* = *term-to-nterm.induct*[*case-names const free bound abs app*]

lemma *term-to-nterm*:
  assumes *no-abs t*
  shows *fst* (*run-state* (*term-to-nterm* Γ *t*) *x*) = *convert-term t*
using *assms*
apply (*induction arbitrary*: *x*)
apply *auto*
by (*auto simp*: *free-term-def free-nterm-def const-term-def const-nterm-def app-term-def app-nterm-def split-beta split*: *prod.splits*)

definition *term-to-nterm'* :: *term* ⇒ *nterm* **where**
term-to-nterm' t = *frun-fresh* (*term-to-nterm* [] *t*) (*frees t*)

lemma *term-to-nterm-mono*: *mono-state* (*term-to-nterm* Γ *x*)
by (*induction* Γ *x rule*: *term-to-nterm.induct*) (*auto intro*: *bind-mono-strong*)

lemma *term-to-nterm-vars0*:
  assumes *wellformed'* (*length* Γ) *t*
  shows *frees* (*fst* (*run-state* (*term-to-nterm* Γ *t*) *s*)) |⊆| *frees t* |∪| *fset-of-list* Γ
using *assms* **proof** (*induction* Γ *t arbitrary*: *s rule*: *term-to-nterm-induct*)
  case (*bound* Γ *i*)
  hence Γ ! *i* |∈| *fset-of-list* Γ
    including *fset.lifting* **by** *transfer auto*

thus *?case*
by (*auto simp*: *State-Monad.return-def*)
next
case (*abs* Γ *t*)
let *?x* = *next s*

from *abs* **have** *frees* (*fst* (*run-state* (*term-to-nterm* (*?x* # Γ) *t*) *?x*)) |⊆| *frees t* |∪| {|*?x*|} |∪| *fset-of-list* Γ
by *simp*
thus *?case*
by (*auto simp*: *create-alt-def split-beta*)
qed (*auto simp*: *split-beta*)

corollary *term-to-nterm-vars*:
assumes *wellformed t*
shows *frees* (*fresh-frun* (*term-to-nterm* [] *t*) *F*) |⊆| *frees t*
proof −
let *?*Γ = []
from *assms* **have** *wellformed'* (*length ?*Γ) *t*
by *simp*
hence *frees* (*fst* (*run-state* (*term-to-nterm ?*Γ *t*) (*fNext F*))) |⊆| (*frees t* |∪| *fset-of-list ?*Γ)
by (*rule term-to-nterm-vars0*)
thus *?thesis*
by (*simp add*: *fresh-fNext-def fresh-frun-def*)
qed

corollary *term-to-nterm-closed*: *closed t* ⟹ *wellformed t* ⟹ *closed* (*term-to-nterm'* *t*)
using *term-to-nterm-vars*[**where** *F* = *frees t* **and** *t* = *t*, *simplified*]
unfolding *closed-except-def term-to-nterm'-def*
by (*simp add*: *fresh-frun-def*)

lemma *term-to-nterm-consts*: *pred-state* (λ*t'*. *consts t'* = *consts t*) (*term-to-nterm* Γ *t*)
apply (*rule pred-stateI*)
apply (*induction t arbitrary*: Γ)
apply (*auto split*: *prod.splits*)
done

6.3 From *nterm* to *Term-Class.term*

fun *nterm-to-term* :: *name list* ⇒ *nterm* ⇒ *term* **where**
nterm-to-term Γ (*Nconst name*) = *Const name* |
nterm-to-term Γ (*Nvar name*) = (*case find-first name* Γ *of Some n* ⇒ *Bound n* | *None* ⇒ *Free name*) |
nterm-to-term Γ (*t* $_n$ *u*) = *nterm-to-term* Γ *t* $ *nterm-to-term* Γ *u* |
nterm-to-term Γ (Λ_n_ *x*. *t*) = Λ *nterm-to-term* (*x* # Γ) *t*

lemma *nterm-to-term*:
assumes *no-abs t fdisjnt* (*fset-of-list* Γ) (*frees t*)
shows *nterm-to-term* Γ *t* = *convert-term t*
using *assms* **proof** (*induction arbitrary*: Γ)
case (*free name*)
then show *?case*
apply *simp*
apply (*auto simp*: *free-nterm-def free-term-def fdisjnt-alt-def split*: *option.splits*)
apply (*rule find-first-none*)
by (*metis fset-of-list-elem*)
next
case (*const name*)
show *?case*
apply *simp*
by (*simp add*: *const-nterm-def const-term-def*)
next
case (*app* t_1 t_2)
then have *nterm-to-term* Γ t_1 = *convert-term* t_1 *nterm-to-term* Γ t_2 = *convert-term* t_2
by (*auto simp*: *fdisjnt-alt-def finter-funion-distrib*)
then show *?case*
apply *simp*
by (*simp add*: *app-nterm-def app-term-def*)
qed

abbreviation *nterm-to-term′* ≡ *nterm-to-term* []

