

An Algebra for Higher-Order Terms

Lars Hupel

December 14, 2021

Abstract

In this formalization, I introduce a higher-order term algebra, generalizing the notions of free variables, matching, and substitution. The need arose from the work on a verified compiler from Isabelle to CakeML [3]. Terms can be thought of as consisting of a *generic* (free variables, constants, application) and a *specific* part. As example applications, this entry provides instantiations for de-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	4
2.1	Fresh monad operations as class operations	6
3	Terms	8
3.1	A simple term type, modelled after Pure's <i>term</i> type	8
3.2	A type class describing terms	8
3.3	Related work	27
3.4	Instantiation of class <i>term</i> for type <i>term</i>	28
4	Wellformedness of patterns	33
5	Terms with explicit bound variable names	37
6	Converting between <i>terms</i> and <i>nterms</i>	39
6.1	α -equivalence	39
6.2	From <i>Term-Class.term</i> to <i>nterm</i>	40
6.3	From <i>nterm</i> to <i>Term-Class.term</i>	41
6.4	Correctness	41
7	Instantiation for <i>HOL-ex.Unification</i> from session <i>HOL-ex</i>	45
8	Instantiation for λ-free terms according to Blanchette	48

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 <proof>
```

end

instance *name* :: *order*
<proof>

instance *name* :: *linorder*
<proof>

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*
<proof>

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*
<proof>

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>

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
<proof>
```

```
lemma update-next-mono[simp, intro]: mono-state update-next
<proof>
```

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

lemma $create\text{-alt-def}[code]: create = \text{State} (\lambda a. (\text{next } a, \text{next } a))$
 $\langle proof \rangle$

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

lemma $next\text{-ge-all}: \text{finite } S \Longrightarrow fresh\text{-in } S \ s \Longrightarrow next \ s \notin S$
 $\langle proof \rangle$

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

lemma $Next\text{-ge-max}: \text{finite } S \Longrightarrow S \neq \{\} \Longrightarrow Next \ S > \text{Max } S$
 $\langle proof \rangle$

lemma $Next\text{-not-member-subset}: \text{finite } S' \Longrightarrow S \subseteq S' \Longrightarrow Next \ S' \notin S$
 $\langle proof \rangle$

lemma $Next\text{-not-member}: \text{finite } S \Longrightarrow Next \ S \notin S$
 $\langle proof \rangle$

lemma $Next\text{-geq-not-member}: \text{finite } S \Longrightarrow s \geq Next \ S \Longrightarrow s \notin S$
 $\langle proof \rangle$

lemma $next\text{-not-member}: \text{finite } S \Longrightarrow s \geq Next \ S \Longrightarrow next \ s \notin S$
 $\langle proof \rangle$

lemma $create\text{-mono}[simp, \text{intro}]: \text{mono-state } create$
 $\langle proof \rangle$

lemma $create\text{-strict-mono}[simp, \text{intro}]: \text{strict-mono-state } create$
 $\langle proof \rangle$

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

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

context includes $fset.lifting \text{ begin}$

lemma $next\text{-ge-fall}: fresh\text{-fin } S \ s \Longrightarrow next \ s \notin S$
 $\langle proof \rangle$

lift-definition $fNext :: 'a \text{ fset} \Rightarrow 'a \text{ is } Next \langle proof \rangle$

lemma *fNext-ge-max*: $S \neq \{\}\implies fNext\ S > fMax\ S$
<proof>

lemma *next-not-fmember*: $s \geq fNext\ S \implies next\ s \notin S$
<proof>

lemma *fNext-geq-not-member*: $s \geq fNext\ S \implies s \notin S$
<proof>

lemma *fNext-not-member*: $fNext\ S \notin S$
<proof>

lemma *fNext-not-member-subset*: $S \subseteq S' \implies fNext\ S' \notin S$
<proof>

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

end

end

end

2.1 Fresh monad operations as class operations

theory *Fresh-Class*

imports

Fresh-Monad

Name

begin

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

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

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

global-interpretation *Fresh-Monad.fresh* *next* *default*

defines *fresh-create* = *create*

and *fresh-Next* = *Next*

and *fresh-fNext* = *fNext*

and *fresh-frun* = *frun-fresh*


```

and fresh-run = run-fresh
⟨proof⟩

lemma [code]: fresh-frun m S = fst (run-state m (fresh-fNext S))
⟨proof⟩

lemma [code]: fresh-run m S = fst (run-state m (fresh-Next S))
⟨proof⟩

instantiation nat :: fresh begin

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

definition next-nat where
next-nat = Suc

instance
⟨proof⟩

end

instantiation char :: default
begin

definition default-char :: char where
default-char = CHR "-"

instance ⟨proof⟩

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⟩

end

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

end

```

Chapter 3

Terms

```
theory Term-Class
imports
  Datatype-Order-Generator.Order-Generator
  Name
  Term-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 ( $\Lambda$  - [71] 71) |
  Bound nat |
  App term term (infixl $ 70)
```

```
derive linorder term
```

3.2 A type class describing terms

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

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

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

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

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 $(Free\ x) = \{ \mid x \mid \}$

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 $(Const\ x) = \{ \mid x \mid \}$

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

<proof>

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* $\mid\cup\mid$ *freess* *xs*
(*proof*)

lemma *freess-single*: *freess* [*x*] = *frees* *x*
(*proof*)

lemma *freess-empty[simp]*: *freess* [] = {}
(*proof*)

lemma *freess-app[simp]*: *freess* (*xs* @ *ys*) = *freess* *xs* $\mid\cup\mid$ *freess* *ys*
(*proof*)

lemma *freess-subset*: set *xs* \subseteq set *ys* \Longrightarrow *freess* *xs* $\mid\subseteq\mid$ *freess* *ys*
(*proof*)

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* $\mid\subseteq\mid$ *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)
(*proof*)

lemma *app-size2*: size *u*₂ < size (*app* *u*₁ *u*₂)
(*proof*)

lemma *unx-some-lemmas*:
unapp *u* = *Some* *x* \Longrightarrow *unconst* *u* = *None*
unapp *u* = *Some* *x* \Longrightarrow *unfree* *u* = *None*
unconst *u* = *Some* *y* \Longrightarrow *unapp* *u* = *None*
unconst *u* = *Some* *y* \Longrightarrow *unfree* *u* = *None*
unfree *u* = *Some* *z* \Longrightarrow *unconst* *u* = *None*

$unfree\ u = Some\ z \implies unapp\ u = None$
(proof)

lemma *unx-none-simps*[simp]:
 $unapp\ (const\ name) = None$
 $unapp\ (free\ name) = None$
 $unconst\ (app\ t\ u) = None$
 $unconst\ (free\ name) = None$
 $unfree\ (const\ name) = None$
 $unfree\ (app\ t\ u) = None$
(proof)

lemma *term-cases*:
 obtains $(free)\ name\ where\ t = free\ name$
 | $(const)\ name\ where\ t = const\ name$
 | $(app)\ u_1\ u_2\ where\ t = app\ u_1\ u_2$
 | $(other)\ unfree\ t = None\ unapp\ t = None\ unconst\ t = None$
(proof)

definition *is-const* **where**
 $is-const\ t \longleftrightarrow (unconst\ t \neq None)$

definition *const-name* **where**
 $const-name\ t = (case\ unconst\ t\ of\ Some\ name \Rightarrow name)$

lemma *is-const-simps*[simp]:
 $is-const\ (const\ name)$
 $\neg\ is-const\ (app\ t\ u)$
 $\neg\ is-const\ (free\ name)$
(proof)

lemma *const-name-simps*[simp]:
 $const-name\ (const\ name) = name$
 $is-const\ t \implies const\ (const-name\ t) = t$
(proof)

