

An Algebra for Higher-Order Terms

Lars Hupel

March 17, 2025

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-Brujin terms, terms with named variables, and Blanchette's λ -free higher-order terms [1]. Furthermore, I implement translation functions between de-Brujin terms and named terms and prove their correctness.

Contents

1	Names as a unique datatype	2
2	A monad for generating fresh names	5
2.1	Fresh monad operations as class operations	7
3	Terms	10
3.1	A simple term type, modelled after Pure's <i>term</i> type	10
3.2	A type class describing terms	10
3.3	Related work	37
3.4	Instantiation of class <i>term</i> for type <i>term</i>	37
4	Wellformedness of patterns	46
5	Terms with explicit bound variable names	56
6	Converting between <i>terms</i> and <i>nterms</i>	59
6.1	α -equivalence	59
6.2	From <i>Term-Class.term</i> to <i>nterm</i>	60
6.3	From <i>nterm</i> to <i>Term-Class.term</i>	61
6.4	Correctness	63
7	Instantiation for <i>HOL-ex.Unification</i> from session <i>HOL-ex</i>	74
8	Instantiation for λ-free terms according to Blanchette	78

Chapter 1

Names as a unique datatype

```
theory Name
imports Main
begin
```

I would like to model names as *strings*. 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
```

```
 $xs < ys \longleftrightarrow (as-string\ xs, as-string\ ys) \in lexord\ \{(u, v). (of-char\ u :: nat) < of-char\ v\}$ 
```

```
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 [] \longleftrightarrow *False*

Name [] < Name (x # xs) \longleftrightarrow *True*

Name (x # xs) < Name (y # ys) \longleftrightarrow (*of-char x::nat*) < *of-char y* \vee *x = y* \wedge
Name xs < Name ys

unfolding *less-name-def* **by** *auto*

lemma *le-name-code*[code]:

Name (x # xs) \leq Name [] \longleftrightarrow *False*

Name [] \leq Name (x # xs) \longleftrightarrow *True*

Name (x # xs) \leq Name (y # ys) \longleftrightarrow (*of-char x::nat*) < *of-char y* \vee *x = y* \wedge
Name xs \leq Name ys

unfolding *less-eq-name-def less-name-def* **by** *auto*

context begin

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

lemma *name-append-less*:

assumes *xs \neq Name []*

shows *append ys xs > ys*

proof –

have *Name (ys @ xs) > Name ys* **if** *xs \neq []* **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-Utills
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  $\Rightarrow$  'a and arb :: 'a
  assumes next-ge: next  $x > x$ 
begin
```

```
abbreviation update-next :: ('a, unit) state where
update-next  $\equiv$  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 $create :: ('a, 'a) \text{ state where}$
 $create = \text{update-next} \gg= (\lambda-. \text{State-Monad.get})$

lemma $create\text{-alt-def}[code]: create = \text{State } (\lambda a. (\text{next } a, \text{next } a))$
unfolding $create\text{-def State-Monad.update-def State-Monad.get-def State-Monad.set-def}$
 $\text{State-Monad.bind-def}$
by simp

abbreviation $\text{fresh-in} :: 'a \text{ set} \Rightarrow 'a \Rightarrow \text{bool where}$
 $\text{fresh-in } S \ s \equiv \text{Ball } S ((\geq) \ s)$

lemma $\text{next-ge-all}: \text{finite } S \Longrightarrow \text{fresh-in } S \ s \Longrightarrow \text{next } s \notin S$
by $(\text{metis antisym less-imp-le less-irrefl next-ge})$

definition $\text{Next} :: 'a \text{ set} \Rightarrow 'a \text{ where}$
 $\text{Next } S = (\text{if } S = \{\} \text{ then } \text{arb} \text{ else } \text{next } (\text{Max } S))$

lemma $\text{Next-ge-max}: \text{finite } S \Longrightarrow S \neq \{\} \Longrightarrow \text{Next } S > \text{Max } S$
unfolding $\text{Next-def using next-ge by simp}$

lemma $\text{Next-not-member-subset}: \text{finite } S' \Longrightarrow S \subseteq S' \Longrightarrow \text{Next } S' \notin S$
unfolding $\text{Next-def using next-ge}$
by $(\text{metis Max-ge Max-mono empty-iff finite-subset leD less-le-trans subset-empty})$

lemma $\text{Next-not-member}: \text{finite } S \Longrightarrow \text{Next } S \notin S$
by $(\text{rule Next-not-member-subset}) \text{ auto}$

lemma $\text{Next-geq-not-member}: \text{finite } S \Longrightarrow s \geq \text{Next } S \Longrightarrow s \notin S$
unfolding $\text{Next-def using next-ge}$
by $(\text{metis (full-types) Max-ge all-not-in-conv leD le-less-trans})$

lemma $\text{next-not-member}: \text{finite } S \Longrightarrow s \geq \text{Next } S \Longrightarrow \text{next } s \notin S$
by $(\text{meson Next-geq-not-member less-imp-le next-ge order-trans})$

lemma $\text{create-mono}[simp, \text{intro}]: \text{mono-state } create$
unfolding $create\text{-def}$
by $(\text{auto intro: bind-mono-strong})$

lemma $\text{create-strict-mono}[simp, \text{intro}]: \text{strict-mono-state } create$
unfolding $create\text{-def}$
by $(\text{rule bind-strict-mono-strong2}) \text{ auto}$

abbreviation run-fresh where
 $\text{run-fresh } m \ S \equiv \text{fst } (\text{run-state } m \ (\text{Next } S))$

abbreviation $\text{fresh-fin} :: 'a \ \text{fset} \Rightarrow 'a \Rightarrow \text{bool where}$
 $\text{fresh-fin } S \ s \equiv \text{fBall } S ((\geq) \ s)$

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

lemma *next-ge-fall*: $\text{fresh-fin } S \ s \implies \text{next } s \notin S$
by (*transfer fixing: next*) (*rule next-ge-all*)

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

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

lemma *next-not-fmember*: $s \geq \text{fNext } S \implies \text{next } s \notin S$
by *transfer* (*rule next-not-member*)

lemma *fNext-geq-not-member*: $s \geq \text{fNext } S \implies s \notin S$
by *transfer* (*rule Next-geq-not-member*)

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

lemma *fNext-not-member-subset*: $S \sqsubseteq S' \implies \text{fNext } S' \notin S$
by *transfer* (*rule Next-not-member-subset*)

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

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 *names*, underscores are appended to generate a fresh name.

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

```

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

primrec fresh-list :: ⟨nat ⇒ 'a :: fresh set ⇒ 'a list⟩ where
  ⟨fresh-list 0 = []⟩ |
  ⟨fresh-list (Suc n) A = Next A # fresh-list n (insert (Next A) A)⟩

lemma fresh-list-length[simp]: ⟨length (fresh-list n A) = n⟩
  by (induction n arbitrary: A) auto

context
  fixes A :: ⟨'a :: fresh set⟩
  assumes finite: ⟨finite A⟩
begin

lemma fresh-list-fresh: ⟨set (fresh-list n A) ∩ A = {}⟩
  using finite
  by (induction n arbitrary: A) (auto simp: Next-not-member)

lemma fresh-list-fresh-elem: ⟨x ∈ set (fresh-list n A) ⇒ x ∉ A⟩
  using fresh-list-fresh by auto

lemma fresh-list-distinct: ⟨distinct (fresh-list n A)⟩
using finite proof (induction n arbitrary: A)
  case (Suc n)
  then have ⟨Next A ∉ set (fresh-list n (insert (Next A) A))⟩
    by (meson Fresh-Class.fresh-list-fresh-elem finite.insertI insertI1)
  then show ?case
    using Suc by auto
qed simp

end

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

end