lemma *nterm-to-term′*: *no-abs t* ⟹ *nterm-to-term′ t* = *convert-term t*
by (*auto simp*: *fdisjnt-alt-def nterm-to-term*)

lemma *nterm-to-term-frees*[*simp*]: *frees* (*nterm-to-term* Γ *t*) = *frees t* − *fset-of-list* Γ
proof (*induction t arbitrary*: Γ)
case (*Nvar name*)
show *?case*
proof (*cases find-first name* Γ)
case *None*
hence *name* |∉| *fset-of-list* Γ
including *fset.lifting*
by *transfer* (*metis find-first-some option.distinct*(*1*))
with *None* **show** *?thesis*
by *auto*
next
case (*Some u*)
hence *name* |∈| *fset-of-list* Γ
including *fset.lifting*
by *transfer* (*metis find-first-none option.distinct*(*1*))
with *Some* **show** *?thesis*
by *auto*

```
  qed
qed (auto split: option.splits)
```

6.4 Correctness

Some proofs in this section have been contributed by Yu Zhang.

```
lemma term-to-nterm-nterm-to-term0:
  assumes wellformed' (length Γ) t fdisjnt (fset-of-list Γ) (frees t) distinct Γ
  assumes fBall (frees t |∪| fset-of-list Γ) (λx. x ≤ s)
  shows nterm-to-term Γ (fst (run-state (term-to-nterm Γ t) s)) = t
using assms proof (induction Γ t arbitrary: s rule: term-to-nterm-induct)
  case (free Γ name)
  hence fdisjnt (fset-of-list Γ) {|name|}
    by simp
  hence name ∉ set Γ
    including fset.lifting by transfer' (simp add: disjnt-def)
  hence find-first name Γ = None
    by (rule find-first-none)
  thus ?case
    by (simp add: State-Monad.return-def)
next
  case (bound Γ i)
  thus ?case
    by (simp add: State-Monad.return-def find-first-some-index)
next
  case (app Γ t1 t2)
  have fdisjnt (fset-of-list Γ) (frees t1)
    apply (rule fdisjnt-subset-right[where N = frees t1 |∪| frees t2])
    using app by auto
  have fdisjnt (fset-of-list Γ) (frees t2)
    apply (rule fdisjnt-subset-right[where N = frees t1 |∪| frees t2])
    using app by auto

  have s: s ≤ snd (run-state (term-to-nterm Γ t1) s)
    apply (rule state-io-relD[OF term-to-nterm-mono])
    apply (rule surjective-pairing)
    done

  show ?case
    apply (auto simp: split-beta)
    subgoal
      apply (rule app)
      subgoal using app by simp
      subgoal by fact
      subgoal by fact
      using app by auto
    subgoal
      apply (rule app)
```

```
      subgoal using app by simp
      subgoal by fact
      subgoal by fact
      using app s by force+
    done
next
  case (abs Γ t)

  have next s |∉| frees t |∪| fset-of-list Γ
    using abs(5)[simplified]
    by (rule next-ge-fall)

  have nterm-to-term (next s # Γ) (fst (run-state (term-to-nterm (next s # Γ) t)
(next s))) = t
    proof (subst abs)
      show wellformed' (length (next s # Γ)) t
        using abs by auto
      show fdisjnt (fset-of-list (next s # Γ)) (frees t)
        apply simp
        apply (rule fdisjnt-insert)
        using ‹next s |∉| frees t |∪| fset-of-list Γ› abs by auto
      show distinct (next s # Γ)
        apply simp
        apply rule
       using ‹next s |∉| frees t |∪| fset-of-list Γ› apply (simp add: fset-of-list-elem)
        apply fact
        done
      show fBall (frees t |∪| fset-of-list (next s # Γ)) (λx. x ≤ next s)
        using abs(5) apply simp
        apply (rule fBall-pred-weaken[where P = λx. x ≤ s])
        subgoal
          apply simp
       by (meson dual-order.strict-implies-order dual-order.strict-trans2 fresh-class.next-ge)
        subgoal by assumption
        done
    qed auto

  thus ?case
    by (auto simp: split-beta create-alt-def)
qed (auto simp: State-Monad.return-def)

lemma term-to-nterm-nterm-to-term:
  assumes wellformed t frees t |⊆| S
  shows nterm-to-term' (frun-fresh (term-to-nterm [] t) (S |∪| Q)) = t
proof (rule term-to-nterm-nterm-to-term0)
  show wellformed' (length []) t
    using assms by simp
next
  show fdisjnt (fset-of-list []) (frees t)
```

```
    unfolding fdisjnt-alt-def by simp
next
  have fBall (S |∪| Q) (λx. x < fresh.fNext next default (S |∪| Q))
    by (metis fNext-geq-not-member fresh-fNext-def le-less-linear fBallI)
  hence fBall (S |∪| Q) (λx. x ≤ fresh.fNext next default (S |∪| Q))
    by (meson fBall-pred-weaken less-eq-name-def)
  thus fBall (frees t |∪| fset-of-list []) (λx. x ≤ fresh.fNext next default (S |∪| Q))
    using ‹frees t |⊆| S›
    by auto
qed simp

corollary term-to-nterm-nterm-to-term-simple:
  assumes wellformed t
  shows nterm-to-term' (term-to-nterm' t) = t
unfolding term-to-nterm'-def using assms
by (metis order-refl sup.idem term-to-nterm-nterm-to-term)

lemma nterm-to-term-eq:
  assumes frees u |⊆| fset-of-list (common-prefix Γ Γ')
  shows nterm-to-term Γ u = nterm-to-term Γ' u
using assms
proof (induction u arbitrary: Γ Γ')
  case (Nvar name)
  hence name ∈ set (common-prefix Γ Γ')
    unfolding frees-nterm.simps
    including fset.lifting
    by transfer' simp
  thus ?case
    by (auto simp: common-prefix-find)
next
  case (Nabs x t)
  hence ∗: frees t − {|x|} |⊆| fset-of-list (common-prefix Γ Γ')
    by simp

  show ?case
    apply simp
    apply (rule Nabs)
    using ∗ Nabs by auto
qed auto

corollary nterm-to-term-eq-closed: closed t ⟹ nterm-to-term Γ t = nterm-to-term
Γ' t
by (rule nterm-to-term-eq) (auto simp: closed-except-def)

lemma nterm-to-term-wellformed: wellformed' (length Γ) (nterm-to-term Γ t)
proof (induction t arbitrary: Γ)
  case (Nabs x t)
  hence wellformed' (Suc (length Γ)) (nterm-to-term (x # Γ) t)
    by (metis length-Cons)
```

