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

Contents

Chapter 1

Names as a unique datatype

theory *Name*
imports *Main*
begin

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

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

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

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

— Mostly copied from *List-Lexorder*

instantiation *name* :: *ord*
begin

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

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

instance ⟨*proof*⟩

end

instance *name* :: *order*
⟨*proof*⟩

instance *name* :: *linorder*
⟨*proof*⟩

lemma *less-name-code*[*code*]:
Name xs < *Name* [] ⟷ *False*
Name [] < *Name* (*x* # *xs*) ⟷ *True*
Name (*x* # *xs*) < *Name* (*y* # *ys*) ⟷ (*of-char x*::*nat*) < *of-char y* ∨ *x* = *y* ∧ *Name xs* < *Name ys*
⟨*proof*⟩

lemma *le-name-code*[*code*]:
Name (*x* # *xs*) ≤ *Name* [] ⟷ *False*
Name [] ≤ *Name* (*x* # *xs*) ⟷ *True*
Name (*x* # *xs*) ≤ *Name* (*y* # *ys*) ⟷ (*of-char x*::*nat*) < *of-char y* ∨ *x* = *y* ∧ *Name xs* ≤ *Name ys*
⟨*proof*⟩

context begin

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

lemma *name-append-less*:
assumes *xs* ≠ *Name* []
shows *append ys xs* > *ys*
⟨*proof*⟩

end

end

Chapter 2

A monad for generating fresh names

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

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

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

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

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

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

lemma *update-next-strict-mono*[*simp*, *intro*]: *strict-mono-state update-next*
⟨*proof*⟩

lemma *update-next-mono*[*simp*, *intro*]: *mono-state update-next*
⟨*proof*⟩

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

lemma *create-alt-def*[*code*]: *create* = *State* (λ*a*. (*next a*, *next a*))
⟨*proof*⟩

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

lemma *next-ge-all*: *finite S* ⟹ *fresh-in S s* ⟹ *next s* ∉ *S*
⟨*proof*⟩

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

lemma *Next-ge-max*: *finite S* ⟹ *S* ≠ {} ⟹ *Next S* > *Max S*
⟨*proof*⟩

lemma *Next-not-member-subset*: *finite S'* ⟹ *S* ⊆ *S'* ⟹ *Next S'* ∉ *S*
⟨*proof*⟩

lemma *Next-not-member*: *finite S* ⟹ *Next S* ∉ *S*
⟨*proof*⟩

lemma *Next-geq-not-member*: *finite S* ⟹ *s* ≥ *Next S* ⟹ *s* ∉ *S*
⟨*proof*⟩

lemma *next-not-member*: *finite S* ⟹ *s* ≥ *Next S* ⟹ *next s* ∉ *S*
⟨*proof*⟩

lemma *create-mono*[*simp*, *intro*]: *mono-state create*
⟨*proof*⟩

lemma *create-strict-mono*[*simp*, *intro*]: *strict-mono-state create*
⟨*proof*⟩

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

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

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

lemma *next-ge-fall*: *fresh-fin S s* ⟹ *next s* |∉| *S*
⟨*proof*⟩

lift-definition *fNext* :: '*a fset* ⇒ '*a* **is** *Next* ⟨*proof*⟩

lemma *fNext-ge-max*: $S \neq \{||\} \Longrightarrow fNext\ S > fMax\ S$
⟨*proof*⟩

lemma *next-not-fmember*: $s \geq fNext\ S \Longrightarrow next\ s\ |\notin|\ S$
⟨*proof*⟩

lemma *fNext-geq-not-member*: $s \geq fNext\ S \Longrightarrow s\ |\notin|\ S$
⟨*proof*⟩

lemma *fNext-not-member*: $fNext\ S\ |\notin|\ S$
⟨*proof*⟩

lemma *fNext-not-member-subset*: $S\ |\subseteq|\ S' \Longrightarrow fNext\ S'\ |\notin|\ S$
⟨*proof*⟩