```

Chapter 3

Terms

```
theory Term-Class
imports
  Datatype-Order-Generator.Order-Generator
  Name
  Term-Utills
  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 ( $\langle \Lambda \rightarrow [\gamma 1] \gamma 1 \rangle$ ) |
  Bound nat |
  App term term (infixl  $\langle \$ \rangle \gamma 0$ )
```

```
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:

- $const :: name \Rightarrow \tau$
- $free :: name \Rightarrow \tau$

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

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) \implies *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* \implies *u* = *const* *name*
assumes *unfree-free[simp]*: *unfree* (*free* *name*) = *Some* *name*
assumes *free-unfree[dest]*: *unfree* *u* = *Some* *name* \implies *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 | \cup | *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 | \cup | *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*: $free\ name_1 = free\ name_2 \implies name_1 = name_2$
assumes *const-inject*: $const\ name_1 = const\ name_2 \implies name_1 = name_2$
assumes *app-inject*: $app\ u_1\ u_2 = app\ u_3\ u_4 \implies u_1 = u_3 \wedge u_2 = u_4$

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

definition *app-term* **where**

app-term $t\ u = t\ \$\ u$

fun *unapp-term* **where**

unapp-term $(t\ \$\ u) = Some\ (t,\ u) \mid$

unapp-term $- = None$

definition *const-term* **where**

const-term $= Const$

fun *unconst-term* **where**

unconst-term $(Const\ name) = Some\ name \mid$

unconst-term $- = None$

definition *free-term* **where**

free-term $= Free$

fun *unfree-term* **where**

unfree-term $(Free\ name) = Some\ name \mid$

unfree-term $- = None$

fun *frees-term* :: *term* \Rightarrow *name fset* **where**

frees-term $(Free\ x) = \{ \mid x \mid \}$ \mid

frees-term $(t_1\ \$\ t_2) = frees-term\ t_1 \mid \cup \mid frees-term\ t_2 \mid$

frees-term $(\Lambda\ t) = frees-term\ t \mid$

frees-term $- = \{ \mid \}$

fun *subst-term* :: *term* \Rightarrow (*name, term*) *fmap* \Rightarrow *term* **where**

subst-term $(Free\ s)\ env = (case\ fmlookup\ env\ s\ of\ Some\ t \Rightarrow t \mid None \Rightarrow Free\ s) \mid$

subst-term $(t_1\ \$\ t_2)\ env = subst-term\ t_1\ env\ \$\ subst-term\ t_2\ env \mid$

subst-term $(\Lambda\ t)\ env = \Lambda\ subst-term\ t\ env \mid$

subst-term $t\ env = t$

fun *consts-term* :: *term* \Rightarrow *name fset* **where**

consts-term $(Const\ x) = \{ \mid x \mid \}$ \mid

consts-term $(t_1\ \$\ t_2) = consts-term\ t_1 \mid \cup \mid consts-term\ t_2 \mid$

consts-term $(\Lambda\ t) = consts-term\ t \mid$

consts-term $- = \{ \mid \}$

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 \Rightarrow name fset **where**
freess = *ffUnion* \circ *fset-of-list* \circ *map frees*

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

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

abbreviation *id-env* :: (name, 'a) fmap \Rightarrow bool **where**
id-env \equiv *fmpred* ($\lambda x y. y = \text{free } x$)

definition *closed-except* :: 'a \Rightarrow name fset \Rightarrow bool **where**
closed-except *t* *S* \longleftrightarrow *frees* *t* | \subseteq | *S*

abbreviation *closed* :: 'a \Rightarrow bool **where**
closed *t* \equiv *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*:

$unapp\ u = Some\ x \implies unconst\ u = None$
 $unapp\ u = Some\ x \implies unfree\ u = None$
 $unconst\ u = Some\ y \implies unapp\ u = None$
 $unconst\ u = Some\ y \implies unfree\ u = None$
 $unfree\ u = Some\ z \implies unconst\ u = None$
 $unfree\ u = Some\ z \implies unapp\ u = None$

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**
 $is-const\ t \iff (unconst\ t \neq None)$

definition *const-name* **where**

$const_name\ t = (case\ unconst\ t\ of\ Some\ name\ \Rightarrow\ name)$

lemma *is-const-simps*[simp]:

$is_const\ (const\ name)$

$\neg\ is_const\ (app\ t\ u)$

$\neg\ is_const\ (free\ name)$

unfolding *is-const-def* **by** *simp+*

lemma *const-name-simps*[simp]:

$const_name\ (const\ name) = name$

$is_const\ t \Longrightarrow\ const\ (const_name\ t) = t$

unfolding *const-name-def is-const-def* **by** *auto*

definition *is-free* **where**

$is_free\ t \longleftrightarrow (unfree\ t \neq\ None)$

definition *free-name* **where**

$free_name\ t = (case\ unfree\ t\ of\ Some\ name\ \Rightarrow\ name)$

lemma *is-free-simps*[simp]:

$is_free\ (free\ name)$

$\neg\ is_free\ (const\ name)$

$\neg\ is_free\ (app\ t\ u)$

unfolding *is-free-def* **by** *simp+*

lemma *free-name-simps*[simp]:

$free_name\ (free\ name) = name$

$is_free\ t \Longrightarrow\ free\ (free_name\ t) = t$

unfolding *free-name-def is-free-def* **by** *auto*

definition *is-app* **where**

$is_app\ t \longleftrightarrow (unapp\ t \neq\ None)$

definition *left* **where**

$left\ t = (case\ unapp\ t\ of\ Some\ (l,\ -) \Rightarrow\ l)$

definition *right* **where**

$right\ t = (case\ unapp\ t\ of\ Some\ (-,\ r) \Rightarrow\ r)$

lemma *app-simps*[simp]:

$\neg\ is_app\ (const\ name)$

$\neg\ is_app\ (free\ name)$

$is_app\ (app\ t\ u)$

unfolding *is-app-def* **by** *simp+*

lemma *left-right-simps*[simp]:

$left\ (app\ l\ r) = l$

$right\ (app\ l\ r) = r$

$is_app\ t \Longrightarrow\ app\ (left\ t)\ (right\ t) = t$

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

definition *ids* :: 'a \Rightarrow name fset **where**
ids t = frees t \cup consts t

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

abbreviation *closed-env* :: (name, 'a) fmap \Rightarrow bool **where**
closed-env \equiv 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 \Rightarrow bool) \Rightarrow 'a \Rightarrow bool

assumes

raw-induct[*case-names const free app abs*]:

$(\bigwedge \text{name. } P (\text{const name})) \implies$
 $(\bigwedge \text{name. } P (\text{free name})) \implies$
 $(\bigwedge t_1 t_2. P t_1 \implies P t_2 \implies P (\text{app } t_1 t_2)) \implies$
 $(\bigwedge t. \text{abs-pred } P t) \implies$
 $P t$

assumes

raw-subst-id: *abs-pred* ($\lambda t. \forall \text{env. id-env env} \longrightarrow \text{subst } t \text{ env} = t$) t **and**

raw-subst-drop: *abs-pred* ($\lambda t. x \notin \text{frees } t \longrightarrow (\forall \text{env. subst } t (\text{fmdrop } x \text{ env}) = \text{subst } t \text{ env})$) t **and**

raw-subst-indep: *abs-pred* ($\lambda t. \forall \text{env}_1 \text{env}_2. \text{closed-env env}_2 \longrightarrow \text{fdisjnt} (\text{fmdom env}_1) (\text{fmdom env}_2) \longrightarrow \text{subst } t (\text{env}_1 ++_f \text{env}_2) = \text{subst} (\text{subst } t \text{ env}_2) \text{env}_1$) t **and**

raw-subst-frees: *abs-pred* ($\lambda t. \forall \text{env. closed-env env} \longrightarrow \text{frees} (\text{subst } t \text{ env}) = \text{frees } t \text{ |-| fmdom env}$) t **and**

raw-subst-consts': *abs-pred* ($\lambda a. \forall x. \text{consts} (\text{subst } a x) = \text{consts } a \cup \text{ffUnion} (\text{consts } |' \text{fmimage } x (\text{frees } a))$) t **and**

abs-pred-trivI: $P t \implies \text{abs-pred } P t$

begin

lemma *subst-id*: *id-env env* \implies *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*: $x \notin \text{frees } t \implies \text{subst } t (\text{fmdrop } x \text{ env}) = \text{subst } t \text{ env}$

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 fmllookup-ran-iff fmllookup-image-iff fmllookup-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 $\bigwedge t_1\ t_2\ u_1\ u_2\ env_1\ env_2. P\ t_1\ u_1\ env_1 \implies match\ t_1\ u_1 = Some\ env_1 \implies P\ t_2\ u_2\ env_2 \implies match\ t_2\ u_2 = Some\ env_2 \implies P\ (t_1\ \$\ t_2)\ (app\ u_1\ u_2)\ (env_1\ ++_f\ env_2)$

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*: $match\ p\ t = Some\ env \implies fmdom\ env = frees\ p$

by (*induction p arbitrary: t env*)

(*fastforce split: option.splits if-splits elim: option-bindE*)⁺

lemma *match-vars*: $match\ p\ t = Some\ env \implies fmpred\ (\lambda\ u. frees\ u\ |\subseteq|\ frees\ t)\ env$

proof (*induction p t env rule: match-some-induct*)

case (*app t₁ t₂ u₁ u₂ env₁ env₂*)

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 $consts\ t\ |\subseteq|\ S\ fmpred\ (\lambda\ u. consts\ u\ |\subseteq|\ S)\ env$

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

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

lemma subst-restrict:

assumes $\text{frees } t \mid\subseteq\mid M$

shows $\text{subst } t \text{ (fmrestrict-fset } M \text{ env)} = \text{subst } t \text{ env}$

proof –

have *: $\text{fmrestrict-fset } M \text{ env} = \text{fmdrop-fset (fmdom env - } M) \text{ env}$

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]: $\text{subst } t \text{ (fmrestrict-fset (frees } t) \text{ env)} = \text{subst } t \text{ env}$

by (simp add: subst-restrict)

corollary subst-cong:

assumes $\bigwedge x. x \mid\in\mid \text{frees } t \implies \text{fmlookup } \Gamma_1 x = \text{fmlookup } \Gamma_2 x$

shows $\text{subst } t \Gamma_1 = \text{subst } t \Gamma_2$

proof –

have $\text{fmrestrict-fset (frees } t) \Gamma_1 = \text{fmrestrict-fset (frees } t) \Gamma_2$

apply (rule fmap-ext)

using assms by simp

thus ?thesis

by (metis subst-restrict')

qed

corollary subst-add-disjnt:

assumes $\text{fdisjnt (frees } t) \text{ (fmdom env}_1)$

shows $\text{subst } t \text{ (env}_1 \text{ ++}_f \text{ env}_2) = \text{subst } t \text{ env}_2$

proof –

have $\text{subst } t \text{ (env}_1 \text{ ++}_f \text{ env}_2) = \text{subst } t \text{ (fmrestrict-fset (frees } t) \text{ (env}_1 \text{ ++}_f \text{ env}_2))$

by (metis subst-restrict')

also have $\dots = \text{subst } t \text{ (fmrestrict-fset (frees } t) \text{ env}_1 \text{ ++}_f \text{ fmrestrict-fset (frees } t) \text{ env}_2)$

by simp

also have $\dots = \text{subst } t \text{ (fmempty ++}_f \text{ fmrestrict-fset (frees } t) \text{ env}_2)$

unfolding fmfiter-alt-defs

apply (subst fmfiter-false)

using assms

by (auto simp: fdisjnt-alt-def intro: fmdomI)

also have $\dots = \text{subst } t \text{ (fmrestrict-fset (frees } t) \text{ env}_2)$

by simp

also have $\dots = \text{subst } t \text{ env}_2$

by simp

finally show *?thesis* .
qed

corollary *subst-add-shadowed-env*:

assumes *frees t | \subseteq | fmdom env₂*
shows *subst t (env₁ ++_f env₂) = subst t env₂*

proof –

have *subst t (env₁ ++_f env₂) = subst t (fmdrop-fset (fmdom env₂) env₁ ++_f env₂)*

by (*subst fmadd-drop-left-dom*) *rule*

also have *... = subst t (fmrestrict-fset (frees t) (fmdrop-fset (fmdom env₂) env₁ ++_f env₂))*

by (*metis subst-restrict'*)

also have *... = subst t (fmrestrict-fset (frees t) (fmdrop-fset (fmdom env₂) env₁ ++_f fmrestrict-fset (frees t) env₂))*

by *simp*

also have *... = subst t (fmempty ++_f fmrestrict-fset (frees t) env₂)*

unfolding *fmfilter-alt-defs*

using *fsubsetD[OF assms]*

by *auto*

also have *... = subst t env₂*

by *simp*

finally show *?thesis* .

qed

corollary *subst-restrict-closed: closed-except t S \implies 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 \implies subst t env = t*

by (*simp add: subst-closed-except-id fdisjnt-alt-def*)

corollary *subst-closed-preserved: closed t \implies closed (subst t env)*

by (*simp add: subst-closed-except-preserved fdisjnt-alt-def*)

context begin

private lemma *subst-indep0*:

assumes *closed-env env₂ fdisjnt (fmdom env₁) (fmdom env₂)*
shows $\text{subst } t \text{ (env}_1 \text{ ++}_f \text{ env}_2) = \text{subst (subst } t \text{ env}_2) \text{ env}_1$
using *assms* **proof** (*induction t arbitrary: env₁ env₂ rule: raw-induct*)
case (*free name*)
show *?case*
using $\langle \text{closed-env env}_2 \rangle$
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 $\text{subst } t \text{ (}\Gamma \text{ ++}_f \Gamma') = \text{subst (subst } t \Gamma') \Gamma$
proof –
have $\text{subst } t \text{ (}\Gamma \text{ ++}_f \Gamma') = \text{subst } t \text{ (fmrestrict-fset (frees } t) \text{ (}\Gamma \text{ ++}_f \Gamma'))$
by (*metis subst-restrict'*)
also have $\dots = \text{subst } t \text{ (fmrestrict-fset (frees } t) \Gamma \text{ ++}_f \Gamma')$
by (*smt fmllookup-add fmllookup-restrict-fset subst-cong*)
also have $\dots = \text{subst } t \text{ (fmrestrict-fset (frees } t \text{ |-| fmdom } \Gamma') \Gamma \text{ ++}_f \Gamma')$
by (*rule subst-cong*) (*simp add: fmfilter-alt-defs(5)*)
also have $\dots = \text{subst (subst } t \Gamma') \text{ (fmrestrict-fset (frees } t \text{ |-| fmdom } \Gamma') \Gamma)$
apply (*rule subst-indep0[OF assms]*)
using *fmdom-restrict-fset*
unfolding *fdisjnt-alt-def*
by *auto*
also have $\dots = \text{subst (subst } t \Gamma') \text{ (fmrestrict-fset (frees (subst } t \Gamma')) \Gamma)$
using *assms* **by** (*auto simp: subst-frees*)
also have $\dots = \text{subst (subst } t \Gamma') \Gamma$
by *simp*
finally show *?thesis* .
qed

lemma *subst-indep'*:

assumes *closed-env Γ' fdisjnt (fmdom Γ') (fmdom Γ)*
shows $\text{subst } t \text{ (}\Gamma' \text{ ++}_f \Gamma) = \text{subst (subst } t \Gamma') \Gamma$
using *assms* **by** (*metis subst-indep fmadd-disjnt*)

lemma *subst-twice*:

assumes $\Gamma' \subseteq_f \Gamma$ *closed-env Γ'*
shows $\text{subst (subst } t \Gamma') \Gamma = \text{subst } t \Gamma$
proof –
have $\text{subst (subst } t \Gamma') \Gamma = \text{subst } t \text{ (}\Gamma \text{ ++}_f \Gamma')$
apply (*rule subst-indep[symmetric]*)