thus *?case*
by *auto*
qed (*auto simp*: *find-first-correct split*: *option.splits*)

corollary *nterm-to-term-closed-wellformed*: *closed t* ⟹ *wellformed* (*nterm-to-term* Γ *t*)
by (*metis Ex-list-of-length nterm-to-term-eq-closed nterm-to-term-wellformed*)

lemma *nterm-to-term-term-to-nterm*:
assumes *frees t* |⊆| *fset-of-list* Γ *length* Γ = *length* Γ′
shows *alpha-equiv* (*fmap-of-list* (*zip* Γ Γ′)) *t* (*fst* (*run-state* (*term-to-nterm* Γ′ (*nterm-to-term* Γ *t*)) *s*))
using *assms* **proof** (*induction* Γ *t arbitrary*: *s* Γ′ *rule*:*nterm-to-term.induct*)
case (*4* Γ *x t*)
show *?case*
apply (*simp add*: *split-beta*)
apply (*rule alpha-equiv.abs*)
using *4.IH*[**where** Γ′ = *next s* # Γ′] *4.prems*
by (*fastforce simp*: *create-alt-def intro*: *alpha-equiv.intros*)
qed
(*force split*: *option.splits intro*: *find-first-some intro*!: *alpha-equiv.intros*
simp: *fset-of-list-elem find-first-in-map split-beta fdisjnt-alt-def*)+

corollary *nterm-to-term-term-to-nterm′*: *closed t* ⟹ *t* ≈$_\alpha$ *term-to-nterm′* (*nterm-to-term′ t*)
unfolding *term-to-nterm′-def closed-except-def*
apply (*rule nterm-to-term-term-to-nterm*[**where** Γ = [] **and** Γ′ = [], *simplified*])
by *auto*

context begin

private lemma *term-to-nterm-alpha-equiv0*:
length Γ*1* = *length* Γ*2* ⟹ *distinct* Γ*1* ⟹ *distinct* Γ*2* ⟹ *wellformed′* (*length* Γ*1*) *t1* ⟹
fresh-fin (*frees t1* |∪| *fset-of-list* Γ*1*) *s1* ⟹ *fdisjnt* (*fset-of-list* Γ*1*) (*frees t1*) ⟹
fresh-fin (*frees t1* |∪| *fset-of-list* Γ*2*) *s2* ⟹ *fdisjnt* (*fset-of-list* Γ*2*) (*frees t1*) ⟹
alpha-equiv (*fmap-of-list* (*zip* Γ*1* Γ*2*)) (*fst*(*run-state* (*term-to-nterm* Γ*1 t1*) *s1*)) (*fst* (*run-state* (*term-to-nterm* Γ*2 t1*) *s2*))
proof (*induction* Γ*1 t1 arbitrary*: Γ*2 s1 s2 rule*:*term-to-nterm-induct*)
case (*free* Γ*1 name*)
then have *name* |∉| *fmdom* (*fmap-of-list* (*zip* Γ*1* Γ*2*))
unfolding *fdisjnt-alt-def*
by *force*
moreover have *name* |∉| *fmran* (*fmap-of-list* (*zip* Γ*1* Γ*2*))
apply *rule*
apply (*subst* (*asm*) *fmran-of-list*)
apply (*subst* (*asm*) *fset-of-list-map*[*symmetric*])