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]:
 $is-free\ (free\ name)$
 $\neg\ is-free\ (const\ name)$
 $\neg\ is-free\ (app\ t\ u)$
(proof)

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

$free\text{-}name\ (free\ name) = name$
 $is\text{-}free\ t \implies free\ (free\text{-}name\ t) = t$
 <proof>

definition *is-app* **where**
 $is\text{-}app\ t \longleftrightarrow (unapp\ t \neq None)$

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]:
 $\neg is\text{-}app\ (const\ name)$
 $\neg is\text{-}app\ (free\ name)$
 $is\text{-}app\ (app\ t\ u)$
 <proof>

lemma *left-right-simps*[simp]:
 $left\ (app\ l\ r) = l$
 $right\ (app\ l\ r) = r$
 $is\text{-}app\ t \implies app\ (left\ t)\ (right\ t) = t$
 <proof>

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

lemma *closed-except-const*[simp]: $closed\text{-}except\ (const\ name)\ S$
 <proof>

abbreviation *closed-env* :: (name, 'a) fmap \Rightarrow bool **where**
 $closed\text{-}env \equiv fmpred\ (\lambda\cdot\ closed)$

lemma *closed-except-self*: $closed\text{-}except\ t\ (frees\ t)$
 <proof>

end

class *term* = *pre-term* + *size* +
fixes

$abs\text{-}pred :: ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$

assumes

$raw\text{-}induct[case\text{-}names\ const\ free\ app\ abs]:$

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

assumes
raw-subst-id: *abs-pred* ($\lambda t. \forall env. id\text{-env } env \longrightarrow subst\ t\ env = t$) **and**
raw-subst-drop: *abs-pred* ($\lambda t. x \notin frees\ t \longrightarrow (\forall env. subst\ t\ (fmdrop\ x\ env) = subst\ t\ env)$) **and**
raw-subst-indep: *abs-pred* ($\lambda t. \forall env_1\ env_2. closed\text{-env } env_2 \longrightarrow fdisjnt\ (fmdom\ env_1)\ (fmdom\ env_2) \longrightarrow subst\ t\ (env_1\ ++_f\ env_2) = subst\ (subst\ t\ env_2)\ env_1$) **and**
raw-subst-frees: *abs-pred* ($\lambda t. \forall env. closed\text{-env } env \longrightarrow frees\ (subst\ t\ env) = frees\ t\ \mid\text{-}\ fmdom\ env$) **and**
raw-subst-consts': *abs-pred* ($\lambda a. \forall x. consts\ (subst\ a\ x) = consts\ a\ \mid\cup\ \text{ffUnion}\ (consts\ \mid^{\dagger}\ fmimage\ x\ (frees\ a))$) **and**
abs-pred-trivI: $P\ t \implies abs\text{-pred } P\ t$
begin

lemma *subst-id*: $id\text{-env } env \implies subst\ t\ env = t$
 $\langle proof \rangle$

lemma *subst-drop*: $x \notin frees\ t \implies subst\ t\ (fmdrop\ x\ env) = subst\ t\ env$
 $\langle proof \rangle$

lemma *subst-frees*: $fmpred\ (\lambda\cdot. closed)\ env \implies frees\ (subst\ t\ env) = frees\ t\ \mid\text{-}\ fmdom\ env$
 $\langle proof \rangle$

lemma *subst-consts'*: $consts\ (subst\ t\ env) = consts\ t\ \mid\cup\ \text{ffUnion}\ (consts\ \mid^{\dagger}\ fmimage\ env\ (frees\ t))$
 $\langle proof \rangle$

fun *match* :: $term \Rightarrow 'a \Rightarrow (name, 'a)\ fmap\ option$ **where**
 $match\ (t_1\ \$\ t_2)\ u = do\ \{$
 $\quad (u_1, u_2) \leftarrow unapp\ u;$
 $\quad env_1 \leftarrow match\ t_1\ u_1;$
 $\quad env_2 \leftarrow match\ t_2\ u_2;$
 $\quad Some\ (env_1\ ++_f\ env_2)$
 $\}$ |
 $match\ (Const\ name)\ u =$
 $\quad (case\ unconst\ u\ of$
 $\quad\quad None \Rightarrow None$
 $\quad\quad | Some\ name' \Rightarrow if\ name = name'\ then\ Some\ fmempty\ else\ None)\ |$
 $match\ (Free\ name)\ u = Some\ (fmap\ of\ list\ [(name, u)])\ |$
 $match\ (Bound\ n)\ u = None\ |$
 $match\ (Abs\ t)\ u = None$

lemma *match-simps*[*simp*]:
 $match\ (t_1\ \$\ t_2)\ (app\ u_1\ u_2) = do\ \{$
 $\quad env_1 \leftarrow match\ t_1\ u_1;$
 $\quad env_2 \leftarrow match\ t_2\ u_2;$
 $\quad Some\ (env_1\ ++_f\ env_2)$
 $\}$

$match (Const\ name) (const\ name') = (if\ name = name' then\ Some\ fmempty\ else\ None)$
 <proof>

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 $P\ t\ u\ env$
 <proof>

lemma *match-dom*: $match\ p\ t = Some\ env \implies fmdom\ env = frees\ p$
 <proof>

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

lemma *match-appE-split*:

assumes $match\ (t_1\ \$\ t_2)\ u = Some\ env$
obtains $u_1\ u_2\ env_1\ env_2$ **where**
 $u = app\ u_1\ u_2\ match\ t_1\ u_1 = Some\ env_1\ match\ t_2\ u_2 = Some\ env_2\ env = env_1$
 $++_f\ env_2$
 <proof>

lemma *subst-consts*:

assumes $consts\ t\ |\subseteq|\ S\ fmpred\ (\lambda\ u. consts\ u\ |\subseteq|\ S)\ env$
shows $consts\ (subst\ t\ env)\ |\subseteq|\ S$
 <proof>

lemma *subst-empty[simp]*: $subst\ t\ fmempty = t$
 <proof>

lemma *subst-drop-fset*: $fdisjnt\ S\ (frees\ t) \implies subst\ t\ (fmdrop-fset\ S\ env) = subst\ t\ env$
 <proof>

lemma *subst-restrict*:

assumes $frees\ t\ |\subseteq|\ M$
shows $subst\ t\ (fmrestrict-fset\ M\ env) = subst\ t\ env$
 <proof>

corollary *subst-restrict'[simp]*: $subst\ t\ (fmrestrict-fset\ (frees\ t)\ env) = subst\ t\ env$
 <proof>

corollary *subst-cong*:

assumes $\bigwedge x. x \in | \text{frees } t \implies \text{fmlookup } \Gamma_1 x = \text{fmlookup } \Gamma_2 x$
shows $\text{subst } t \Gamma_1 = \text{subst } t \Gamma_2$
 $\langle \text{proof} \rangle$

corollary *subst-add-disjnt*:
assumes $\text{fdisjnt } (\text{frees } t) (\text{fmdom } \text{env}_1)$
shows $\text{subst } t (\text{env}_1 ++_f \text{env}_2) = \text{subst } t \text{env}_2$
 $\langle \text{proof} \rangle$

corollary *subst-add-shadowed-env*:
assumes $\text{frees } t \subseteq | \text{fmdom } \text{env}_2$
shows $\text{subst } t (\text{env}_1 ++_f \text{env}_2) = \text{subst } t \text{env}_2$
 $\langle \text{proof} \rangle$

corollary *subst-restrict-closed*: $\text{closed-except } t S \implies \text{subst } t (\text{fmrestrict-fset } S \text{env}) = \text{subst } t \text{env}$
 $\langle \text{proof} \rangle$