```

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

end

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

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

context begin

private lemma matches-alt-def0:
  assumes length ps = length vs
  shows map-option (λenv. m ++f env) (matches ps vs) = map-option (foldl (++f)
  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 matches 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 matches-alt-def:
  assumes length ps = length vs
  shows matches ps vs = map-option (foldl (++f) 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*]: $\text{length } xs \neq \text{length } ys \implies \text{matchs } xs \text{ } ys = \text{None}$
by (*induct xs ys rule: matchs.induct*) *fastforce*+

corollary *matchs-some-eq-length*: $\text{matchs } xs \text{ } ys = \text{Some } env \implies \text{length } xs = \text{length } ys$
by (*metis option.distinct(1) matchs-neq-length-none*)

lemma *matchs-app*[*simp*]:
assumes $\text{length } xs_2 = \text{length } ys_2$
shows $\text{matchs } (xs_1 @ xs_2) (ys_1 @ ys_2) =$
 $\text{matchs } xs_1 \text{ } ys_1 \gg (\lambda env_1. \text{matchs } xs_2 \text{ } ys_2 \gg (\lambda env_2. \text{Some } (env_1 ++_f env_2)))$
using *assms*
by (*induct xs_1 ys_1 rule: matchs.induct*) *fastforce*+

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

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

fun *find-match* :: $(\text{term} \times 'a) \text{ list} \Rightarrow 'a \Rightarrow ((\text{name}, 'a) \text{ fmap} \times \text{term} \times 'a) \text{ option}$
where
find-match [] - = *None* |
find-match ((*pat*, *rhs*) # *cs*) *t* =
 (*case match pat t of*
 Some env \Rightarrow *Some* (*env*, *pat*, *rhs*)
 | *None* \Rightarrow *find-match cs t*)

lemma *find-match-map*:
 $\text{find-match } (\text{map } (\lambda(\text{pat}, t). (\text{pat}, \text{f pat } t))) \text{ } cs \text{ } t =$
 $\text{map-option } (\lambda(\text{env}, \text{pat}, \text{rhs}). (\text{env}, \text{pat}, \text{f pat } \text{rhs})) (\text{find-match } cs \text{ } t)$
by (*induct cs*) (*auto split: option.splits*)

lemma *find-match-elim*:

assumes $\text{find-match } cs \ t = \text{Some } (env, pat, rhs)$
shows $(pat, rhs) \in \text{set } cs \ \text{match } pat \ t = \text{Some } env$
using $assms$
by $(\text{induct } cs) \ (\text{auto } \text{split}: \text{option.splits})$

lemma $\text{match-subst-closed}$:

assumes $\text{match } pat \ t = \text{Some } env \ \text{closed-except } rhs \ (\text{frees } pat) \ \text{closed } t$
shows $\text{closed } (\text{subst } rhs \ env)$
using $assms$
by $(\text{smt } \text{fminusE } \text{fmpred-iff } \text{fset-mp } \text{fsubsetI } \text{closed-except-def } \text{match-vars } \text{match-dom } \text{subst-frees})$

fun $\text{rewrite-step} :: (\text{term} \times 'a) \Rightarrow 'a \Rightarrow 'a \ \text{option}$ **where**
 $\text{rewrite-step } (t_1, t_2) \ u = \text{map-option } (\text{subst } t_2) \ (\text{match } t_1 \ u)$

abbreviation $\text{rewrite-step}' :: (\text{term} \times 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$ $(\langle - / \vdash / - \rightarrow / - \rangle$
 $[50, 0, 50] \ 50)$ **where**
 $r \vdash t \rightarrow u \equiv \text{rewrite-step } r \ t = \text{Some } u$

lemma $\text{rewrite-step-closed}$:

assumes $\text{frees } t_2 \ |\subseteq| \ \text{frees } t_1 \ (t_1, t_2) \vdash \ u \rightarrow \ u' \ \text{closed } u$
shows $\text{closed } u'$

proof –

from $assms$ **obtain** env **where** $*$: $\text{match } t_1 \ u = \text{Some } env$
by $auto$
then have $\text{closed } (\text{subst } t_2 \ env)$
apply $(\text{rule } \text{match-subst-closed}[\text{where } pat = t_1 \ \text{and } t = u])$
using $assms$ **unfolding** closed-except-def **by** $auto$
with $*$ **show** $?thesis$
using $assms$ **by** $auto$

qed

definition $\text{matches} :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ **(infix** $\langle \lesssim \rangle$ $50)$ **where**
 $t \lesssim u \iff (\exists \ env. \ \text{subst } t \ env = u)$

lemma $\text{matchesI}[\text{intro}]$: $\text{subst } t \ env = u \implies t \lesssim u$
unfolding matches-def **by** $auto$

lemma $\text{matchesE}[\text{elim}]$:

assumes $t \lesssim u$
obtains env **where** $\text{subst } t \ env = u$
using $assms$ **unfolding** matches-def **by** $blast$

definition $\text{overlapping} :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ **where**
 $\text{overlapping } s \ t \iff (\exists \ u. \ s \lesssim u \wedge t \lesssim u)$

lemma overlapping-refl : $\text{overlapping } t \ t$
unfolding overlapping-def matches-def **by** $blast$

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

lemma *overlappingI[intro]*: $s \lesssim u \implies t \lesssim u \implies overlapping\ s\ t$
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-neg*: $non-overlapping\ t\ u \implies t \neq u$
by (*metis overlapping-refl*)

end

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

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-elim*, 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  $\Rightarrow$  if-const name |
  None  $\Rightarrow$  (case unfree t of
    Some name  $\Rightarrow$  if-free name |
    None  $\Rightarrow$ 
      (case unapp t of
        Some (t, u)  $\Rightarrow$  if-app t u
        | None  $\Rightarrow$  otherwise)))

```

lemma *term-cases-cong*[*fundef-cong*]:

```

assumes t = u otherwise1 = otherwise2
assumes ( $\bigwedge$ name. t = const name  $\Rightarrow$  if-const1 name = if-const2 name)
assumes ( $\bigwedge$ name. t = free name  $\Rightarrow$  if-free1 name = if-free2 name)
assumes ( $\bigwedge$ u1 u2. t = app u1 u2  $\Rightarrow$  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  $\bigwedge$ 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 \Rightarrow 'a \times 'a list **where**

```

[simp del]: strip-comb t =
  (case unapp t of
    Some (t, u)  $\Rightarrow$ 
      (let (f, args) = strip-comb t in (f, args @ [u]))
    | None  $\Rightarrow$  (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 \implies *strip-comb* t = (t, [])
by (subst *strip-comb.simps*; auto)+

lemma *strip-comb-induct*[case-names app no-app]:
 assumes $\bigwedge x y. P x \implies P$ (app x y)
 assumes $\bigwedge t. \text{unapp } t = \text{None} \implies P t$
 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*: $t' \in \text{set} (\text{snd} (\text{strip-comb } t)) \implies \text{size } t' < \text{size } t$
by (induction t rule: *strip-comb-induct*) (auto split: *prod.splits*)

lemma *sstrip-comb-termination*[termination-simp]:
 $(f, ts) = \text{strip-comb } t \implies t' \in \text{set } ts \implies \text{size } t' < \text{size } t$
by (*metis* *snd-conv strip-comb-size*)

lemma *strip-comb-empty*: $\text{snd} (\text{strip-comb } t) = [] \implies \text{fst} (\text{strip-comb } t) = t$
by (induction t rule: *strip-comb-induct*) (auto split: *prod.splits*)

lemma *strip-comb-app*: $\text{fst} (\text{strip-comb} (\text{app } t u)) = \text{fst} (\text{strip-comb } t)$
by (*simp* split: *prod.splits*)

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

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

corollary *list-comb-snoc*: $\text{app} (\text{list-comb } f xs) y = \text{list-comb } f (xs @ [y])$
by *simp*

lemma *list-comb-size*[simp]: $\text{size} (\text{list-comb } f xs) = \text{size } f + \text{size-list } \text{size } xs$

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

lemma *subst-list-comb*: *subst (list-comb f xs) env = list-comb (subst f env) (map*
(λt. 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  $\Rightarrow$  nat where
[simp del]: left-nesting t = term-cases ( $\lambda$ -. 0) ( $\lambda$ -. 0) ( $\lambda$ 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 Conswhere 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) (λt 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  $\bigwedge \text{name. } P \text{ (free name)}$ 
  assumes  $\bigwedge \text{name. } P \text{ (const name)}$ 
  assumes  $\bigwedge t_1 t_2. P t_1 \implies \text{no-abs } t_1 \implies P t_2 \implies \text{no-abs } t_2 \implies P \text{ (app } t_1 t_2)$ 
  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 u1 u2)
      with assms have P u1 P u2
        using 1 by auto
      with assms  $\langle \text{no-abs } t \rangle$  show ?thesis
        unfolding  $\langle t = \rightarrow \rangle$  by auto
    next
      case other
      then show ?thesis
        using  $\langle \text{no-abs } t \rangle$ 
        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 matches:
  assumes matches pats ts = Some env list-all P ts
  shows fmpred (λ-. P) env
using assms
by (induction pats ts arbitrary: env rule: matches.induct) (auto elim!: option-bindE
intro: match)

end

end

locale subst-syntactic-and = simple-syntactic-and +
  assumes subst: P t  $\implies$  fmpred (λ-. P) env  $\implies$  P (subst t env)
begin

lemma rewrite-step:
  assumes (lhs, rhs)  $\vdash$  t  $\rightarrow$  t' P t P rhs
  shows P t'
using assms by (auto intro: match subst)

end

locale simple-syntactic-or =
  fixes P :: 'a::term  $\Rightarrow$  bool
  assumes app: P (app t u)  $\longleftrightarrow$  P t  $\vee$  P u
begin

context
  notes app[simp]
begin

lemma list-comb: P (list-comb f xs)  $\longleftrightarrow$  P f  $\vee$  list-ex P xs
by (induction xs arbitrary: f) auto

lemma match:
  assumes match pat t = Some env  $\neg$  P t
  shows fmpred (λ- t.  $\neg$  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* $\lambda t. \text{closed-except } t \text{ for } S$
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 $P\text{-}t\text{-}const: P t (const \text{ name}) \Longrightarrow t = const \text{ name}$

assumes $P\text{-}const\text{-}const: P (const \text{ name}) (const \text{ name})$

assumes $P\text{-}t\text{-}app: P t (app u_1 u_2) \Longrightarrow \exists t_1 t_2. t = app t_1 t_2 \wedge P t_1 u_1 \wedge P t_2 u_2$

assumes $P\text{-}app\text{-}app: P t_1 u_1 \Longrightarrow P t_2 u_2 \Longrightarrow P (app t_1 t_2) (app u_1 u_2)$

begin

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

lemma *related-match:*

assumes $match x u = Some env P t u$

obtains env' **where** $match x t = Some env' P\text{-}env env' env$

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 $P\text{-}t\text{-}app[OF \langle P t (app w_1 w_2) \rangle]$ **by** *auto*

with app **obtain** $env_1' env_2'$

where $match v_1 u_1 = Some env_1' P\text{-}env env_1' env_1$

and $match v_2 u_2 = Some env_2' P\text{-}env env_2' env_2$

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.prem)*

show $match (v_1 \$ v_2) t = Some (env_1' ++_f env_2')$

unfolding $\langle t = \rightarrow \rangle$ **by** *fact*

next

show $P\text{-}env (env_1' ++_f env_2') (env_1 ++_f env_2)$

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.prem)
    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
qed

end

locale term-struct-rel-strong = term-struct-rel +
  assumes P-const-t: P (const name) t  $\implies$  t = const name
  assumes P-app-t: P (app u1 u2) t  $\implies$   $\exists$  t1 t2. t = app t1 t2  $\wedge$  P u1 t1  $\wedge$  P u2
  t2
begin

```

```

lemma unconst-rel:  $P\ t\ u \implies \text{unconst } t = \text{unconst } u$ 
by (metis P-const-t P-t-const const-name-simps(2) is-const-def unconst-const)

lemma unapp-rel:  $P\ t\ u \implies \text{rel-option } (\text{rel-prod } P\ P)\ (\text{unapp } t)\ (\text{unapp } u)$ 
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  $P\ \text{env}$  (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  $\Rightarrow$  ' $b$ ::term where
[simp del]: convert-term  $t = \text{term-cases } \text{const } \text{free } (\lambda t\ u.\ \text{app } (\text{convert-term } t)\ (\text{convert-term } u))\ \text{undefined } t$ 

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 t1 t2)
  from  $\langle \text{no-abs } t' \rangle$  show ?case
    apply cases
    using app by (auto dest: term-inject)
qed

lemma convert-term-idem:
  assumes no-abs t
  shows  $\text{convert-term } (\text{convert-term } t) = \text{convert-term } t$ 
using assms by (induction t) auto

lemma convert-term-frees[simp]:
  assumes no-abs t
  shows  $\text{frees } (\text{convert-term } t) = \text{frees } t$ 
using assms by induction auto

lemma convert-term-consts[simp]:
  assumes no-abs t
  shows  $\text{consts } (\text{convert-term } t) = \text{consts } t$ 
using assms by induction auto

```

The following lemma does not generalize to when $\text{match } t \ u = \text{None}$. 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 $\exists t. \text{is-abs } t$, 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 (fmap 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  $\Rightarrow$  bool)  $\Rightarrow$  term  $\Rightarrow$  bool where
[code del]: abs-pred P t  $\longleftrightarrow$ 
  ( $\forall x. t = \text{Bound } x \longrightarrow P t$ )  $\wedge$ 
  ( $\forall t'. t = \Lambda t' \longrightarrow P t' \longrightarrow P t$ )

```

```

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]:  $\neg$  is-const (Free name)
unfolding is-const-def by simp

lemma is-free-app[simp]:  $\neg$  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)  $\implies$  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]:  $\neg$  is-free (name $$ args)
by (fastforce dest: list-comb-free simp: const-term-def)