apply (*subst* (*asm*) *distinct-clearjunk-id*)
subgoal
apply (*subst map-fst-zip*)
apply *fact*
apply *fact*
done
apply (*subst* (*asm*) *map-snd-zip*)
apply *fact*
using *free* **unfolding** *fdisjnt-alt-def*
by *fastforce*
ultimately show *?case*
by (*force intro*:*alpha-equiv.intros*)
next
case (*abs* Γ *t*)
have ∗: *next s1* > *s1 next s2* > *s2*
using *next-ge* **by** *force*+
with *abs* **have** *next s1* ∉ *set* Γ *next s2* ∉ *set* Γ*2*
by (*metis fBall-funion fset-of-list-elem next-ge-fall*)+
have *fresh-fin* (*frees t* |∪| *fset-of-list* Γ) (*next s1*)
fresh-fin (*frees t* |∪| *fset-of-list* Γ*2*) (*next s2*)
using ∗ *abs*
by (*smt dual-order.trans fBall-pred-weaken frees-term.simps*(*3*) *less-imp-le*)+
have *fdisjnt* (*finsert* (*next s1*) (*fset-of-list* Γ)) (*frees t*)
fdisjnt (*finsert* (*next s2*) (*fset-of-list* Γ*2*)) (*frees t*)
unfolding *fdisjnt-alt-def* **using** *abs frees-term.simps*
by (*metis fdisjnt-alt-def finter-finsert-right funionCI inf-commute next-ge-fall*)+
have *wellformed′* (*Suc* (*length* Γ*2*)) *t*
using *wellformed′.simps abs*
by (*metis Suc-eq-plus1*)
show *?case*
apply (*auto simp*: *split-beta create-alt-def*)
apply *rule*
apply (*rule abs.IH*[*of - next s2* # Γ*2*, *simplified*])
by (*fact* | *rule*)+
next
case (*app* Γ*1* t_1 t_2)
hence *wellformed′* (*length* Γ*1*) t_1 *wellformed′* (*length* Γ*1*) t_2
and *fresh-fin* (*frees* t_1 |∪| *fset-of-list* Γ*1*) *s1 fresh-fin* (*frees* t_1 |∪| *fset-of-list* Γ*2*) *s2*
and *fdisjnt* (*fset-of-list* Γ*1*) (*frees* t_1) *fdisjnt* (*fset-of-list* Γ*2*) (*frees* t_1)
and *fdisjnt* (*fset-of-list* Γ*1*) (*frees* t_2) *fdisjnt* (*fset-of-list* Γ*2*) (*frees* t_2)
using *app*
unfolding *fdisjnt-alt-def*
by (*auto simp*: *inf-sup-distrib1*)
have *s1* ≤ *snd* (*run-state* (*term-to-nterm* Γ*1* t_1) *s1*) *s2* ≤ *snd* (*run-state* (*term-to-nterm* Γ*2* t_1) *s2*)
using *term-to-nterm-mono*
by (*simp add*: *state-io-rel-def*)+
hence *fresh-fin* (*frees* t_2 |∪| *fset-of-list* Γ*1*) (*snd* (*run-state* (*term-to-nterm* Γ*1*

t_1) *s1*))
using ‹*fresh-fin* (*frees* (t_1 $ t_2) |∪| *fset-of-list* Γ*1*) *s1*›
by *force*

have *fresh-fin* (*frees* t_2 |∪| *fset-of-list* Γ*2*) (*snd* (*run-state* (*term-to-nterm* Γ*2* t_1) *s2*))
apply *rule*
using *app frees-term.simps* ‹*s2* ≤ -› *dual-order.trans*
by (*metis fbspec funion-iff*)

show *?case*
apply (*auto simp*: *split-beta create-alt-def*)
apply (*rule alpha-equiv.app*)
subgoal
apply (*rule app*)
apply (*simp* | *fact*)+
done
subgoal
apply (*rule app*)
apply (*simp* | *fact*)+
done
done
qed (*force intro*: *alpha-equiv.intros simp*: *fmlookup-of-list in-set-zip*)+

lemma *term-to-nterm-alpha-equiv*:
assumes *length* Γ*1* = *length* Γ*2* *distinct* Γ*1* *distinct* Γ*2* *closed t*
assumes *wellformed'* (*length* Γ*1*) *t*
assumes *fresh-fin* (*fset-of-list* Γ*1*) *s1 fresh-fin* (*fset-of-list* Γ*2*) *s2*
shows *alpha-equiv* (*fmap-of-list* (*zip* Γ*1* Γ*2*)) (*fst* (*run-state* (*term-to-nterm* Γ*1* *t*) *s1*)) (*fst* (*run-state* (*term-to-nterm* Γ*2* *t*) *s2*))
— An instantiated version of this lemma with Γ*1* = [] and Γ*2* = [] would not make sense because then it would just be a special case of *alpha-eq-refl*.
using *assms*
by (*simp add*: *fdisjnt-alt-def closed-except-def term-to-nterm-alpha-equiv0*)