lemma *subst-closed-except-id*:
assumes $\text{closed-except } t S \text{fdisjnt } (\text{fmdom } \text{env}) S$
shows $\text{subst } t \text{env} = t$
 $\langle \text{proof} \rangle$

lemma *subst-closed-except-preserved*:
assumes $\text{closed-except } t S \text{fdisjnt } (\text{fmdom } \text{env}) S$
shows $\text{closed-except } (\text{subst } t \text{env}) S$
 $\langle \text{proof} \rangle$

corollary *subst-closed-id*: $\text{closed } t \implies \text{subst } t \text{env} = t$
 $\langle \text{proof} \rangle$

corollary *subst-closed-preserved*: $\text{closed } t \implies \text{closed } (\text{subst } t \text{env})$
 $\langle \text{proof} \rangle$

context begin

private lemma *subst-indep0*:
assumes $\text{closed-env } \text{env}_2 \text{fdisjnt } (\text{fmdom } \text{env}_1) (\text{fmdom } \text{env}_2)$
shows $\text{subst } t (\text{env}_1 ++_f \text{env}_2) = \text{subst } (\text{subst } t \text{env}_2) \text{env}_1$
 $\langle \text{proof} \rangle$

lemma *subst-indep*:
assumes $\text{closed-env } \Gamma'$
shows $\text{subst } t (\Gamma ++_f \Gamma') = \text{subst } (\text{subst } t \Gamma') \Gamma$
 $\langle \text{proof} \rangle$

lemma *subst-indep'*:
assumes $\text{closed-env } \Gamma' \text{fdisjnt } (\text{fmdom } \Gamma') (\text{fmdom } \Gamma)$

shows $\text{subst } t (\Gamma' ++_f \Gamma) = \text{subst } (\text{subst } t \Gamma') \Gamma$
 ⟨proof⟩

lemma *subst-twice*:

assumes $\Gamma' \subseteq_f \Gamma$ *closed-env* Γ'

shows $\text{subst } (\text{subst } t \Gamma') \Gamma = \text{subst } t \Gamma$

⟨proof⟩

end

fun *matches* :: *term list* \Rightarrow *'a list* \Rightarrow (*name*, *'a*) *fmap option* **where**
matches [] [] = *Some fmempty* |
matches (t # ts) (u # us) = *do* { $\text{env}_1 \leftarrow \text{match } t \ u$; $\text{env}_2 \leftarrow \text{matches } ts \ us$; *Some*
 ($\text{env}_1 ++_f \text{env}_2$) } |
matches - - = *None*

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

context begin

private lemma *matches-alt-def0*:

assumes $\text{length } ps = \text{length } vs$

shows $\text{map-option } (\lambda \text{env}. m ++_f \text{env}) (\text{matches } ps \ vs) = \text{map-option } (\text{foldl } (++_f) \ m) (\text{those } (\text{map2 } \text{match } ps \ vs))$

⟨proof⟩

lemma *matches-alt-def*:

assumes $\text{length } ps = \text{length } vs$

shows $\text{matches } ps \ vs = \text{map-option } (\text{foldl } (++_f) \ \text{fmempty}) (\text{those } (\text{map2 } \text{match } ps \ vs))$

⟨proof⟩

end

lemma *matches-neq-length-none*[*simp*]: $\text{length } xs \neq \text{length } ys \implies \text{matches } xs \ ys = \text{None}$

⟨proof⟩

corollary *matches-some-eq-length*: $\text{matches } xs \ ys = \text{Some } \text{env} \implies \text{length } xs = \text{length } ys$

⟨proof⟩

lemma *matches-app*[*simp*]:

assumes $\text{length } xs_2 = \text{length } ys_2$

shows $\text{matches } (xs_1 @ xs_2) (ys_1 @ ys_2) =$

$\text{matches } xs_1 \ ys_1 \gg= (\lambda \text{env}_1. \text{matches } xs_2 \ ys_2 \gg= (\lambda \text{env}_2. \text{Some } (\text{env}_1 ++_f \text{env}_2)))$

⟨proof⟩

corollary *matches-appI*:

assumes $\text{matches } xs \ ys = \text{Some } env_1 \ \text{matches } xs' \ ys' = \text{Some } env_2$
shows $\text{matches } (xs \ @ \ xs') \ (ys \ @ \ ys') = \text{Some } (env_1 \ ++_f \ env_2)$
(proof)

corollary *matches-dom*:

assumes $\text{matches } ps \ ts = \text{Some } env$
shows $\text{fndom } env = \text{freess } ps$
(proof)

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*:

find-match (map $(\lambda(\text{pat}, t). (\text{pat}, f \ \text{pat} \ t)) \ cs) \ t =$
 map-option $(\lambda(\text{env}, \text{pat}, \text{rhs}). (\text{env}, \text{pat}, f \ \text{pat} \ \text{rhs})) \ (\text{find-match} \ cs \ t)$
(proof)

lemma *find-match-elem*:

assumes $\text{find-match } cs \ t = \text{Some } (\text{env}, \text{pat}, \text{rhs})$
shows $(\text{pat}, \text{rhs}) \in \text{set } cs \ \text{match } \text{pat} \ t = \text{Some } env$
(proof)

lemma *match-subst-closed*:

assumes $\text{match } \text{pat} \ t = \text{Some } env \ \text{closed-except } rhs \ (\text{frees } \text{pat}) \ \text{closed } t$
shows $\text{closed } (\text{subst } rhs \ env)$
(proof)

fun *rewrite-step* :: $(\text{term} \times 'a) \Rightarrow 'a \Rightarrow 'a \ \text{option}$ **where**
rewrite-step (t₁, t₂) u = map-option (subst t₂) (match t₁ u)

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

lemma *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)

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

lemma *matchesI[intro]*: $\text{subst } t \ \text{env} = u \implies t \lesssim u$

<proof>

lemma *matchesE[elim]*:

assumes $t \lesssim u$

obtains *env* **where** $\text{subst } t \text{ env} = u$

<proof>

definition *overlapping* :: $'a \Rightarrow 'a \Rightarrow \text{bool}$ **where**

$\text{overlapping } s \ t \longleftrightarrow (\exists u. s \lesssim u \wedge t \lesssim u)$

lemma *overlapping-refl*: $\text{overlapping } t \ t$

<proof>

lemma *overlapping-sym*: $\text{overlapping } t \ u \Longrightarrow \text{overlapping } u \ t$

<proof>

lemma *overlappingI[intro]*: $s \lesssim u \Longrightarrow t \lesssim u \Longrightarrow \text{overlapping } s \ t$

<proof>

lemma *overlappingE[elim]*:

assumes $\text{overlapping } s \ t$

obtains *u* **where** $s \lesssim u \ t \lesssim u$

<proof>

abbreviation $\text{non-overlapping } s \ t \equiv \neg \text{overlapping } s \ t$

corollary *non-overlapping-implies-neg*: $\text{non-overlapping } t \ u \Longrightarrow t \neq u$

<proof>

end

inductive *rewrite-first* :: $(\text{term} \times 'a::\text{term}) \text{ list} \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$ **where**

match: $\text{match } pat \ t = \text{Some } env \Longrightarrow \text{rewrite-first } ((pat, rhs) \# -) \ t \ (\text{subst } rhs \ env)$

|

nomatch: $\text{match } pat \ t = \text{None} \Longrightarrow \text{rewrite-first } cs \ t \ t' \Longrightarrow \text{rewrite-first } ((pat, -) \# cs) \ t \ t'$

code-pred $(modes: i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}) \ \text{rewrite-first} \ \langle proof \rangle$

lemma *rewrite-firstE*:

assumes $\text{rewrite-first } cs \ t \ t'$

obtains *pat rhs env* **where** $(pat, rhs) \in \text{set } cs \ \text{match } pat \ t = \text{Some } env \ t' = \text{subst } rhs \ env$