corollary const-list-comb-neq-free[simp]: name $$ args  $\neq$  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  $\implies$ 
        (case those (map2 match ps vs) of
          Some envs  $\implies$  Some (foldl (++) env envs)
          | None  $\implies$  None)
      | None  $\implies$  None)
using assms

```


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

lemma *matches-match-list-comb[simp]*: $\text{match (name \$\$ xs) (name \$\$ ys) = matches xs ys}$

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 $\text{match (name \$\$ xs \$ x) (name \$\$ ys) = matches (xs @ [x]) ys}$

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 \Rightarrow nat fset where*

bounds (Bound i) = { | i | }

bounds (t₁ \$ t₂) = bounds t₁ \cup bounds t₂ |

bounds (Λ t) = ($\lambda i. i - 1$) \upharpoonright (bounds t - { | 0 | }) |

bounds - = { | | }

definition *shift-nat :: nat \Rightarrow int \Rightarrow nat where*

[simp]: shift-nat n k = (if k \geq 0 then n + nat k else n - nat |k|)

fun *incr-bounds :: int \Rightarrow nat \Rightarrow term \Rightarrow term where*

incr-bounds inc lev (Bound i) = (if i \geq 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 (t₁ \$ t₂) = incr-bounds inc lev t₁ \$ incr-bounds inc lev t₂ |

incr-bounds - - t = t

lemma *incr-bounds-frees[simp]*: $\text{frees (incr-bounds n k t) = frees t}$

by (*induction n k t rule: incr-bounds.induct*) *auto*

lemma *incr-bounds-zero[simp]*: $\text{incr-bounds 0 i t = t}$

by (*induct t arbitrary: i*) *auto*

fun *replace-bound :: nat \Rightarrow term \Rightarrow term \Rightarrow 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 (t₁ \$ t₂) t = replace-bound lev t₁ t \$ replace-bound lev t₂ t |

replace-bound lev $(\Lambda u) t = \Lambda \text{replace-bound } (lev + 1) u t \mid$
replace-bound - t - = t

abbreviation *β -reduce* $:: term \Rightarrow term \Rightarrow term \ (\langle - \ [-]_{\beta} \rangle)$ **where**
 $t [u]_{\beta} \equiv \text{replace-bound } 0 t u$

lemma *replace-bound-frees*: *frees* (*replace-bound* $n t t'$) $\mid \subseteq \mid$ *frees* $t \mid \cup \mid$ *frees* t'
by (*induction* $n t t'$ *rule*: *replace-bound.induct*) *auto*

lemma *replace-bound-eq*:
assumes $i \mid \notin \mid$ *bounds* t
shows *replace-bound* $i t t' = \text{incr-bounds } (-1) (i + 1) t$
using *assms*
by (*induct* t *arbitrary*: i) *force+*

fun *wellformed'* $:: nat \Rightarrow term \Rightarrow bool$ **where**
wellformed' $n (t_1 \$ t_2) \longleftrightarrow \text{wellformed}' n t_1 \wedge \text{wellformed}' n t_2 \mid$
wellformed' $n (\text{Bound } n')$ $\longleftrightarrow n' < n \mid$
wellformed' $n (\Lambda t) \longleftrightarrow \text{wellformed}' (n + 1) t \mid$
wellformed' $- - \longleftrightarrow \text{True}$

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

abbreviation *wellformed* $:: term \Rightarrow bool$ **where**
wellformed $\equiv \text{wellformed}' 0$

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

lemma *wellformed-replace-bound-eq*: *wellformed* $t \implies \text{replace-bound } k t u = t$
by (*rule* *wellformed'-replace-bound-eq*) *simp+*

lemma *incr-bounds-eq*: $n \geq k \implies \text{wellformed}' k t \implies \text{incr-bounds } i n t = t$
by (*induct* t *arbitrary*: $k n$) *force+*

lemma *incr-bounds-subst*:
assumes $\bigwedge t. t \in \text{fmran}' \text{env} \implies \text{wellformed } t$
shows *incr-bounds* $i n (\text{subst } t \text{env}) = \text{subst } (\text{incr-bounds } i n t) \text{env}$
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 matches xs ys = Some env1
  by blast

show ?case
proof (rule snoc(3))
  show t = name $$ (ys @ [y])
    unfolding ⟨t = -⟩ ⟨t' = -⟩
    by simp
  next
  have matches [x] [y] = Some env2
    using ⟨match x y = -⟩ by simp
  thus matches (xs @ [x]) (ys @ [y]) = Some env
    unfolding ⟨env = -⟩ using ⟨matches 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 $\text{match } (\text{const } \text{name}) (\text{const } \text{name}' :: 'a) \neq \text{None}$
by (*rule match-list-comb-list-comb-some*) (*simp add: is-const-def*)+
hence $\text{name} = \text{name}'$
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 $\text{length } \text{ps} \neq \text{length } \text{vs}$
shows $\text{match } (\text{name } \$\$ \text{ps}) (\text{name}' \$\$ \text{vs}) = \text{None}$
proof (*rule ccontr*)
assume $\text{match } (\text{name } \$\$ \text{ps}) (\text{name}' \$\$ \text{vs}) \neq \text{None}$
then obtain *env* **where** $\text{match } (\text{name } \$\$ \text{ps}) (\text{name}' \$\$ \text{vs}) = \text{Some } \text{env}$
by *blast*
hence $\text{length } \text{ps} = \text{length } \text{vs}$
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-matches*:
assumes $\text{matches } \text{ps } \text{ts}_2 = \text{Some } \text{env}_2 \text{ list-all2 } P \text{ ts}_1 \text{ ts}_2$
obtains env_1 **where** $\text{matches } \text{ps } \text{ts}_1 = \text{Some } \text{env}_1 \text{ P-env } \text{env}_1 \text{ env}_2$
proof –
fix *name* — *dummy*

from *assms* **have** $\text{match } (\text{name } \$\$ \text{ps}) (\text{name } \$\$ \text{ts}_2) = \text{Some } \text{env}_2$
by *simp*
moreover have $P (\text{name } \$\$ \text{ts}_1) (\text{name } \$\$ \text{ts}_2)$
using *assms* **by** (*auto intro: P-const-const list-combI*)
ultimately obtain env_1 **where** $\text{match } (\text{name } \$\$ \text{ps}) (\text{name } \$\$ \text{ts}_1) = \text{Some}$
 $\text{env}_1 \text{ P-env } \text{env}_1 \text{ env}_2$
by (*metis related-match*)
hence $\text{matches } \text{ps } \text{ts}_1 = \text{Some } \text{env}_1$
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  $\Rightarrow$  bool where
linear (Free -)  $\longleftrightarrow$  True |
linear (Const -)  $\longleftrightarrow$  True |
linear (t1 $ t2)  $\longleftrightarrow$  linear t1  $\wedge$  linear t2  $\wedge$   $\neg$  is-free t1  $\wedge$  fdisjnt (frees t1) (frees t2) |
linear -  $\longleftrightarrow$  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  $\implies$  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  $\Rightarrow$  bool where
  linears []  $\longleftrightarrow$  True |
  linears (t # ts)  $\longleftrightarrow$  linear t  $\wedge$  fdisjnt (frees t) (freess ts)  $\wedge$  linears ts

```

lemma linears-butlastI[*intro*]: linears ts \implies 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)  $\sqsubseteq$  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 \cup freess ys)

```

proof (rule fdisjnt-union-right)
  show fdisjnt (frees z) (freess zs)
  using <linears (z # zs)> by simp

```

next

```

  have frees z  $\sqsubseteq$  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  $\implies$   $t \in \text{set } ts \implies$  linear t
by (induct ts) auto

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

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

lemma linears-strip-comb: linear t  $\implies$  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))  $\sqsubseteq$  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 $\text{linears } (xs @ ys)$ $\text{linear } x$ $\text{fdisjnt } (\text{frees } x)$ $(\text{freess } (xs @ ys))$
by *auto*
hence $\text{linears } xs$
using *Cons* **by** *simp*
moreover **have** $\text{fdisjnt } (\text{frees } x)$ $(\text{freess } xs)$
proof (*rule fdisjnt-subset-right*)
show $\text{freess } xs \subseteq \text{freess } (xs @ ys)$ **by** *simp*
qed *fact*
ultimately **show** $\text{linears } (x \# xs)$
using $\langle \text{linear } x \rangle$ **by** *auto*

have $\text{fdisjnt } (\text{freess } xs)$ $(\text{freess } ys)$
by (*rule Cons*) *fact*
moreover **have** $\text{fdisjnt } (\text{frees } x)$ $(\text{freess } ys)$
proof (*rule fdisjnt-subset-right*)
show $\text{freess } ys \subseteq \text{freess } (xs @ ys)$ **by** *simp*
qed *fact*
ultimately **show** $\text{fdisjnt } (\text{freess } (x \# xs))$ $(\text{freess } ys)$
unfolding *fdisjnt-alt-def*
by *auto*
qed (*auto simp: fdisjnt-alt-def*)

lemma *linear-list-comb*:
assumes $\text{linear } f$ $\text{linears } xs$ $\text{fdisjnt } (\text{frees } f)$ $(\text{freess } xs) \rightarrow \text{is-free } f$
shows $\text{linear } (\text{list-comb } f xs)$
using *assms*
proof (*induction xs arbitrary: f*)
case (*Cons x xs*)

hence $*$: $\text{fdisjnt } (\text{frees } f)$ $(\text{frees } x \cup \text{freess } xs)$
by *simp*

have $\text{linear } (\text{list-comb } (f \$ x) xs)$
proof (*rule Cons*)
have $\text{linear } x$
using *Cons* **by** *simp*
moreover **have** $\text{fdisjnt } (\text{frees } f)$ $(\text{frees } x)$
using $*$ **by** (*auto intro: fdisjnt-subset-right*)
ultimately **show** $\text{linear } (f \$ x)$
using *assms Cons* **by** *simp*

next
show $\text{linears } xs$
using *Cons* **by** *simp*

next
have $\text{fdisjnt } (\text{frees } f)$ $(\text{freess } xs)$
using $*$ **by** (*auto intro: fdisjnt-subset-right*)
moreover **have** $\text{fdisjnt } (\text{frees } x)$ $(\text{freess } xs)$
using *Cons* **by** *simp*
ultimately **show** $\text{fdisjnt } (\text{frees } (f \$ x))$ $(\text{freess } xs)$

```

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

```

corollary *linear-list-comb'*: $\text{linears } xs \implies \text{linear } (\text{name } \$\$ \text{ } xs)$
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*: $\text{linear } t \implies \text{wellformed}' n t$
by (*induct t*) *auto*

```

lemma pat-cases:
  obtains (free) s where t = Free s
    | (comb) name args where linears args t = name  $\$ \$$  args
    | (nonlinear)  $\neg$  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  $t = \text{name } \$\$$  [] 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

```

```

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

```

```

lemma linear-pat-induct[consumes 1, case-names free comb]:
  assumes linear t
  assumes  $\bigwedge s. P (\text{Free } s)$ 
  assumes  $\bigwedge \text{name args. linears } \text{args} \implies (\bigwedge \text{arg. arg} \in \text{set } \text{args} \implies P \text{ arg}) \implies P$ 
    (name  $\$ \$$  args)

```

```

  shows  $P t$ 
using wf-measure[of size]  $\langle \text{linear } t \rangle$ 
proof (induction t)
  case (less t)

  show ?case
    using  $\langle \text{linear } t \rangle$ 
    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  $\langle t = - \rangle$ 
    proof (rule assms)
      fix arg
      assume  $arg \in \text{set } args$ 

      then have  $(arg, t) \in \text{measure size}$ 
      unfolding  $\langle t = - \rangle$ 
      by (induction args) auto

      moreover have linear arg
      using  $\langle arg \in \text{set } args \rangle \langle \text{linears } args \rangle$ 
      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
     $None \Rightarrow (\forall \text{env. subst (convert-term } t) \text{ env} \neq u) \mid$ 
     $Some \text{ env} \Rightarrow \text{subst (convert-term } t) \text{ env} = u$ 
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 ++f 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 \implies *linear t* \implies *subst (convert-term t) env = u*
by (metis (mono-tags) *match-subst-correctness0 option.simps(5)*)

lemma *match-subst-none*:

match t u = None \implies *linear t* \implies *subst (convert-term t) env = u* \implies *False*
by (metis (mono-tags, lifting) *match-subst-correctness0 option.simps(4)*)

end

lemma *match-matches*: *match t u = Some env* \implies *linear t* \implies $t \lesssim u$
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 t1 u1  $\implies$  non-overlapping (t1 $
t2) (u1 $ u2)
unfolding overlapping-def matches-def by auto

lemma non-overlapping-appI2: non-overlapping t2 u2  $\implies$  non-overlapping (t1 $
t2) (u1 $ u2)
unfolding overlapping-def matches-def by auto

lemma non-overlapping-app-constI: non-overlapping (t1 $ t2) (Const name)
unfolding overlapping-def matches-def by simp

lemma non-overlapping-const-appI: non-overlapping (Const name) (t1 $ t2)
unfolding overlapping-def matches-def by simp

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

lemma match-overlapping:
  assumes linear t1 linear t2
  assumes match t1 u = Some env1 match t2 u = Some env2
  shows overlapping t1 t2
proof -
  define env1' where env1' = (fmmmap convert-term env1 :: (name, term) fmap)
  define env2' where env2' = (fmmmap convert-term env2 :: (name, term) fmap)
  from assms have match t1 (convert-term u :: term) = Some env1' match t2
(convert-term u :: term) = Some env2'
  unfolding env1'-def env2'-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 ( $\langle \Lambda_n \cdot. \rightarrow [0, 50] 50 \rangle$ ) |  
  Napp nterm nterm (infixl  $\langle \$_n \rangle 70$ )
```