end

global-interpretation *nrelated*: *term-struct-rel-strong* (λ*t n*. *t* = *nterm-to-term* Γ *n*) **for** Γ
proof (*standard*, *goal-cases*)
case (*5 name t*)
then show *?case* **by** (*cases t*) (*auto simp*: *const-term-def const-nterm-def split*: *option.splits*)
next
case (*6* u_1 u_2 *t*)
then show *?case* **by** (*cases t*) (*auto simp*: *app-term-def app-nterm-def split*: *option.splits*)
qed (*auto simp*: *const-term-def const-nterm-def app-term-def app-nterm-def*)

lemma *env-nrelated-closed*:
assumes *nrelated.P-env* Γ *env nenv closed-env nenv*
shows *nrelated.P-env* Γ′ *env nenv*
proof
fix *x*
from *assms* **have** *rel-option* (λ*t n*. *t* = *nterm-to-term* Γ *n*) (*fmlookup env x*) (*fmlookup nenv x*)
by *auto*
thus *rel-option* (λ*t n*. *t* = *nterm-to-term* Γ′ *n*) (*fmlookup env x*) (*fmlookup nenv x*)
using *assms*
by (*cases rule*: *option.rel-cases*) (*auto dest*: *fmdomI simp*: *nterm-to-term-eq-closed*)
qed

lemma *nrelated-subst*:
assumes *nrelated.P-env* Γ *env nenv closed-env nenv fdisjnt* (*fset-of-list* Γ) (*fmdom nenv*)
shows *subst* (*nterm-to-term* Γ *t*) *env* = *nterm-to-term* Γ (*subst t nenv*)
using *assms*
proof (*induction t arbitrary*: Γ *env nenv*)
case (*Nvar name*)
thus *?case*
proof (*cases rule*: *fmrel-cases*[**where** *x* = *name*])
case (*some* t_1 t_2)
with *Nvar* **have** *name* |∉| *fset-of-list* Γ
unfolding *fdisjnt-alt-def* **by** (*auto dest*: *fmdomI*)
hence *find-first name* Γ = *None*
including *fset.lifting* **by** *transfer*′ (*simp add*: *find-first-none*)
with *some* **show** *?thesis*
by *auto*
qed (*auto split*: *option.splits*)
next
case (*Nabs x t*)
show *?case*
apply *simp*
apply (*subst subst-drop*[*symmetric*, **where** *x* = *x*])
subgoal by *simp*
apply (*rule Nabs*)
using *Nabs* **unfolding** *fdisjnt-alt-def*
by (*auto intro*: *env-nrelated-closed*)
qed *auto*

lemma *nterm-to-term-insert-dupl*:
assumes *y* ∈ *set* (*take n* Γ) *n* ≤ *length* Γ
shows *nterm-to-term* Γ *t* = *incr-bounds* (− *1*) (*Suc n*) (*nterm-to-term* (*insert-nth n y* Γ) *t*)
using *assms*
proof (*induction t arbitrary*: *n* Γ)
case (*Nvar name*)

```
  show ?case
    proof (cases y = name)
      case True
      with Nvar obtain i where find-first name Γ = Some i i < n
        by (auto elim: find-first-some-strong)

      hence find-first name (take n Γ) = Some i
        by (rule find-first-prefix)

      show ?thesis
        apply simp
        apply (subst ‹find-first name Γ = Some i›)
        apply simp
        apply (subst find-first-append)
        apply (subst ‹find-first name (take n Γ) = Some i›)
        apply simp
        using ‹i < n› by simp
    next
      case False
      show ?thesis
        apply (simp del: insert-nth-take-drop)
        apply (subst find-first-insert-nth-neq)
        subgoal using False by simp
        by (cases find-first name Γ) auto
    qed
next
  case (Nabs x t)
  show ?case
    apply simp
    apply (subst Nabs(1)[where n = Suc n])
    using Nabs by auto
qed auto

lemma nterm-to-term-bounds-dupl:
  assumes i < length Γ j < length Γ i < j
  assumes Γ ! i = Γ ! j
  shows j |∉| bounds (nterm-to-term Γ t)
using assms
proof (induction t arbitrary: Γ i j)
  case (Nvar name)
  show ?case
    proof (cases find-first name Γ)
      case (Some k)
      show ?thesis
        proof
          assume j |∈| bounds (nterm-to-term Γ (Nvar name))
          with Some have find-first name Γ = Some j
            by simp
```

```
        moreover have find-first name Γ ≠ Some j
          proof (rule find-first-later)
            show i < length Γ j < length Γ i < j
              by fact+
          next
            show Γ ! j = name
              by (rule find-first-correct) fact
            thus Γ ! i = name
              using Nvar by simp
          qed

        ultimately show False
          by blast
      qed
    qed simp
next
  case (Nabs x t)
  show ?case
    proof
      assume j |∈| bounds (nterm-to-term Γ (Λn x. t))
      then obtain j' where j' |∈| bounds (nterm-to-term (x # Γ) t) j' > 0 j = j'
− 1
        by auto
      hence Suc j |∈| bounds (nterm-to-term (x # Γ) t)
        by simp

      moreover have Suc j |∉| bounds (nterm-to-term (x # Γ) t)
        proof (rule Nabs)
          show Suc i < length (x # Γ) Suc j < length (x # Γ) Suc i < Suc j (x #
Γ) ! Suc i = (x # Γ) ! Suc j
            using Nabs by simp+
        qed

      ultimately show False
        by blast
    qed
  qed auto

fun subst-single :: nterm ⇒ name ⇒ nterm ⇒ nterm where
subst-single (Nvar s) s' t' = (if s = s' then t' else Nvar s) |
subst-single (t1 $n t2) s' t' = subst-single t1 s' t' $n subst-single t2 s' t' |
subst-single (Λn x. t) s' t' = (Λn x. (if x = s' then t else subst-single t s' t')) |
subst-single t - - = t

lemma subst-single-eq: subst-single t s t' = subst t (fmap-of-list [(s, t')])
proof (induction t)
  case (Nabs x t)
  then show ?case
    by (cases x = s) (simp add: fmfilter-alt-defs)+
```