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

end

end

end

2.1 Fresh monad operations as class operations

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

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

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

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

global-interpretation *Fresh-Monad.fresh next default*
defines *fresh-create* = *create*
and *fresh-Next* = *Next*
and *fresh-fNext* = *fNext*
and *fresh-frun* = *frun-fresh*

and *fresh-run* = *run-fresh*
⟨*proof*⟩

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

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*›

⟨*proof*⟩

context
fixes *A* :: ‹*'a* :: *fresh set*›
assumes *finite*: ‹*finite A*›
begin

lemma *fresh-list-fresh*: ‹*set* (*fresh-list n A*) ∩ *A* = {}›
⟨*proof*⟩

lemma *fresh-list-fresh-elem*: ‹*x* ∈ *set* (*fresh-list n A*) ⟹ *x* ∉ *A*›
⟨*proof*⟩

lemma *fresh-list-distinct*: ‹*distinct* (*fresh-list n A*)›
⟨*proof*⟩

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-Utils
HOL−Library.Disjoint-FSets
begin

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

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

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

derive *linorder term*

3.2 A type class describing terms

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

instance
⟨*proof*⟩

end

context *pre-term* **begin**

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

lemma *freess-cons*[*simp*]: *freess* (*x* # *xs*) = *frees x* |∪| *freess xs*
⟨*proof*⟩

lemma *freess-single*: *freess* [*x*] = *frees x*
⟨*proof*⟩

lemma *freess-empty*[*simp*]: *freess* [] = {||}
⟨*proof*⟩

lemma *freess-app*[*simp*]: *freess* (*xs* @ *ys*) = *freess xs* |∪| *freess ys*
⟨*proof*⟩

lemma *freess-subset*: *set xs* ⊆ *set ys* ⟹ *freess xs* |⊆| *freess ys*
⟨*proof*⟩

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

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

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

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

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

lemma *app-size1*: *size* u_1 < *size* (*app* u_1 u_2)
⟨*proof*⟩

lemma *app-size2*: *size* u_2 < *size* (*app* u_1 u_2)
⟨*proof*⟩

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

unfree u = Some z ⟹ unapp u = None
⟨*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 ⟷ (*unconst t* ≠ *None*)

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

lemma *is-const-simps*[*simp*]:
is-const (const name)
¬ *is-const (app t u)*
¬ *is-const (free name)*
⟨*proof*⟩

lemma *const-name-simps*[*simp*]:
const-name (const name) = name
is-const t ⟹ const (const-name t) = t
⟨*proof*⟩

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

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

lemma *is-free-simps*[*simp*]:
is-free (free name)
¬ *is-free (const name)*
¬ *is-free (app t u)*
⟨*proof*⟩

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

free-name (*free name*) = *name*
is-free t ⟹ *free* (*free-name t*) = *t*
⟨*proof*⟩

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

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

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

lemma *app-simps*[*simp*]:
¬ *is-app* (*const name*)
¬ *is-app* (*free name*)
is-app (*app t u*)
⟨*proof*⟩

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

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

lemma *closed-except-const*[*simp*]: *closed-except* (*const name*) *S*
⟨*proof*⟩

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

lemma *closed-except-self*: *closed-except t* (*frees t*)
⟨*proof*⟩

end

class *term* = *pre-term* + *size* +
fixes
abs-pred :: (*'a* ⇒ *bool*) ⇒ *'a* ⇒ *bool*
assumes
raw-induct[*case-names const free app abs*]:
(⋀*name*. *P* (*const name*)) ⟹
(⋀*name*. *P* (*free name*)) ⟹
(⋀t_1 t_2. *P* t_1 ⟹ *P* t_2 ⟹ *P* (*app* t_1 t_2)) ⟹
(⋀*t*. *abs-pred P t*) ⟹
P t

assumes
raw-subst-id: *abs-pred* ($\lambda t.\ \forall\, env.$ *id-env env* $\longrightarrow$ *subst t env* $= t$) *t* **and**
raw-subst-drop: *abs-pred* ($\lambda t.\ x\ |\notin|$ *frees t* $\longrightarrow$ ($\forall\, env.$ *subst t* (*fmdrop x env*) $=$ *subst t env*)) *t* **and**
raw-subst-indep: *abs-pred* ($\lambda t.\ \forall\, env_1\ env_2.$ *closed-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) *t* **and**
raw-subst-frees: *abs-pred* ($\lambda t.\ \forall\, env.$ *closed-env env* $\longrightarrow$ *frees* (*subst t env*) $=$ *frees t* $|-|$ *fmdom env*) *t* **and**
raw-subst-consts′: *abs-pred* ($\lambda a.\ \forall\, x.$ *consts* (*subst a x*) $=$ *consts a* $|\cup|$ *ffUnion* (*consts* $|`|$ *fmimage x* (*frees a*))) *t* **and**
abs-pred-trivI: *P t* $\Longrightarrow$ *abs-pred P t*
begin