```
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 (t1 $n t2) = frees-nterm t1 |∪| frees-nterm t2 |
  frees-nterm (Λn 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 (t1 $n t2) env = subst-nterm t1 env $n subst-nterm t2 env |
  subst-nterm (Λn x. t) env = (Λn 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 (t1 $n t2) = consts-nterm t1 |∪| consts-nterm t2 |
  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 = (Λn 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 *terms* and *nterms*

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

6.1 α -equivalence

```
inductive alpha-equiv :: (name, name) fmap  $\Rightarrow$  nterm  $\Rightarrow$  nterm  $\Rightarrow$  bool where
  const: alpha-equiv env (Nconst x) (Nconst x) |
  var1: x  $\notin$  fmdom env  $\Longrightarrow$  x  $\notin$  fmran env  $\Longrightarrow$  alpha-equiv env (Nvar x) (Nvar x) |
  var2: fmlookup env x = Some y  $\Longrightarrow$  alpha-equiv env (Nvar x) (Nvar y) |
  abs: alpha-equiv (fmupd x y env) n1 n2  $\Longrightarrow$  alpha-equiv env ( $\Lambda_n$  x. n1) ( $\Lambda_n$  y. n2) |
  app: alpha-equiv env n1 n2  $\Longrightarrow$  alpha-equiv env m1 m2  $\Longrightarrow$  alpha-equiv env (n1 $n m1) (n2 $n m2)
```

```
code-pred alpha-equiv .
```

```
abbreviation alpha-eq :: nterm  $\Rightarrow$  nterm  $\Rightarrow$  bool (infixl  $\langle \approx_\alpha \rangle$  50) where
  alpha-eq n1 n2  $\equiv$  alpha-equiv fmempty n1 n2
```

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

```

apply (rule alpha-equiv.app; rule Napp)
using Napp.premis 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  $\Rightarrow$  term  $\Rightarrow$  (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  $\Gamma$  (Bound n) = State-Monad.return (Nvar ( $\Gamma$  ! n)) |
term-to-nterm  $\Gamma$  ( $\Lambda$  t) = do {
  n  $\leftarrow$  fresh-create;
  e  $\leftarrow$  term-to-nterm (n #  $\Gamma$ ) t;
  State-Monad.return ( $\Lambda_n$  n. e)
} |
term-to-nterm  $\Gamma$  (t1 $ t2) = do {
  e1  $\leftarrow$  term-to-nterm  $\Gamma$  t1;
  e2  $\leftarrow$  term-to-nterm  $\Gamma$  t2;
  State-Monad.return (e1 $n e2)
}

```

```

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  $\Gamma$  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  $\Rightarrow$  nterm where
term-to-nterm' t = frun-fresh (term-to-nterm [] t) (frees t)

```

```

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

```

```

lemma term-to-nterm-vars0:
assumes wellformed' (length  $\Gamma$ ) t
shows frees (fst (run-state (term-to-nterm  $\Gamma$  t) s))  $\subseteq$  frees t  $\cup$  fset-of-list  $\Gamma$ 
using assms proof (induction  $\Gamma$  t arbitrary: s rule: term-to-nterm-induct)
case (bound  $\Gamma$  i)
hence  $\Gamma$  ! i  $\in$  fset-of-list  $\Gamma$ 
including fset.lifting by transfer auto

```

```

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

  from abs have frees (fst (run-state (term-to-nterm (?x #  $\Gamma$ ) t) ?x))  $\subseteq$  frees t
   $\cup$   $\{|?x\}$   $\cup$  fset-of-list  $\Gamma$ 
  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)  $\subseteq$  frees t
proof -
  let ? $\Gamma$  = []
  from assms have wellformed' (length ? $\Gamma$ ) t
  by simp
  hence frees (fst (run-state (term-to-nterm ? $\Gamma$  t) (fNext F)))  $\subseteq$  (frees t  $\cup$ 
  fset-of-list ? $\Gamma$ )
  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  $\implies$  wellformed t  $\implies$  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 ( $\lambda t'. \text{consts } t' = \text{consts } t$ ) (term-to-nterm
   $\Gamma$  t)
apply (rule pred-stateI)
apply (induction t arbitrary:  $\Gamma$ )
apply (auto split: prod.splits)
done

```

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

```

fun nterm-to-term :: name list  $\Rightarrow$  nterm  $\Rightarrow$  term where
  nterm-to-term  $\Gamma$  (Nconst name) = Const name |
  nterm-to-term  $\Gamma$  (Nvar name) = (case find-first name  $\Gamma$  of Some n  $\Rightarrow$  Bound n |
  None  $\Rightarrow$  Free name) |
  nterm-to-term  $\Gamma$  (t $n u) = nterm-to-term  $\Gamma$  t $ nterm-to-term  $\Gamma$  u |
  nterm-to-term  $\Gamma$  ( $\Lambda_n$  x. t) =  $\Lambda$  nterm-to-term (x #  $\Gamma$ ) t

```

```

lemma nterm-to-term:
  assumes no-abs t fdisjnt (fset-of-list  $\Gamma$ ) (frees t)
  shows nterm-to-term  $\Gamma$  t = convert-term t
using assms proof (induction arbitrary:  $\Gamma$ )
  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 t1 t2)
  then have nterm-to-term  $\Gamma$  t1 = convert-term t1 nterm-to-term  $\Gamma$  t2 = convert-term t2
    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'* \equiv *nterm-to-term* \square

lemma *nterm-to-term'*: *no-abs t \implies 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:  $\Gamma$ )
  case (Nvar name)
  show ?case
    proof (cases find-first name  $\Gamma$ )
      case None
      hence name  $\notin$  fset-of-list  $\Gamma$ 
      including fset.lifting
      by transfer (metis find-first-some option.distinct(1))
      with None show ?thesis
      by auto
    next
      case (Some u)
      hence name  $\in$  fset-of-list  $\Gamma$ 
      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  $\Gamma$ ) t fdisjnt (fset-of-list  $\Gamma$ ) (frees t) distinct  $\Gamma$ 
  assumes fBall (frees t  $\cup$  fset-of-list  $\Gamma$ ) ( $\lambda x. x \leq s$ )
  shows nterm-to-term  $\Gamma$  (fst (run-state (term-to-nterm  $\Gamma$  t) s)) = t
using assms proof (induction  $\Gamma$  t arbitrary: s rule: term-to-nterm-induct)
  case (free  $\Gamma$  name)
  hence fdisjnt (fset-of-list  $\Gamma$ ) {name}
    by simp
  hence name  $\notin$  set  $\Gamma$ 
    including fset.lifting by transfer' (simp add: disjnt-def)
  hence find-first name  $\Gamma$  = None
    by (rule find-first-none)
  thus ?case
    by (simp add: State-Monad.return-def)
next
  case (bound  $\Gamma$  i)
  thus ?case
    by (simp add: State-Monad.return-def find-first-some-index)
next
  case (app  $\Gamma$  t1 t2)
  have fdisjnt (fset-of-list  $\Gamma$ ) (frees t1)
    apply (rule fdisjnt-subset-right[where N = frees t1  $\cup$  frees t2])
    using app by auto
  have fdisjnt (fset-of-list  $\Gamma$ ) (frees t2)
    apply (rule fdisjnt-subset-right[where N = frees t1  $\cup$  frees t2])
    using app by auto

  have s: s  $\leq$  snd (run-state (term-to-nterm  $\Gamma$  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  $\Gamma$  t)

have next s  $\notin$  frees t  $\cup$  fset-of-list  $\Gamma$ 
  using abs(5) next-ge-fall by auto

have nterm-to-term (next s  $\#$   $\Gamma$ ) (fst (run-state (term-to-nterm (next s  $\#$   $\Gamma$ ) t)
(next s))) = t
  proof (subst abs)
    show wellformed' (length (next s  $\#$   $\Gamma$ )) t
      using abs by auto
    show fdisjnt (fset-of-list (next s  $\#$   $\Gamma$ )) (frees t)
      apply simp
      apply (rule fdisjnt-insert)
      using  $\langle$ next s  $\notin$  frees t  $\cup$  fset-of-list  $\Gamma$  $\rangle$  abs by auto
    show distinct (next s  $\#$   $\Gamma$ )
      apply simp
      apply rule
    using  $\langle$ next s  $\notin$  frees t  $\cup$  fset-of-list  $\Gamma$  $\rangle$  apply (simp add: fset-of-list-elem)
    apply fact
    done
  have fBall (frees t  $\cup$  fset-of-list  $\Gamma$ ) ( $\lambda x. x \leq$  next s)
  proof (rule fBall-pred-weaken)
    show fBall (frees t  $\cup$  fset-of-list  $\Gamma$ ) ( $\lambda x. x \leq$  s)
      using abs(5) by simp
  next
  show  $\bigwedge x. x \in$  frees t  $\cup$  fset-of-list  $\Gamma \implies x \leq$  s  $\implies x \leq$  next s
    by (metis Fresh-Class.next-ge dual-order.strict-trans less-eq-name-def)
  qed
  thus fBall (frees t  $\cup$  fset-of-list (next s  $\#$   $\Gamma$ )) ( $\lambda x. x \leq$  next s)
    by simp
  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  $\subseteq$  S
  shows nterm-to-term' (frun-fresh (term-to-nterm [] t) (S  $\cup$  Q)) = t
  proof (rule term-to-nterm-nterm-to-term0)
    show wellformed' (length []) t
      using assms by simp
  next

```

show $fdisjnt$ ($fset\text{-of-list []$) ($frees\ t$)
unfolding $fdisjnt\text{-alt-def}$ **by** $simp$
next
have $fBall$ ($S \mid\cup\mid Q$) ($\lambda x. x < fresh.fNext\ next\ default\ (S \mid\cup\mid Q)$)
by ($metis\ fNext\ geq\ not\ member\ fresh\ fNext\ def\ le\ less\ linear\ fBallI$)
hence $fBall$ ($S \mid\cup\mid Q$) ($\lambda x. x \leq fresh.fNext\ next\ default\ (S \mid\cup\mid Q)$)
by ($meson\ fBall\ pred\ weaken\ less\ eq\ name\ def$)
thus $fBall$ ($frees\ t \mid\cup\mid fset\text{-of-list []$) ($\lambda x. x \leq fresh.fNext\ next\ default\ (S \mid\cup\mid Q)$)
using $\langle frees\ t \mid\subseteq\mid S \rangle$
by $auto$
qed $simp$