qed *auto*

lemma *nterm-to-term-subst-replace-bound*:
assumes *closed* u' $n \leq$ *length* Γ $x \notin$ *set* (*take* n Γ)
shows *nterm-to-term* Γ (*subst-single* u x u') = *replace-bound* n (*nterm-to-term* (*insert-nth* n x Γ) u) (*nterm-to-term* Γ u')
using *assms*
proof (*induction* u *arbitrary*: n Γ)
case (*Nvar name*)
note *insert-nth-take-drop*[*simp del*]
show *?case*
proof (*cases name* = x)
case *True*
thus *?thesis*
using *Nvar*
apply (*simp add*: *find-first-insert-nth-eq*)
apply (*subst incr-bounds-eq*[**where** k = *0*])
subgoal by *simp*
apply (*rule nterm-to-term-closed-wellformed*)
by *auto*
next
case *False*
thus *?thesis*
apply *auto*
apply (*subst find-first-insert-nth-neq*)
subgoal by *simp*
by (*cases find-first name* Γ) *auto*
qed
next
case (*Nabs* y t)
note *insert-nth-take-drop*[*simp del*]
show *?case*
proof (*cases* x = y)
case *True*
have *nterm-to-term* (y # Γ) t = *replace-bound* (*Suc* n) (*nterm-to-term* (y # *insert-nth* n y Γ) t) (*nterm-to-term* Γ u')
proof (*subst replace-bound-eq*)
show *Suc* n |∉| *bounds* (*nterm-to-term* (y # *insert-nth* n y Γ) t)
apply (*rule nterm-to-term-bounds-dupl*[**where** i = *0*])
subgoal by *simp*
subgoal using *Nabs*(*3*) **by** (*simp add*: *insert-nth-take-drop*)
subgoal by *simp*
apply *simp*
apply (*subst nth-insert-nth-index-eq*)
using *Nabs* **by** *auto*
show *nterm-to-term* (y # Γ) t = *incr-bounds* (− *1*) (*Suc* n + *1*) (*nterm-to-term* (y # *insert-nth* n y Γ) t)
apply (*subst nterm-to-term-insert-dupl*[**where** Γ = y # Γ **and** y = y **and** n = *Suc* n])

```
        using Nabs by auto
      qed
    with True show ?thesis
      by auto
  next
    case False
     have nterm-to-term (y # Γ) (subst-single t x u′) = replace-bound (Suc n)
(nterm-to-term (y # insert-nth n x Γ) t) (nterm-to-term Γ u′)
      apply (subst Nabs(1)[of Suc n])
      subgoal by fact
      subgoal using Nabs by simp
      subgoal using False Nabs by simp
      apply (subst nterm-to-term-eq-closed[where t = u′])
      using Nabs by auto
    with False show ?thesis
      by auto
  qed
qed auto

corollary nterm-to-term-subst-β:
  assumes closed u′
  shows nterm-to-term Γ (subst u (fmap-of-list [(x, u′)])) = nterm-to-term (x #
Γ) u [nterm-to-term Γ u′]β
using assms
by (rule nterm-to-term-subst-replace-bound[where n = 0, simplified, unfolded subst-single-eq])

end
```

Chapter 7

Instantiation for *HOL−ex.Unification* from session *HOL−ex*

theory *Unification-Compat*
imports
HOL−ex.Unification
Term-Class
begin

The Isabelle library provides a unification algorithm on lambda-free terms. To illustrate flexibility of the term algebra, I instantiate my class with that term type. The major issue is that those terms are parameterized over the constant and variable type, which cannot easily be supported by the classy approach, where those types are fixed to *name*. As a workaround, I introduce a class that requires the constant and variable type to be isomorphic to *name*.