<proof>

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

lemma *find-match-rewrite-first*:

assumes $\text{find-match } cs \ t = \text{Some } (env, pat, rhs)$

shows *rewrite-first cs t (subst rhs env)*
 ⟨*proof*⟩

definition *term-cases* :: (name ⇒ 'b) ⇒ (name ⇒ 'b) ⇒ ('a ⇒ 'a ⇒ 'b) ⇒ 'b ⇒ 'a::term ⇒ 'b **where**

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

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

assumes *t = u otherwise1 = otherwise2*
assumes (∧ name. *t = const name ⇒ if-const1 name = if-const2 name*)
assumes (∧ name. *t = free name ⇒ if-free1 name = if-free2 name*)
assumes (∧ u₁ u₂. *t = app u₁ u₂ ⇒ if-app1 u₁ u₂ = if-app2 u₁ u₂*)
shows *term-cases if-const1 if-free1 if-app1 otherwise1 t = term-cases if-const2 if-free2 if-app2 otherwise2 u*
 ⟨*proof*⟩

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
 ⟨*proof*⟩

lemma *term-cases-template*:

assumes ∧ x. *f x = term-cases if-const if-free if-app otherwise x*
shows *f (const name) = if-const name*
and *f (free name) = if-free name*
and *f (app t u) = if-app t u*
 ⟨*proof*⟩

context *term begin*

function (*sequential*) *strip-comb* :: 'a ⇒ 'a × 'a list **where**

[*simp del*]: *strip-comb t =*
 (case unapp t of
 Some (t, u) ⇒
 (let (f, args) = *strip-comb t in (f, args @ [u])*)
 | None ⇒ (t, []))
 ⟨*proof*⟩

termination

$\langle proof \rangle$

lemma *strip-comb-simps*[simp]:

$strip-comb (app\ t\ u) = (let\ (f,\ args) = strip-comb\ t\ in\ (f,\ args\ @\ [u]))$

$unapp\ t = None \implies strip-comb\ t = (t,\ [])$

$\langle proof \rangle$

lemma *strip-comb-induct*[case-names app no-app]:

assumes $\bigwedge x\ y.\ P\ x \implies P\ (app\ x\ y)$

assumes $\bigwedge t.\ unapp\ t = None \implies P\ t$

shows $P\ t$

$\langle proof \rangle$

lemma *strip-comb-size*: $t' \in set\ (snd\ (strip-comb\ t)) \implies size\ t' < size\ t$

$\langle proof \rangle$

lemma *sstrip-comb-termination*[termination-simp]:

$(f,\ ts) = strip-comb\ t \implies t' \in set\ ts \implies size\ t' < size\ t$

$\langle proof \rangle$

lemma *strip-comb-empty*: $snd\ (strip-comb\ t) = [] \implies fst\ (strip-comb\ t) = t$

$\langle proof \rangle$

lemma *strip-comb-app*: $fst\ (strip-comb\ (app\ t\ u)) = fst\ (strip-comb\ t)$

$\langle proof \rangle$

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]: $list-comb\ f\ (xs\ @\ ys) = list-comb\ (list-comb\ f\ xs)\ ys$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *subst-list-comb*: $subst\ (list-comb\ f\ xs)\ env = list-comb\ (subst\ f\ env)\ (map\ (\lambda t.\ subst\ t\ env)\ xs)$

$\langle proof \rangle$

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

$const-list-comb\ name \equiv list-comb\ (const\ name)$

lemma *list-strip-comb*[simp]: $list-comb\ (fst\ (strip-comb\ t))\ (snd\ (strip-comb\ t)) = t$

$\langle proof \rangle$

lemma *strip-list-comb*: $\text{strip-comb } (\text{list-comb } f \text{ } ys) = (\text{fst } (\text{strip-comb } f), \text{snd } (\text{strip-comb } f) \text{ @ } ys)$
 ⟨proof⟩

lemma *strip-list-comb-const*: $\text{strip-comb } (\text{name } \$\$ \text{ } xs) = (\text{const } \text{name}, xs)$
 ⟨proof⟩

lemma *frees-list-comb[simp]*: $\text{frees } (\text{list-comb } t \text{ } xs) = \text{frees } t \text{ } |\cup| \text{ } \text{freess } xs$
 ⟨proof⟩

lemma *consts-list-comb*: $\text{consts } (\text{list-comb } f \text{ } xs) = \text{consts } f \text{ } |\cup| \text{ } \text{ffUnion } (\text{fset-of-list } (\text{map } \text{consts } xs))$
 ⟨proof⟩

lemma *ids-list-comb*: $\text{ids } (\text{list-comb } f \text{ } xs) = \text{ids } f \text{ } |\cup| \text{ } \text{ffUnion } (\text{fset-of-list } (\text{map } \text{ids } xs))$
 ⟨proof⟩

lemma *frees-strip-comb*: $\text{frees } t = \text{frees } (\text{fst } (\text{strip-comb } t)) \text{ } |\cup| \text{ } \text{freess } (\text{snd } (\text{strip-comb } t))$
 ⟨proof⟩

lemma *list-comb-cases'*:
obtains $(\text{app } \text{is-app } (\text{list-comb } f \text{ } xs) \text{ } | \text{ (empty) } \text{list-comb } f \text{ } xs = f \text{ } xs = [])$
 ⟨proof⟩

lemma *list-comb-cases[consumes 1]*:
assumes $t = \text{list-comb } f \text{ } xs$
obtains $(\text{head}) t = f \text{ } xs = [] \text{ } | (\text{app}) u \text{ } v \text{ } \mathbf{where} \text{ } t = \text{app } u \text{ } v$
 ⟨proof⟩

end

fun *left-nesting* :: 'a::term \Rightarrow nat **where**
 [simp del]: $\text{left-nesting } t = \text{term-cases } (\lambda-. 0) (\lambda-. 0) (\lambda t u. \text{Suc } (\text{left-nesting } t)) 0$
 t

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

lemma *list-comb-nesting[simp]*: $\text{left-nesting } (\text{list-comb } f \text{ } xs) = \text{left-nesting } f \text{ } + \text{length } xs$
 ⟨proof⟩

lemma *list-comb-cond-inj*:
assumes $\text{list-comb } f \text{ } xs = \text{list-comb } g \text{ } ys \text{ } \text{left-nesting } f = \text{left-nesting } g$

shows $xs = ys \ f = g$
 ⟨*proof*⟩

lemma *list-comb-inj-second*: *inj (list-comb f)*
 ⟨*proof*⟩

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*⟩

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 $P t$
 ⟨*proof*⟩

lemma *no-abs-cases*[*consumes 1, cases pred: no-abs*]:
assumes *no-abs t*
obtains (*free*) *name* **where** $t = \text{free name}$
 | (*const*) *name* **where** $t = \text{const name}$
 | (*app*) $t_1 t_2$ **where** $t = \text{app } t_1 t_2 \ \text{no-abs } t_1 \ \text{no-abs } t_2$
 ⟨*proof*⟩