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

lemma $nterm\text{-to-term-eq}$:
assumes $frees\ u \mid\subseteq\mid fset\text{-of-list (common-prefix } \Gamma\ \Gamma')$
shows $nterm\text{-to-term } \Gamma\ u = nterm\text{-to-term } \Gamma'\ u$
using $assms$
proof ($induction\ u\ arbitrary: \Gamma\ \Gamma'$)
case ($Nvar\ name$)
hence $name \in set (common-prefix\ \Gamma\ \Gamma')$
unfolding $frees\ nterm.\ simps$
including $fset.\ lifting$
by $transfer'\ simp$
thus $?case$
by ($auto\ simp: common\ prefix\ find$)
next
case ($Nabs\ x\ t$)
hence $*: frees\ t - \{|x|\} \mid\subseteq\mid fset\text{-of-list (common-prefix } \Gamma\ \Gamma')$
by $simp$

show $?case$
apply $simp$
apply ($rule\ Nabs$)
using $*\ Nabs$ **by** $auto$
qed $auto$

corollary $nterm\text{-to-term-eq-closed}$: $closed\ t \implies nterm\text{-to-term } \Gamma\ t = nterm\text{-to-term } \Gamma'\ t$
by ($rule\ nterm\text{-to-term-eq}$) ($auto\ simp: closed\ except\ def$)

lemma $nterm\text{-to-term-wellformed}$: $wellformed' (length\ \Gamma) (nterm\text{-to-term } \Gamma\ t)$
proof ($induction\ t\ arbitrary: \Gamma$)
case ($Nabs\ x\ t$)
hence $wellformed' (Suc (length\ \Gamma)) (nterm\text{-to-term } (x \# \Gamma)\ t)$

```

    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  $\implies$  wellformed (nterm-to-term
 $\Gamma$  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  $\subseteq$  fset-of-list  $\Gamma$  length  $\Gamma =$  length  $\Gamma'$ 
  shows alpha-equiv (fmap-of-list (zip  $\Gamma$   $\Gamma'$ )) t (fst (run-state (term-to-nterm  $\Gamma'$ 
  (nterm-to-term  $\Gamma$  t)) s))
using assms proof (induction  $\Gamma$  t arbitrary: s  $\Gamma'$  rule:nterm-to-term.induct)
  case ( $\_4$   $\Gamma$  x t)
  show ?case
    apply (simp add: split-beta)
    apply (rule alpha-equiv.abs)
    using  $\_4$ .IH[where  $\Gamma' =$  next s #  $\Gamma'$ ]  $\_4$ .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  $\implies$   $t \approx_\alpha$  term-to-nterm' (nterm-to-term'
  t)
unfolding term-to-nterm'-def closed-except-def
apply (rule nterm-to-term-term-to-nterm'[where  $\Gamma = []$  and  $\Gamma' = []$ , simplified])
by auto

context begin

private lemma term-to-nterm-alpha-equiv0:
  length  $\Gamma 1 =$  length  $\Gamma 2 \implies$  distinct  $\Gamma 1 \implies$  distinct  $\Gamma 2 \implies$  wellformed' (length
   $\Gamma 1$ ) t1  $\implies$ 
  fresh-fin (frees t1  $\cup$  fset-of-list  $\Gamma 1$ ) s1  $\implies$  fdisjnt (fset-of-list  $\Gamma 1$ ) (frees t1)
 $\implies$ 
  fresh-fin (frees t1  $\cup$  fset-of-list  $\Gamma 2$ ) s2  $\implies$  fdisjnt (fset-of-list  $\Gamma 2$ ) (frees t1)
 $\implies$ 
  alpha-equiv (fmap-of-list (zip  $\Gamma 1$   $\Gamma 2$ )) (fst (run-state (term-to-nterm  $\Gamma 1$  t1) s1))
  (fst (run-state (term-to-nterm  $\Gamma 2$  t1) s2))
proof (induction  $\Gamma 1$  t1 arbitrary:  $\Gamma 2$  s1 s2 rule:term-to-nterm-induct)
  case (free  $\Gamma 1$  name)
  then have name  $\notin$  fmdom (fmap-of-list (zip  $\Gamma 1$   $\Gamma 2$ ))
    unfolding fdisjnt-alt-def
    by force
  moreover have name  $\notin$  fmran (fmap-of-list (zip  $\Gamma 1$   $\Gamma 2$ ))
    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 mapfst-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  $\Gamma$  t)
have *: next s1 > s1 next s2 > s2
  using next-ge by force+
from abs.premis(5,7) have next s1  $\notin$  set  $\Gamma$  next s2  $\notin$  set  $\Gamma$ 2
  unfolding fBall-funion
  by (metis fset-of-list-elem next-ge-fall)+
moreover have fresh-fin (frees t  $\cup$  fset-of-list  $\Gamma$ ) (next s1)
  fresh-fin (frees t  $\cup$  fset-of-list  $\Gamma$ 2) (next s2)
  using * abs
  by (smt dual-order.trans fBall-pred-weaken frees-term.simps(3) less-imp-le)+
moreover have fdisjnt (finsert (next s1) (fset-of-list  $\Gamma$ )) (frees t)
  fdisjnt (finsert (next s2) (fset-of-list  $\Gamma$ 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)+
moreover have wellformed' (Suc (length  $\Gamma$ 2)) t
  using wellformed'.simps abs
  by (metis Suc-eq-plus1)
ultimately show ?case
  using abs.premis(1,2,3)
  by (auto simp: split-beta create-alt-def
    intro: alpha-equiv.abs abs.IH[of - next s2 #  $\Gamma$ 2, simplified])
next
case (app  $\Gamma$ 1 t1 t2)
hence wellformed' (length  $\Gamma$ 1) t1 wellformed' (length  $\Gamma$ 1) t2
and fresh-fin (frees t1  $\cup$  fset-of-list  $\Gamma$ 1) s1 fresh-fin (frees t1  $\cup$  fset-of-list  $\Gamma$ 2)
s2
and fdisjnt (fset-of-list  $\Gamma$ 1) (frees t1) fdisjnt (fset-of-list  $\Gamma$ 2) (frees t1)
and fdisjnt (fset-of-list  $\Gamma$ 1) (frees t2) fdisjnt (fset-of-list  $\Gamma$ 2) (frees t2)
  using app
  unfolding fdisjnt-alt-def
  by (auto simp: inf-sup-distrib1)
have s1  $\leq$  snd (run-state (term-to-nterm  $\Gamma$ 1 t1) s1) s2  $\leq$  snd (run-state (term-to-nterm
 $\Gamma$ 2 t1) s2)
  using term-to-nterm-mono
  by (simp add: state-io-rel-def)+

```

```

hence fresh-fin (frees  $t_2 \mid \cup \mid$  fset-of-list  $\Gamma 1$ ) (snd (run-state (term-to-nterm  $\Gamma 1$ 
 $t_1$ )  $s1$ ))
  using  $\langle$ fresh-fin (frees ( $t_1 \ \$ \ t_2$ )  $\mid \cup \mid$  fset-of-list  $\Gamma 1$ )  $s1$  $\rangle$ 
  by force

have fresh-fin (frees  $t_2 \mid \cup \mid$  fset-of-list  $\Gamma 2$ ) (snd (run-state (term-to-nterm  $\Gamma 2$   $t_1$ 
 $s2$ ))
  apply rule
  using app frees-term.simps  $\langle s2 \leq \rightarrow$  dual-order.trans
  by (metis funion-iff)

show ?case
  apply (auto simp: split-beta create-alt-def)
  apply (rule alpha-equiv.app)
  subgoal
    using app.IH
    using  $\langle$ fBall (frees  $t_1 \mid \cup \mid$  fset-of-list  $\Gamma 1$ ) ( $\lambda y. y \leq s1$ ) $\rangle$ 
       $\langle$ fBall (frees  $t_1 \mid \cup \mid$  fset-of-list  $\Gamma 2$ ) ( $\lambda y. y \leq s2$ ) $\rangle$ 
       $\langle$ fdisjnt (fset-of-list  $\Gamma 1$ ) (frees  $t_1$ ) $\rangle$ 
       $\langle$ fdisjnt (fset-of-list  $\Gamma 2$ ) (frees  $t_1$ ) $\rangle$   $\langle$ wellformed' (length  $\Gamma 1$ )  $t_1$  $\rangle$ 
      app.prems(1) app.prems(2) app.prems(3) by blast
    subgoal
      using app.IH
      using  $\langle$ fBall (frees  $t_2 \mid \cup \mid$  fset-of-list  $\Gamma 1$ ) ( $\lambda y. y \leq$  snd (run-state (term-to-nterm
 $\Gamma 1$   $t_1$ )  $s1$ )) $\rangle$ 
         $\langle$ fBall (frees  $t_2 \mid \cup \mid$  fset-of-list  $\Gamma 2$ ) ( $\lambda y. y \leq$  snd (run-state (term-to-nterm
 $\Gamma 2$   $t_1$ )  $s2$ )) $\rangle$ 
         $\langle$ fdisjnt (fset-of-list  $\Gamma 1$ ) (frees  $t_2$ ) $\rangle$ 
         $\langle$ fdisjnt (fset-of-list  $\Gamma 2$ ) (frees  $t_2$ ) $\rangle$ 
         $\langle$ wellformed' (length  $\Gamma 1$ )  $t_2$  $\rangle$ 
        app.prems(1) app.prems(2) app.prems(3) by blast
      done
    qed (force intro: alpha-equiv.intros simp: fmllookup-of-list in-set-zip)+

lemma term-to-nterm-alpha-equiv:
  assumes length  $\Gamma 1 =$  length  $\Gamma 2$  distinct  $\Gamma 1$  distinct  $\Gamma 2$  closed  $t$ 
  assumes wellformed' (length  $\Gamma 1$ )  $t$ 
  assumes fresh-fin (fset-of-list  $\Gamma 1$ )  $s1$  fresh-fin (fset-of-list  $\Gamma 2$ )  $s2$ 
  shows alpha-equiv (fmap-of-list (zip  $\Gamma 1$   $\Gamma 2$ )) (fst (run-state (term-to-nterm  $\Gamma 1$ 
 $t$ )  $s1$ )) (fst (run-state (term-to-nterm  $\Gamma 2$   $t$ )  $s2$ ))
  — An instantiated version of this lemma with  $\Gamma 1 = []$  and  $\Gamma 2 = []$  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 ( $\lambda t n. t =$  nterm-to-term
 $\Gamma n$ ) for  $\Gamma$ 