hide-const (**open**) *Unification.subst*

class *is-name* =
fixes *of-name* :: *name* ⇒ *'a*
assumes *bij*: *bij of-name*
begin

definition *to-name* :: *'a* ⇒ *name* **where**
to-name = *inv of-name*

lemma *to-of-name*[*simp*]: *to-name* (*of-name a*) = *a*
unfolding *to-name-def* **using** *bij* **by** (*metis bij-inv-eq-iff*)

lemma *of-to-name*[*simp*]: *of-name* (*to-name a*) = *a*
unfolding *to-name-def* **using** *bij* **by** (*meson bij-inv-eq-iff*)

lemma *of-name-inj*: *of-name name*$_1$ = *of-name name*$_2$ ⟹ *name*$_1$ = *name*$_2$

using *bij* **by** (*metis to-of-name*)

end

instantiation *name* :: *is-name* **begin**

definition *of-name-name* :: *name* ⇒ *name* **where**
[*code-unfold*]: *of-name-name x* = *x*

instance by *standard* (*auto simp*: *of-name-name-def bij-betw-def inj-on-def*)

end

lemma [*simp*, *code-unfold*]: (*to-name* :: *name* ⇒ *name*) = *id*
unfolding *to-name-def of-name-name-def* **by** *auto*

instantiation *trm* :: (*is-name*) *pre-term* **begin**

definition *app-trm* **where**
app-trm = *Comb*

definition *unapp-trm* **where**
unapp-trm t = (*case t of Comb t u* ⇒ *Some* (*t*, *u*) | - ⇒ *None*)

definition *const-trm* **where**
const-trm n = *trm.Const* (*of-name n*)

definition *unconst-trm* **where**
unconst-trm t = (*case t of trm.Const a* ⇒ *Some* (*to-name a*) | - ⇒ *None*)

definition *free-trm* **where**
free-trm n = *Var* (*of-name n*)

definition *unfree-trm* **where**
unfree-trm t = (*case t of Var a* ⇒ *Some* (*to-name a*) | - ⇒ *None*)

primrec *consts-trm* :: *'a trm* ⇒ *name fset* **where**
consts-trm (*Var* -) = {||} |
consts-trm (*trm.Const c*) = {| *to-name c* |} |
consts-trm (*M* · *N*) = *consts-trm M* |∪| *consts-trm N*

context
includes *fset.lifting*
begin

lift-definition *frees-trm* :: *'a trm* ⇒ *name fset* **is** λ*t*. *to-name* ‘ *vars-of t*
by *auto*

end

lemma *frees-trm*[*code*, *simp*]:
frees (*Var v*) = {| *to-name v* |}
frees (*trm.Const c*) = {||}
frees (*M* · *N*) = *frees M* |∪| *frees N*
including *fset.lifting*
by (*transfer*; *auto*)+

primrec *subst-trm* :: *'a trm* ⇒ (*name*, *'a trm*) *fmap* ⇒ *'a trm* **where**
subst-trm (*Var v*) *env* = (*case fmlookup env* (*to-name v*) *of Some v'* ⇒ *v'* | - ⇒ *Var v*) |
subst-trm (*trm.Const c*) - = *trm.Const c* |
subst-trm (*M* · *N*) *env* = *subst-trm M env* · *subst-trm N env*

instance
by *standard*
(*auto*
simp: *app-trm-def unapp-trm-def const-trm-def unconst-trm-def free-trm-def unfree-trm-def of-name-inj*
split: *trm.splits option.splits*)

end

instantiation *trm* :: (*is-name*) *term* **begin**

definition *abs-pred-trm* :: (*'a trm* ⇒ *bool*) ⇒ *'a trm* ⇒ *bool* **where**
abs-pred-trm P t ⟷ *True*

instance proof (*standard*, *goal-cases*)
case (*1 P t*)
then show *?case*
proof (*induction t*)
case *Var*
then show *?case*
unfolding *free-trm-def*
by (*metis of-to-name*)
next
case *Const*
then show *?case*
unfolding *const-trm-def*
by (*metis of-to-name*)
qed (*auto simp*: *app-trm-def*)
qed (*auto simp*: *abs-pred-trm-def*)

end

lemma *assoc-alt-def*[*simp*]:
assoc x y t = (*case map-of t x of Some y'* ⇒ *y'* | - ⇒ *y*)
by (*induction t*) *auto*

lemma *subst-eq*: *Unification.subst t s* = *subst t* (*fmap-of-list s*)
by (*induction t*) (*auto split*: *option.splits simp*: *fmlookup-of-list*)

end

Chapter 8

Instantiation for λ-free terms according to Blanchette

theory *Lambda-Free-Compat*
imports *Unification-Compat Lambda-Free-RPOs.Lambda-Free-Term*
begin

Another instantiation of the algebra for Blanchette et al.'s term type [1].