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 $P :: 'a :: \text{term} \Rightarrow \text{bool}$
assumes *app: P (app t u) ⟷ P t ∧ P u*
begin

context
notes *app*[*simp*]
begin

lemma *list-comb*: $P (\text{list-comb } f \text{ } xs) \longleftrightarrow P f \wedge \text{list-all } P \text{ } xs$
<proof>

corollary *list-combE*:
 assumes $P (\text{list-comb } f \text{ } xs)$
 shows $P f \wedge x \in \text{set } xs \implies P x$
<proof>

lemma *match*:
 assumes $\text{match pat } t = \text{Some env } P t$
 shows $\text{fmpred } (\lambda-. P) \text{ env}$
<proof>

lemma *matchs*:
 assumes $\text{matchs pats } ts = \text{Some env list-all } P \text{ } ts$
 shows $\text{fmpred } (\lambda-. P) \text{ env}$
<proof>

end

end

locale *subst-syntactic-and* = *simple-syntactic-and* +
 assumes $\text{subst: } P t \implies \text{fmpred } (\lambda-. P) \text{ env} \implies P (\text{subst } t \text{ env})$
begin

lemma *rewrite-step*:
 assumes $(\text{lhs}, \text{rhs}) \vdash t \rightarrow t' \wedge P t \wedge P \text{ rhs}$
 shows $P t'$
<proof>

end

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

context
 notes $\text{app}[\text{simp}]$
begin

lemma *list-comb*: $P (\text{list-comb } f \text{ } xs) \longleftrightarrow P f \vee \text{list-ex } P \text{ } xs$
<proof>

lemma *match*:
 assumes $\text{match pat } t = \text{Some env } \neg P t$
 shows $\text{fmpred } (\lambda-. t. \neg P t) \text{ env}$

<proof>

end

sublocale *neg: simple-syntactic-and* $\lambda t. \neg P t$

<proof>

end

global-interpretation *no-abs: simple-syntactic-and no-abs*

<proof>

global-interpretation *closed: simple-syntactic-and* $\lambda t. \text{closed-except } t \text{ } S \text{ for } S$

<proof>

global-interpretation *closed: subst-syntactic-and closed*

<proof>

corollary *closed-list-comb: closed* $(\text{name } \$\$ \text{ args}) \longleftrightarrow \text{list-all closed args}$

<proof>

locale *term-struct-rel =*

fixes $P :: 'a::\text{term} \Rightarrow 'b::\text{term} \Rightarrow \text{bool}$

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

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

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

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

begin

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

$P\text{-env} \equiv \text{fmrel } P$

lemma *related-match:*

assumes $\text{match } x \ u = \text{Some env } P \ t \ u$

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

<proof>

lemma *list-combI:*

assumes $\text{list-all2 } P \ us_1 \ us_2 \ P \ t_1 \ t_2$

shows $P (\text{list-comb } t_1 \ us_1) (\text{list-comb } t_2 \ us_2)$

<proof>

lemma *list-combE:*

assumes $P \ t (\text{name } \$\$ \ \text{args})$

obtains args' **where** $t = \text{name } \$\$ \ \text{args}' \ \text{list-all2 } P \ \text{args}' \ \text{args}$

<proof>

end

locale *term-struct-rel-strong* = *term-struct-rel* +
assumes *P-const-t*: $P \text{ (const name) } t \implies t = \text{const name}$
assumes *P-app-t*: $P \text{ (app } u_1 \ u_2) } t \implies \exists t_1 \ t_2. t = \text{app } t_1 \ t_2 \wedge P \ u_1 \ t_1 \wedge P \ u_2$
 t_2
begin

lemma *unconst-rel*: $P \ t \ u \implies \text{unconst } t = \text{unconst } u$
 $\langle \text{proof} \rangle$

lemma *unapp-rel*: $P \ t \ u \implies \text{rel-option (rel-prod } P \ P) (\text{unapp } t) (\text{unapp } u)$
 $\langle \text{proof} \rangle$

lemma *match-rel*:
assumes $P \ t \ u$
shows $\text{rel-option } P\text{-env (match } p \ t) (\text{match } p \ u)$
 $\langle \text{proof} \rangle$

lemma *find-match-rel*:
assumes $\text{list-all2 (rel-prod (=) } P) \ cs \ cs' \ P \ t \ t'$
shows $\text{rel-option (rel-prod } P\text{-env (rel-prod (=) } P)) (\text{find-match } cs \ t) (\text{find-match}$
 $cs' \ t')$
 $\langle \text{proof} \rangle$

end

fun *convert-term* :: $'a::\text{term} \Rightarrow 'b::\text{term}$ **where**
 $[\text{simp del}]$: $\text{convert-term } t = \text{term-cases const free } (\lambda t \ u. \text{app (convert-term } t)$
 $(\text{convert-term } u)) \ \text{undefined } t$

lemmas $\text{convert-term-simps}[\text{simp}] = \text{term-cases-template}[OF \ \text{convert-term.simps}]$

lemma *convert-term-id*:
assumes $\text{no-abs } t$
shows $\text{convert-term } t = t$
 $\langle \text{proof} \rangle$

lemma *convert-term-no-abs*:
assumes $\text{no-abs } t$
shows $\text{no-abs (convert-term } t)$
 $\langle \text{proof} \rangle$

lemma *convert-term-inj*:
assumes $\text{no-abs } t \ \text{no-abs } t' \ \text{convert-term } t = \text{convert-term } t'$
shows $t = t'$
 $\langle \text{proof} \rangle$

lemma *convert-term-idem*:
assumes $\text{no-abs } t$
shows $\text{convert-term (convert-term } t) = \text{convert-term } t$

<proof>

lemma *convert-term-frees*[simp]:
 assumes *no-abs t*
 shows *frees (convert-term t) = frees t*
<proof>

lemma *convert-term-consts*[simp]:
 assumes *no-abs t*
 shows *consts (convert-term t) = consts t*
<proof>

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

lemma *convert-term-match*:
 assumes *match t u = Some env*
 shows *match t (convert-term u) = Some (fmmap convert-term env)*
<proof>

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>
```

end

```
lemma is-const-free[simp]:  $\neg$  is-const (Free name)
<proof>
```

```
lemma is-free-app[simp]:  $\neg$  is-free (t $ u)
<proof>
```

```
lemma is-free-free[simp]: is-free (Free name)
<proof>
```

```
lemma is-const-const[simp]: is-const (Const name)
<proof>
```

```
lemma list-comb-free: is-free (list-comb f xs)  $\implies$  is-free f
<proof>
```

```
lemma const-list-comb-free[simp]:  $\neg$  is-free (name $$ args)
<proof>
```