```

```

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 u1 u2 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  $\Gamma$  env nenv closed-env nenv
  shows nrelated.P-env  $\Gamma'$  env nenv
proof
  fix x
  from assms have rel-option ( $\lambda t n. t = \text{nterm-to-term } \Gamma n$ ) (fmlookup env x)
  (fmlookup nenv x)
  by auto
  thus rel-option ( $\lambda t n. t = \text{nterm-to-term } \Gamma' n$ ) (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  $\Gamma$  env nenv closed-env nenv fdisjnt (fset-of-list  $\Gamma$ ) (fmdom nenv)
  shows subst (nterm-to-term  $\Gamma$  t) env = nterm-to-term  $\Gamma$  (subst t nenv)
using assms
proof (induction t arbitrary:  $\Gamma$  env nenv)
  case (Nvar name)
  thus ?case
    proof (cases rule: fmrel-cases[where  $x = name$ ])
      case (some t1 t2)
      with Nvar have name  $\notin$  fset-of-list  $\Gamma$ 
      unfolding fdisjnt-alt-def by (auto dest: fmdomI)
      hence find-first name  $\Gamma = None$ 
      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  $y \in \text{set } (\text{take } n \ \Gamma)$   $n \leq \text{length } \Gamma$ 
  shows  $\text{nterm-to-term } \Gamma \ t = \text{incr-bounds } (- \ 1) \ (\text{Suc } n) \ (\text{nterm-to-term } (\text{insert-nth } n \ y \ \Gamma) \ t)$ 
  using assms
  proof (induction  $t$  arbitrary:  $n \ \Gamma$ )
    case (Nvar  $\text{name}$ )
      show ?case
        proof (cases  $y = \text{name}$ )
          case True
            with Nvar obtain  $i$  where  $\text{find-first name } \Gamma = \text{Some } i \ i < n$ 
              by (auto elim: find-first-some-strong)

            hence  $\text{find-first name } (\text{take } n \ \Gamma) = \text{Some } i$ 
              by (rule find-first-prefix)

            show ?thesis
              apply simp
              apply (subst  $\langle \text{find-first name } \Gamma = \text{Some } i \rangle$ )
              apply simp
              apply (subst find-first-append)
              apply (subst  $\langle \text{find-first name } (\text{take } n \ \Gamma) = \text{Some } i \rangle$ )
              apply simp
              using  $\langle i < n \rangle$  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  $\text{find-first name } \Gamma$ ) auto
        qed
      qed
    next
      case (Nabs  $x \ t$ )
        show ?case
          apply simp
          apply (subst  $\text{Nabs}(1)[\text{where } n = \text{Suc } n]$ )
          using Nabs by auto
      qed
  qed auto

lemma nterm-to-term-bounds-dupl:
  assumes  $i < \text{length } \Gamma$   $j < \text{length } \Gamma$   $i < j$ 
  assumes  $\Gamma \ ! \ i = \Gamma \ ! \ j$ 
  shows  $j \notin \text{bounds } (\text{nterm-to-term } \Gamma \ t)$ 
  using assms
  proof (induction  $t$  arbitrary:  $\Gamma \ i \ j$ )

```



```

case (Nvar name)
show ?case
  proof (cases find-first name  $\Gamma$ )
    case (Some k)
      show ?thesis
        proof
          assume  $j \in |\text{bounds}(\text{nterm-to-term } \Gamma \text{ (Nvar name)})|$ 
          with Some have find-first name  $\Gamma = \text{Some } j$ 
            by simp

          moreover have find-first name  $\Gamma \neq \text{Some } j$ 
            proof (rule find-first-later)
              show  $i < \text{length } \Gamma \ j < \text{length } \Gamma \ i < j$ 
                by fact+
              next
                show  $\Gamma ! j = \text{name}$ 
                  by (rule find-first-correct) fact
                thus  $\Gamma ! i = \text{name}$ 
                  using Nvar by simp
              qed

            ultimately show False
              by blast
            qed
          qed simp
        next
          case (Nabs x t)
            show ?case
              proof
                assume  $j \in |\text{bounds}(\text{nterm-to-term } \Gamma \ (\Lambda_n \ x. \ t))|$ 
                then obtain  $j'$  where  $j' \in |\text{bounds}(\text{nterm-to-term } (x \# \Gamma) \ t) \ j' > 0 \ j = j'$ 
                - 1
                  by auto
                hence  $\text{Suc } j \in |\text{bounds}(\text{nterm-to-term } (x \# \Gamma) \ t)|$ 
                  by simp

                moreover have  $\text{Suc } j \notin |\text{bounds}(\text{nterm-to-term } (x \# \Gamma) \ t)|$ 
                  proof (rule Nabs)
                    show  $\text{Suc } i < \text{length } (x \# \Gamma) \ \text{Suc } j < \text{length } (x \# \Gamma) \ \text{Suc } i < \text{Suc } j \ (x \# \Gamma) ! \text{Suc } i = (x \# \Gamma) ! \text{Suc } j$ 
                      using Nabs by simp+
                    qed
                  qed

                ultimately show False
                  by blast
                qed
              qed auto
            qed
          fun subst-single :: nterm  $\Rightarrow$  name  $\Rightarrow$  nterm  $\Rightarrow$  nterm where

```

$\text{subst-single } (Nvar\ s)\ s'\ t' = (\text{if } s = s' \text{ then } t' \text{ else } Nvar\ s) \mid$
 $\text{subst-single } (t_1\ \$n\ t_2)\ s'\ t' = \text{subst-single } t_1\ s'\ t'\ \$n\ \text{subst-single } t_2\ s'\ t' \mid$
 $\text{subst-single } (\Lambda_n\ x.\ t)\ s'\ t' = (\Lambda_n\ x.\ (\text{if } x = s' \text{ then } t \text{ else } \text{subst-single } t\ s'\ t')) \mid$
 $\text{subst-single } t\ -\ - = t$

lemma *subst-single-eq*: $\text{subst-single } t\ s\ t' = \text{subst } t\ (\text{fmap-of-list } [(s, t')])$

proof (*induction t*)
case (*Nabs x t*)
then show *?case*
by (*cases x = s*) (*simp add: fmapfilter-alt-defs*)+
qed *auto*

lemma *nterm-to-term-subst-replace-bound*:

assumes *closed u' n ≤ length Γ x ∉ set (take n Γ)*
shows $\text{nterm-to-term } \Gamma\ (\text{subst-single } u\ x\ u') = \text{replace-bound } n\ (\text{nterm-to-term } (\text{insert-nth } n\ x\ \Gamma)\ u)\ (\text{nterm-to-term } \Gamma\ u')$

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 $\text{nterm-to-term } (y\ \#\ \Gamma)\ t = \text{replace-bound } (Suc\ n)\ (\text{nterm-to-term } (y\ \#\ \text{insert-nth } n\ y\ \Gamma)\ t)\ (\text{nterm-to-term } \Gamma\ u')$

proof (*subst replace-bound-eq*)

show $Suc\ n\ |\notin| \text{ bounds } (\text{nterm-to-term } (y\ \#\ \text{insert-nth } n\ y\ \Gamma)\ t)$

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 #  $\Gamma$ ) t = incr-bounds (- 1) (Suc n + 1)
    (nterm-to-term (y # insert-nth n y  $\Gamma$ ) t)
    apply (subst nterm-to-term-insert-dupl[where  $\Gamma = y \# \Gamma$  and  $y = y$ 
and  $n = \text{Suc } n$ ])
    using Nabs by auto
    qed
    with True show ?thesis
    by auto
  next
  case False
  have nterm-to-term (y #  $\Gamma$ ) (subst-single t x u') = replace-bound (Suc n)
  (nterm-to-term (y # insert-nth n x  $\Gamma$ ) t) (nterm-to-term  $\Gamma$  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  $\Gamma$  (subst u (fmap-of-list [(x, u')])) = nterm-to-term (x #
 $\Gamma$ ) u [nterm-to-term  $\Gamma$  u'] $_{\beta}$ 
  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  $\Rightarrow$  'a
  assumes bij: bij of-name
begin
```

```
definition to-name :: 'a  $\Rightarrow$  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 name1 = of-name name2  $\implies$  name1 = name2
```

```

using bij by (metis to-of-name)

end

instantiation name :: is-name begin

definition of-name-name :: name  $\Rightarrow$  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  $\Rightarrow$  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  $\Rightarrow$  Some (t, u) | -  $\Rightarrow$  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  $\Rightarrow$  Some (to-name a) | -  $\Rightarrow$  None)

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

definition unfree-trm where
unfree-trm t = (case t of Var a  $\Rightarrow$  Some (to-name a) | -  $\Rightarrow$  None)

primrec consts-trm :: 'a trm  $\Rightarrow$  name fset where
consts-trm (Var -) =  $\{\{\}\}$  |
consts-trm (trm.Const c) =  $\{\{ \text{to-name } c \}\}$  |
consts-trm (M · N) = consts-trm M  $\cup$  consts-trm N

context
  includes fset.lifting
begin

lift-definition frees-trm :: 'a trm  $\Rightarrow$  name fset is  $\lambda t.$  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 fnlookup 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: fmllookup-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  $\Rightarrow$  Some (t, u) | -  $\Rightarrow$  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)  $\Rightarrow$  Some (to-name a) | -  $\Rightarrow$  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)  $\Rightarrow$  Some (to-name a) | -  $\Rightarrow$  None)
```

```
context
```

```
  includes fset.lifting
```

```
begin
```

```
lift-definition frees-tm :: ('a, 'b) tm  $\Rightarrow$  name fset is  $\lambda t.$  to-name 'vars t
```



```

by auto

lift-definition consts-tm :: ('a, 'b) tm ⇒ name fset is λt. 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 un-
    free-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.