lemma *subst-id*: *id-env env* $\Longrightarrow$ *subst t env* $= t$
⟨*proof*⟩

lemma *subst-drop*: *x* $|\notin|$ *frees t* $\Longrightarrow$ *subst t* (*fmdrop x env*) $=$ *subst t env*
⟨*proof*⟩

lemma *subst-frees*: *fmpred* (λ-. *closed*) *env* $\Longrightarrow$ *frees* (*subst t env*) $=$ *frees t* $|-|$ *fmdom env*
⟨*proof*⟩

lemma *subst-consts′*: *consts* (*subst t env*) $=$ *consts t* $|\cup|$ *ffUnion* (*consts* $|`|$ *fmimage env* (*frees t*))
⟨*proof*⟩

fun *match* :: *term* $\Rightarrow$ *'a* $\Rightarrow$ (*name*, *'a*) *fmap option* **where**
match (t_1 \$ t_2) u = *do* {
  (u_1, u_2) $\leftarrow$ *unapp u*;
  env_1 $\leftarrow$ *match* t_1 u_1;
  env_2 $\leftarrow$ *match* t_2 u_2;
  Some (env_1 $++_f$ env_2)
} |
match (*Const name*) u =
  (*case unconst u of*
    None $\Rightarrow$ *None*
  | *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* {
    env_1 $\leftarrow$ *match* t_1 u_1;
    env_2 $\leftarrow$ *match* t_2 u_2;
    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 ⋀t_1 t_2 u_1 u_2 env_1 env_2. *P* t_1 u_1 env_1 ⟹ *match* t_1 u_1 = *Some* env_1 ⟹ *P* t_2 u_2 env_2 ⟹ *match* t_2 u_2 = *Some* env_2 ⟹ *P* (t_1 $ t_2) (*app* u_1 u_2) (env_1 ++$_f$ env_2)
assumes ⋀*name*. *P* (*Const name*) (*const name*) *fmempty*
assumes ⋀*name u*. *P* (*Free name*) *u* (*fmupd name u fmempty*)
shows *P t u env*
⟨*proof*⟩

lemma *match-dom*: *match p t = Some env* ⟹ *fmdom env = frees p*
⟨*proof*⟩

lemma *match-vars*: *match p t = Some env* ⟹ *fmpred* (λ- *u*. *frees u* |⊆| *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* |⊆| *S fmpred* (λ- *u*. *consts u* |⊆| *S*) *env*
shows *consts* (*subst t env*) |⊆| *S*
⟨*proof*⟩

lemma *subst-empty*[*simp*]: *subst t fmempty = t*
⟨*proof*⟩

lemma *subst-drop-fset*: *fdisjnt S* (*frees t*) ⟹ *subst t* (*fmdrop-fset S env*) = *subst t env*
⟨*proof*⟩

lemma *subst-restrict*:
assumes *frees t* |⊆| *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 $\bigwedge x.\ x\ |\in|\ \mathit{frees}\ t \Longrightarrow \mathit{fmlookup}\ \Gamma_1\ x = \mathit{fmlookup}\ \Gamma_2\ x$
shows $\mathit{subst}\ t\ \Gamma_1 = \mathit{subst}\ t\ \Gamma_2$
⟨*proof*⟩

corollary *subst-add-disjnt*:
assumes $\mathit{fdisjnt}\ (\mathit{frees}\ t)\ (\mathit{fmdom}\ \mathit{env}_1)$
shows $\mathit{subst}\ t\ (\mathit{env}_1 +\!\!+_f\ \mathit{env}_2) = \mathit{subst}\ t\ \mathit{env}_2$
⟨*proof*⟩

corollary *subst-add-shadowed-env*:
assumes $\mathit{frees}\ t\ |\subseteq|\ \mathit{fmdom}\ \mathit{env}_2$
shows $\mathit{subst}\ t\ (\mathit{env}_1 +\!\!+_f\ \mathit{env}_2) = \mathit{subst}\ t\ \mathit{env}_2$
⟨*proof*⟩

corollary *subst-restrict-closed*: $\mathit{closed\text{-}except}\ t\ S \Longrightarrow \mathit{subst}\ t\ (\mathit{fmrestrict\text{-}fset}\ S\ \mathit{env})$
$= \mathit{subst}\ t\ \mathit{env}$
⟨*proof*⟩