```
corollary const-list-comb-neq-free[simp]: name $$ args  $\neq$  free name'
<proof>
```

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

lemma *match-list-comb-list-comb-eq-lengths*[simp]:
assumes $\text{length } ps = \text{length } vs$
shows $\text{match } (\text{list-comb } f \ ps) \ (\text{list-comb } g \ vs) =$
 $(\text{case match } f \ g \ \text{of}$
 $\quad \text{Some } env \Rightarrow$
 $\quad (\text{case those } (\text{map2 match } ps \ vs) \ \text{of}$
 $\quad \quad \text{Some } envs \Rightarrow \text{Some } (\text{foldl } (++) \ env \ envs)$
 $\quad \quad | \ \text{None} \Rightarrow \text{None})$
 $\quad | \ \text{None} \Rightarrow \text{None})$
 $\langle \text{proof} \rangle$

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

fun *bounds* :: $\text{term} \Rightarrow \text{nat fset}$ **where**
 $\text{bounds } (\text{Bound } i) = \{ | \ i \ | \}$ |
 $\text{bounds } (t_1 \ \$ \ t_2) = \text{bounds } t_1 \ \cup \ \text{bounds } t_2$ |
 $\text{bounds } (\Lambda \ t) = (\lambda i. \ i - 1) \ |^{|} \ (\text{bounds } t - \{ | \ 0 \ | \})$ |
 $\text{bounds } - = \{ | \ | \}$

definition *shift-nat* :: $\text{nat} \Rightarrow \text{int} \Rightarrow \text{nat}$ **where**
 $[\text{simp}]: \text{shift-nat } n \ k = (\text{if } k \geq 0 \ \text{then } n + \text{nat } k \ \text{else } n - \text{nat } |k|)$

fun *incr-bounds* :: $\text{int} \Rightarrow \text{nat} \Rightarrow \text{term} \Rightarrow \text{term}$ **where**
 $\text{incr-bounds } inc \ lev \ (\text{Bound } i) = (\text{if } i \geq lev \ \text{then } \text{Bound } (\text{shift-nat } i \ inc) \ \text{else } \text{Bound}$
 $\ i) \ |$
 $\text{incr-bounds } inc \ lev \ (\Lambda \ u) = \Lambda \ \text{incr-bounds } inc \ (lev + 1) \ u \ |$
 $\text{incr-bounds } inc \ lev \ (t_1 \ \$ \ t_2) = \text{incr-bounds } inc \ lev \ t_1 \ \$ \ \text{incr-bounds } inc \ lev \ t_2 \ |$
 $\text{incr-bounds } - - \ t = t$

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

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

fun *replace-bound* :: $\text{nat} \Rightarrow \text{term} \Rightarrow \text{term} \Rightarrow \text{term}$ **where**
 $\text{replace-bound } lev \ (\text{Bound } i) \ t = (\text{if } i < lev \ \text{then } \text{Bound } i \ \text{else if } i = lev \ \text{then}$
 $\ \text{incr-bounds } (\text{int } lev) \ 0 \ t \ \text{else } \text{Bound } (i - 1)) \ |$
 $\text{replace-bound } lev \ (t_1 \ \$ \ t_2) \ t = \text{replace-bound } lev \ t_1 \ t \ \$ \ \text{replace-bound } lev \ t_2 \ t \ |$
 $\text{replace-bound } lev \ (\Lambda \ u) \ t = \Lambda \ \text{replace-bound } (lev + 1) \ u \ t \ |$
 $\text{replace-bound } - \ t = t$

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

lemma *replace-bound-frees*: $\text{frees } (\text{replace-bound } n \ t \ t') \ \subseteq \ \text{frees } t \ \cup \ \text{frees } t'$
 $\langle \text{proof} \rangle$

lemma *replace-bound-eq*:
assumes $i \notin | \text{bounds } t$
shows $\text{replace-bound } i \ t \ t' = \text{incr-bounds } (-1) \ (i + 1) \ t$
 $\langle \text{proof} \rangle$

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

lemma *wellformed-inc*:
assumes $\text{wellformed}' \ k \ t \ k \leq n$
shows $\text{wellformed}' \ n \ t$
 $\langle \text{proof} \rangle$

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

lemma *wellformed'-replace-bound-eq*:
assumes $\text{wellformed}' \ n \ t \ k \geq n$
shows $\text{replace-bound } k \ t \ u = t$
 $\langle \text{proof} \rangle$

lemma *wellformed-replace-bound-eq*: $\text{wellformed } t \Longrightarrow \text{replace-bound } k \ t \ u = t$
 $\langle \text{proof} \rangle$

lemma *incr-bounds-eq*: $n \geq k \Longrightarrow \text{wellformed}' \ k \ t \Longrightarrow \text{incr-bounds } i \ n \ t = t$
 $\langle \text{proof} \rangle$

lemma *incr-bounds-subst*:
assumes $\bigwedge t. t \in \text{fmran}' \ \text{env} \Longrightarrow \text{wellformed } t$
shows $\text{incr-bounds } i \ n \ (\text{subst } t \ \text{env}) = \text{subst } (\text{incr-bounds } i \ n \ t) \ \text{env}$
 $\langle \text{proof} \rangle$

lemma *incr-bounds-wellformed*:
assumes $\text{wellformed}' \ m \ u$
shows $\text{wellformed}' \ (k + m) \ (\text{incr-bounds } (i \ k) \ n \ u)$
 $\langle \text{proof} \rangle$

lemma *replace-bound-wellformed*:
assumes $\text{wellformed } u \ \text{wellformed}' \ (\text{Suc } k) \ t \ i \leq k$
shows $\text{wellformed}' \ k \ (\text{replace-bound } i \ t \ u)$
 $\langle \text{proof} \rangle$

lemma *subst-wellformed*:
assumes $\text{wellformed}' \ n \ t \ \text{fmpred } (\lambda-. \ \text{wellformed}) \ \text{env}$
shows $\text{wellformed}' \ n \ (\text{subst } t \ \text{env})$

<proof>

global-interpretation *wellformed: simple-syntactic-and wellformed' n for n*

<proof>

global-interpretation *wellformed: subst-syntactic-and wellformed*

<proof>

lemma *match-list-combE:*

assumes *match (name \$\$ xs) t = Some env*

obtains *ys where t = name \$\$ ys matchs xs ys = Some env*

<proof>

lemma *left-nesting-neq-match:*

left-nesting f ≠ left-nesting g ⇒ is-const (fst (strip-comb f)) ⇒ match f g = None

<proof>

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*

<proof>

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>

end

lemma *match-list-comb-list-comb-none-name[simp]:*

assumes *name ≠ name'*

shows *match (name \$\$ ps) (name' \$\$ vs) = None*

<proof>

lemma *match-list-comb-list-comb-none-length[simp]:*

assumes *length ps ≠ length vs*

shows *match (name \$\$ ps) (name' \$\$ vs) = None*

<proof>

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

corollary *related-matches:*

assumes *matchs ps ts₂ = Some env₂ list-all2 P ts₁ ts₂*

obtains env_1 **where** $matches\ ps\ ts_1 = Some\ env_1\ P-env\ env_1\ env_2$
<proof>

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>
```

```
fun linears :: term list  $\Rightarrow$  bool where
linears []  $\longleftrightarrow$  True |
linears (t # ts)  $\longleftrightarrow$  linear t  $\wedge$  fdisjnt (frees t) (frees ts)  $\wedge$  linears ts
```

lemma *linears-butlastI*[intro]: *linears ts* \implies *linears (butlast ts)*
<proof>

lemma *linears-appI*[intro]:
 assumes *linears xs linears ys fdisjnt (freess xs) (freess ys)*
 shows *linears (xs @ ys)*
<proof>

lemma *linears-linear*: *linears ts* \implies $t \in \text{set } ts \implies$ *linear t*
<proof>

lemma *linears-singleI*[intro]: *linear t* \implies *linears [t]*
<proof>

lemma *linear-strip-comb*: *linear t* \implies *linear (fst (strip-comb t))*
<proof>

lemma *linears-strip-comb*: *linear t* \implies *linears (snd (strip-comb t))*
<proof>

lemma *linears-appendD*:
 assumes *linears (xs @ ys)*
 shows *linears xs linears ys fdisjnt (freess xs) (freess ys)*
<proof>

lemma *linear-list-comb*:
 assumes *linear f linears xs fdisjnt (frees f) (freess xs) \neg is-free f*
 shows *linear (list-comb f xs)*
<proof>

corollary *linear-list-comb'*: *linears xs* \implies *linear (name \$\$ xs)*
<proof>

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*
<proof>

lemma *wellformed-linearI*: *linear t* \implies *wellformed' n t*
<proof>

lemma *pat-cases*:
 obtains *(free) s* **where** *t = Free s*
 | *(comb) name args* **where** *linears args t = name \$\$ args*
 | *(nonlinear) \neg linear t*
<proof>

corollary *linear-pat-cases*[consumes 1]:

assumes *linear t*
obtains (*free*) *s* **where** $t = \text{Free } s$
| (*comb*) *name args* **where** *linears args t = name \$\$ args*
⟨*proof*⟩

lemma *linear-pat-induct*[*consumes 1, case-names free comb*]:

assumes *linear t*
assumes $\bigwedge s. P (\text{Free } s)$
assumes $\bigwedge \text{name args. linears args} \implies (\bigwedge \text{arg. arg} \in \text{set args} \implies P \text{ arg}) \implies P$
(*name \$\$ args*)
shows $P t$
⟨*proof*⟩

context begin

private lemma *match-subst-correctness0*:

assumes *linear t*
shows *case match t u of*
 $\text{None} \implies (\forall \text{env. subst (convert-term } t) \text{ env} \neq u) \mid$
 $\text{Some env} \implies \text{subst (convert-term } t) \text{ env} = u$
⟨*proof*⟩

lemma *match-subst-some*[*simp*]:

$\text{match } t \text{ u} = \text{Some env} \implies \text{linear } t \implies \text{subst (convert-term } t) \text{ env} = u$
⟨*proof*⟩

lemma *match-subst-none*:

$\text{match } t \text{ u} = \text{None} \implies \text{linear } t \implies \text{subst (convert-term } t) \text{ env} = u \implies \text{False}$
⟨*proof*⟩

end

lemma *match-matches*: $\text{match } t \text{ u} = \text{Some env} \implies \text{linear } t \implies t \lesssim u$
⟨*proof*⟩

lemma *overlapping-var1I*: *overlapping (Free name) t*
⟨*proof*⟩

lemma *overlapping-var2I*: *overlapping t (Free name)*
⟨*proof*⟩

lemma *non-overlapping-appI1*: $\text{non-overlapping } t_1 \ u_1 \implies \text{non-overlapping } (t_1 \$ t_2) (u_1 \$ u_2)$
⟨*proof*⟩

lemma *non-overlapping-appI2*: $\text{non-overlapping } t_2 \ u_2 \implies \text{non-overlapping } (t_1 \$ t_2) (u_1 \$ u_2)$

<proof>

lemma *non-overlapping-app-constI: non-overlapping (t₁ \$ t₂) (Const name)*
<proof>

lemma *non-overlapping-const-appI: non-overlapping (Const name) (t₁ \$ t₂)*
<proof>

lemma *non-overlapping-const-constI: x ≠ y ⇒ non-overlapping (Const x) (Const y)*
<proof>

lemma *match-overlapping:*
 assumes *linear t₁ linear t₂*
 assumes *match t₁ u = Some env₁ match t₂ u = Some env₂*
 shows *overlapping t₁ t₂*
<proof>

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 ( $\Lambda_n$  -. - [0, 50] 50) |  
  Napp nterm nterm (infixl  $\$_n$  70)
```

```
derive linorder nterm
```

```
instantiation nterm :: pre-term begin
```

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

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

```
definition const-nterm where  
const-nterm = Nconst
```

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

```
definition free-nterm where  
free-nterm = Nvar
```

```

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

fun frees-nterm :: nterm  $\Rightarrow$  name fset where
frees-nterm (Nvar x) = { | x | } |
frees-nterm (t1 $n t2) = frees-nterm t1 | $\cup$ | frees-nterm t2 |
frees-nterm ( $\Lambda_n$  x. t) = frees-nterm t - { | x | } |
frees-nterm (Nconst -) = { || }

fun subst-nterm :: nterm  $\Rightarrow$  (name, nterm) fmap  $\Rightarrow$  nterm where
subst-nterm (Nvar s) env = (case fmlookup env s of Some t  $\Rightarrow$  t | None  $\Rightarrow$  Nvar
s) |
subst-nterm (t1 $n t2) env = subst-nterm t1 env $n subst-nterm t2 env |
subst-nterm ( $\Lambda_n$  x. t) env = ( $\Lambda_n$  x. subst-nterm t (fmdrop x env)) |
subst-nterm t env = t

fun consts-nterm :: nterm  $\Rightarrow$  name fset where
consts-nterm (Nconst x) = { | x | } |
consts-nterm (t1 $n t2) = consts-nterm t1 | $\cup$ | consts-nterm t2 |
consts-nterm (Nabs - t) = consts-nterm t |
consts-nterm (Nvar -) = { || }

instance
<proof>

end

instantiation nterm :: term begin

definition abs-pred-nterm :: (nterm  $\Rightarrow$  bool)  $\Rightarrow$  nterm  $\Rightarrow$  bool where
[code del]: abs-pred P t  $\longleftrightarrow$  ( $\forall t'$  x. t = ( $\Lambda_n$  x. t')  $\longrightarrow$  P t'  $\longrightarrow$  P t)

instance <proof>

end

lemma no-abs-abs[simp]:  $\neg$  no-abs ( $\Lambda_n$  x. t)
<proof>

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  $\langle$ proof $\rangle$ 
```

```
abbreviation alpha-eq :: nterm  $\Rightarrow$  nterm  $\Rightarrow$  bool (infixl  $\approx_\alpha$  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
 $\langle$ proof $\rangle$ 
```