hide-const (**open**) *Lambda-Free-Term.subst*

instantiation *tm* :: (*is-name*, *is-name*) *pre-term* **begin**

definition *app-tm* **where**
app-tm = *tm.App*

definition *unapp-tm* **where**
unapp-tm t = (*case t of App t u* ⇒ *Some* (*t*, *u*) | - ⇒ *None*)

definition *const-tm* **where**
const-tm n = *Hd* (*Sym* (*of-name n*))

definition *unconst-tm* **where**
unconst-tm t = (*case t of Hd* (*Sym a*) ⇒ *Some* (*to-name a*) | - ⇒ *None*)

definition *free-tm* **where**
free-tm n = *Hd* (*Var* (*of-name n*))

definition *unfree-tm* **where**
unfree-tm t = (*case t of Hd* (*Var a*) ⇒ *Some* (*to-name a*) | - ⇒ *None*)

context
  includes *fset.lifting*
begin

lift-definition *frees-tm* :: ('*a*, '*b*) *tm* ⇒ *name fset* **is** λ*t*. *to-name* ‘ *vars t*

by *auto*

lift-definition *consts-tm* :: ('*a*, '*b*) *tm* ⇒ *name fset* **is** λ*t*. *to-name* ‘ *syms t*
by *auto*

end

lemma *frees-tm*[*code*, *simp*]:
frees (*App f x*) = *frees f* |∪| *frees x*
frees (*Hd h*) = (*case h of Sym* - ⇒ *fempty* | *Var v* ⇒ {| *to-name v* |})
including *fset.lifting*
by (*transfer*; *auto split*: *hd.splits*)+

lemma *consts-tm*[*code*, *simp*]:
consts (*App f x*) = *consts f* |∪| *consts x*
consts (*Hd h*) = (*case h of Var* - ⇒ *fempty* | *Sym v* ⇒ {| *to-name v* |})
including *fset.lifting*
by (*transfer*; *auto split*: *hd.splits*)+

definition *subst-tm* :: ('*a*, '*b*) *tm* ⇒ (*name*, ('*a*, '*b*) *tm*) *fmap* ⇒ ('*a*, '*b*) *tm* **where**
subst-tm t env =
Lambda-Free-Term.subst (*fmlookup-default env* (*Hd* ∘ *Var* ∘ *of-name*) ∘ *to-name*)
t

lemma *subst-tm*[*code*, *simp*]:
subst (*App t u*) *env* = *App* (*subst t env*) (*subst u env*)
subst (*Hd h*) *env* = (*case h of*
Sym s ⇒ *Hd* (*Sym s*) |
Var x ⇒ (*case fmlookup env* (*to-name x*) *of*
Some t' ⇒ *t'*
| *None* ⇒ *Hd* (*Var x*)))
unfolding *subst-tm-def*
by (*auto simp*: *fmlookup-default-def split*: *hd.splits option.splits*)

instance
by *standard*
(*auto*
simp: *app-tm-def unapp-tm-def const-tm-def unconst-tm-def free-tm-def unfree-tm-def of-name-inj*
split: *tm.splits hd.splits option.splits*)

end

instantiation *tm* :: (*is-name*, *is-name*) *term* **begin**

definition *abs-pred-tm* :: (('*a*, '*b*) *tm* ⇒ *bool*) ⇒ ('*a*, '*b*) *tm* ⇒ *bool* **where**
abs-pred-tm P t ⟷ *True*

instance proof (*standard*, *goal-cases*)

```
  case (1 P t)
  then show ?case
    proof (induction t)
      case (Hd h)
      then show ?case
        apply (cases h)
        apply (auto simp: free-tm-def const-tm-def)
        apply (metis of-to-name)+
        done
    qed (auto simp: app-tm-def)
qed (auto simp: abs-pred-tm-def)

end

lemma apps-list-comb: apps f xs = list-comb f xs
by (induction xs arbitrary: f) (auto simp: app-tm-def)

end
```

Bibliography

[1] J. C. Blanchette, U. Waldmann, and D. Wand. Formalization of recursive path orders for lambda-free higher-order terms. *Archive of Formal Proofs*, Sept. 2016. http://isa-afp.org/entries/Lambda_Free_RPOs.html, Formal proof development.

[2] E. Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation*, 1(1):31–46, mar 1985.

[3] L. Hupel and T. Nipkow. A verified compiler from isabelle/hol to cakeml. In A. Ahmed, editor, *Programming Languages and Systems*, pages 999–1026, Cham, 2018. Springer International Publishing.

[4] M. Schmidt-Schauß and J. Siekmann. Unification algebras: An axiomatic approach to unification, equation solving and constraint solving. Technical Report SEKI-report SR-88-09, FB Informatik, Universität Kaiserslautern, 1988.

[5] C. Sternagel and R. Thiemann. Deriving class instances for datatypes. *Archive of Formal Proofs*, Mar. 2015. http://isa-afp.org/entries/Deriving.html, Formal proof development.

[6] C. Sternagel and R. Thiemann. First-order terms. *Archive of Formal Proofs*, Feb. 2018. http://isa-afp.org/entries/First_Order_Terms.html, Formal proof development.

[7] C. Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

[8] C. Urban, S. Berghofer, and C. Kaliszyk. Nominal 2. *Archive of Formal Proofs*, Feb. 2013. http://isa-afp.org/entries/Nominal2.shtml, Formal proof development.

[9] J. G. Williams. *Instantiation Theory*. Springer-Verlag, 1991.