```
corollary alpha-eq-refl: alpha-eq t t
```

<proof>

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 Γ t) x) = convert-term t*

<proof>

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 Γ x)*

<proof>

lemma *term-to-nterm-vars0*:

assumes *wellformed' (length Γ) t*

shows *frees (fst (run-state (term-to-nterm Γ t) s)) | \subseteq | frees t | \cup | fset-of-list Γ*

<proof>

including *fset.lifting* *<proof>*

corollary *term-to-nterm-vars*:

assumes *wellformed t*

shows *frees (fresh-frun (term-to-nterm [] t) F) | \subseteq | frees t*

<proof>

corollary *term-to-nterm-closed*: *closed t \implies wellformed t \implies closed (term-to-nterm' t)*

<proof>

lemma *term-to-nterm-consts*: *pred-state ($\lambda t'. consts t' = consts t$) (term-to-nterm*

Γt
 $\langle proof \rangle$

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

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

lemma *nterm-to-term*:
assumes *no-abs t fdisjnt (fset-of-list* Γ) (*frees t*)
shows *nterm-to-term* Γ *t* = *convert-term t*
 $\langle proof \rangle$

abbreviation *nterm-to-term'* \equiv *nterm-to-term* \square

lemma *nterm-to-term'*: *no-abs t* \implies *nterm-to-term'* *t* = *convert-term t*
 $\langle proof \rangle$

lemma *nterm-to-term-frees[simp]*: *frees (nterm-to-term* Γ *t*) = *frees t* - *fset-of-list*
 Γ
 $\langle proof \rangle$
including *fset.lifting*
 $\langle proof \rangle$
including *fset.lifting*
 $\langle proof \rangle$

6.4 Correctness

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

lemma *term-to-nterm-nterm-to-term0*:
assumes *wellformed' (length* Γ) *t fdisjnt (fset-of-list* Γ) (*frees t*) *distinct* Γ
assumes *fBall (frees t* \cup *fset-of-list* Γ) ($\lambda x. x \leq s$)
shows *nterm-to-term* Γ (*fst (run-state (term-to-nterm* Γ *t*) *s*) = *t*
 $\langle proof \rangle$
including *fset.lifting* $\langle proof \rangle$

lemma *term-to-nterm-nterm-to-term*:
assumes *wellformed t frees t* \subseteq *S*
shows *nterm-to-term'* (*frun-fresh (term-to-nterm* \square *t*) (*S* \cup *Q*)) = *t*
 $\langle proof \rangle$

corollary *term-to-nterm-nterm-to-term-simple*:
assumes *wellformed t*

shows $nterm\text{-to-term}' (term\text{-to-nterm}' t) = t$
<proof>

lemma *nterm-to-term-eq*:

assumes $frees\ u \mid \subseteq \mid fset\text{-of-list}\ (common\text{-prefix}\ \Gamma\ \Gamma')$

shows $nterm\text{-to-term}\ \Gamma\ u = nterm\text{-to-term}\ \Gamma'\ u$

<proof>

including *fset.lifting*

<proof>

corollary *nterm-to-term-eq-closed*: $closed\ t \implies nterm\text{-to-term}\ \Gamma\ t = nterm\text{-to-term}\ \Gamma'\ t$

<proof>

lemma *nterm-to-term-wellformed*: $wellformed'\ (length\ \Gamma)\ (nterm\text{-to-term}\ \Gamma\ t)$

<proof>

corollary *nterm-to-term-closed-wellformed*: $closed\ t \implies wellformed\ (nterm\text{-to-term}\ \Gamma\ t)$

<proof>

lemma *nterm-to-term-term-to-nterm*:

assumes $frees\ t \mid \subseteq \mid fset\text{-of-list}\ \Gamma\ length\ \Gamma = length\ \Gamma'$

shows $alpha\text{-equiv}\ (fmap\text{-of-list}\ (zip\ \Gamma\ \Gamma'))\ t\ (fst\ (run\text{-state}\ (term\text{-to-nterm}\ \Gamma'\ (nterm\text{-to-term}\ \Gamma\ t))\ s))$

<proof>

corollary *nterm-to-term-term-to-nterm'*: $closed\ t \implies t \approx_\alpha term\text{-to-nterm}'\ (nterm\text{-to-term}'\ t)$

<proof>

context begin

private lemma *term-to-nterm-alpha-equiv0*:

$length\ \Gamma 1 = length\ \Gamma 2 \implies distinct\ \Gamma 1 \implies distinct\ \Gamma 2 \implies wellformed'\ (length\ \Gamma 1)\ t1 \implies$

$fresh\text{-fin}\ (frees\ t1 \mid \cup \mid fset\text{-of-list}\ \Gamma 1)\ s1 \implies fdisjnt\ (fset\text{-of-list}\ \Gamma 1)\ (frees\ t1) \implies$

$fresh\text{-fin}\ (frees\ t1 \mid \cup \mid fset\text{-of-list}\ \Gamma 2)\ s2 \implies fdisjnt\ (fset\text{-of-list}\ \Gamma 2)\ (frees\ t1) \implies$

$alpha\text{-equiv}\ (fmap\text{-of-list}\ (zip\ \Gamma 1\ \Gamma 2))\ (fst\ (run\text{-state}\ (term\text{-to-nterm}\ \Gamma 1\ t1)\ s1))\ (fst\ (run\text{-state}\ (term\text{-to-nterm}\ \Gamma 2\ t1)\ s2))$

<proof>

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 $fresh\text{-fin}\ (fset\text{-of-list}\ \Gamma 1)\ s1\ fresh\text{-fin}\ (fset\text{-of-list}\ \Gamma 2)\ s2$

shows $alpha\text{-equiv}\ (fmap\text{-of-list}\ (zip\ \Gamma 1\ \Gamma 2))\ (fst\ (run\text{-state}\ (term\text{-to-nterm}\ \Gamma 1\ t)\ s1))\ (fst\ (run\text{-state}\ (term\text{-to-nterm}\ \Gamma 2\ t)\ s2))$

t) $s1$) (fst ($run\text{-}state$ ($term\text{-}to\text{-}nterm$ $\Gamma 2$ 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*.
 $\langle proof \rangle$

end

global-interpretation *nrelated: term-struct-rel-strong* ($\lambda t n. t = nterm\text{-}to\text{-}term$ Γn) **for** Γ
 $\langle proof \rangle$

lemma *env-nrelated-closed*:
assumes *nrelated.P-env* Γ *env nenv closed-env nenv*
shows *nrelated.P-env* Γ' *env nenv*
 $\langle proof \rangle$

lemma *nrelated-subst*:
assumes *nrelated.P-env* Γ *env nenv closed-env nenv fdisjnt* ($fset\text{-}of\text{-}list$ Γ) ($fmdom$ *nenv*)
shows *subst* ($nterm\text{-}to\text{-}term$ Γ t) *env* = $nterm\text{-}to\text{-}term$ Γ (*subst* t *nenv*)
 $\langle proof \rangle$
including *fset.lifting* $\langle proof \rangle$

lemma *nterm-to-term-insert-dupl*:
assumes $y \in set$ ($take$ n Γ) $n \leq length$ Γ
shows $nterm\text{-}to\text{-}term$ Γ $t = incr\text{-}bounds$ ($- 1$) (Suc n) ($nterm\text{-}to\text{-}term$ ($insert\text{-}nth$ n y Γ) t)
 $\langle proof \rangle$

lemma *nterm-to-term-bounds-dupl*:
assumes $i < length$ Γ $j < length$ Γ $i < j$
assumes $\Gamma ! i = \Gamma ! j$
shows $j \notin | bounds$ ($nterm\text{-}to\text{-}term$ Γ t)
 $\langle proof \rangle$

fun *subst-single* :: $nterm \Rightarrow name \Rightarrow nterm \Rightarrow nterm$ **where**
subst-single ($Nvar$ s) $s' t' = (if$ $s = s'$ $then$ t' $else$ $Nvar$ s) |
subst-single (t_1 $\$n$ t_2) $s' t' = subst\text{-}single$ t_1 $s' t' \$n$ *subst-single* t_2 $s' t'$ |
subst-single (Λ_n $x. t$) $s' t' = (\Lambda_n$ $x. (if$ $x = s'$ $then$ t $else$ *subst-single* t $s' t'))$ |
subst-single $t - - = t$

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

lemma *nterm-to-term-subst-replace-bound*:
assumes *closed* u' $n \leq length$ Γ $x \notin set$ ($take$ n Γ)
shows $nterm\text{-}to\text{-}term$ Γ (*subst-single* u x u') = *replace-bound* n ($nterm\text{-}to\text{-}term$ ($insert\text{-}nth$ n x Γ) u) ($nterm\text{-}to\text{-}term$ Γ u')
 $\langle proof \rangle$

corollary *nterm-to-term-subst-β*:

assumes *closed u'*

shows $nterm\text{-to-term } \Gamma (subst\ u (fmap\text{-of-list } [(x, u')])) = nterm\text{-to-term } (x \#$
 $\Gamma) u [nterm\text{-to-term } \Gamma u']_{\beta}$
<proof>

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
 $\langle$ proof $\rangle$ 
```

```
lemma of-to-name[simp]: of-name (to-name a) = a
 $\langle$ proof $\rangle$ 
```

```
lemma of-name-inj: of-name name1 = of-name name2  $\implies$  name1 = name2
```

```

⟨proof⟩

end

instantiation name :: is-name begin

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

instance ⟨proof⟩

end

lemma [simp, code-unfold]: (to-name :: name ⇒ name) = id
⟨proof⟩

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

definition app-trm where
app-trm = Comb

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

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

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

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

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

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

context
  includes fset.lifting
begin

lift-definition frees-trm :: 'a trm ⇒ name fset is λt. to-name ' vars-of t
  ⟨proof⟩

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
⟨proof⟩

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
⟨proof⟩

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⟩

end

lemma assoc-alt-def[simp]:
  assoc x y t = (case map-of t x of Some y' ⇒ y' | - ⇒ y)
⟨proof⟩

lemma subst-eq: Unification.subst t s = subst t (fmap-of-list s)
⟨proof⟩

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
```

⟨proof⟩

lift-definition *consts-tm* :: ('a, 'b) tm ⇒ name fset is λt. to-name ' syms t
⟨proof⟩

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*

⟨proof⟩

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*

⟨proof⟩

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)))

⟨proof⟩

instance

⟨proof⟩

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⟩

end

lemma *apps-list-comb*: apps f xs = list-comb f xs

⟨proof⟩

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.