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

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

corollary *subst-closed-id*: $\mathit{closed}\ t \Longrightarrow \mathit{subst}\ t\ \mathit{env} = t$
⟨*proof*⟩

corollary *subst-closed-preserved*: $\mathit{closed}\ t \Longrightarrow \mathit{closed}\ (\mathit{subst}\ t\ \mathit{env})$
⟨*proof*⟩

context begin

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

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

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

shows *subst t* $(\Gamma' +\!\!+_f \Gamma)$ = *subst* (*subst t* Γ') Γ
⟨*proof*⟩

lemma *subst-twice*:
assumes $\Gamma' \subseteq_f \Gamma$ *closed-env* Γ'
shows *subst* (*subst t* Γ') Γ = *subst t* Γ
⟨*proof*⟩

end

fun *matchs* :: *term list* ⇒ *'a list* ⇒ (*name*, *'a*) *fmap option* **where**
matchs [] [] = *Some fmempty* |
matchs (*t* # *ts*) (*u* # *us*) = *do* { env_1 ← *match t u*; env_2 ← *matchs ts us*; *Some* (env_1 $+\!\!+_f$ env_2) } |
matchs - - = *None*

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

context begin

private lemma *matchs-alt-def0*:
assumes *length ps* = *length vs*
shows *map-option* (λ*env*. *m* $+\!\!+_f$ *env*) (*matchs ps vs*) = *map-option* (*foldl* ($+\!\!+_f$) *m*) (*those* (*map2 match ps vs*))
⟨*proof*⟩

lemma *matchs-alt-def*:
assumes *length ps* = *length vs*
shows *matchs ps vs* = *map-option* (*foldl* ($+\!\!+_f$) *fmempty*) (*those* (*map2 match ps vs*))
⟨*proof*⟩

end

lemma *matchs-neq-length-none*[*simp*]: *length xs* ≠ *length ys* ⟹ *matchs xs ys* = *None*
⟨*proof*⟩

corollary *matchs-some-eq-length*: *matchs xs ys* = *Some env* ⟹ *length xs* = *length ys*
⟨*proof*⟩

lemma *matchs-app*[*simp*]:
assumes *length* xs_2 = *length* ys_2
shows *matchs* (xs_1 @ xs_2) (ys_1 @ ys_2) =
matchs xs_1 ys_1 ⤜ (λenv_1. *matchs* xs_2 ys_2 ⤜ (λenv_2. *Some* (env_1 $+\!\!+_f$ env_2)))
⟨*proof*⟩

corollary *matchs-appI*:
assumes *matchs xs ys* = *Some env*$_1$ *matchs xs' ys'* = *Some env*$_2$
shows *matchs* (*xs* @ *xs'*) (*ys* @ *ys'*) = *Some* (*env*$_1$ ++$_f$ *env*$_2$)
⟨*proof*⟩

corollary *matchs-dom*:
assumes *matchs ps ts* = *Some env*
shows *fmdom env* = *freess ps*
⟨*proof*⟩

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

lemma *find-match-map*:
find-match (*map* (λ(*pat*, *t*). (*pat*, *f pat t*)) *cs*) *t* =
map-option (λ(*env*, *pat*, *rhs*). (*env*, *pat*, *f pat rhs*)) (*find-match cs t*)
⟨*proof*⟩

lemma *find-match-elem*:
assumes *find-match cs t* = *Some* (*env*, *pat*, *rhs*)
shows (*pat*, *rhs*) ∈ *set cs match pat t* = *Some env*
⟨*proof*⟩

lemma *match-subst-closed*:
assumes *match pat t* = *Some env closed-except rhs* (*frees pat*) *closed t*
shows *closed* (*subst rhs env*)
⟨*proof*⟩

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

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

lemma *rewrite-step-closed*:
assumes *frees t*$_2$ |⊆| *frees t*$_1$ (*t*$_1$, *t*$_2$) ⊢ *u* → *u'* *closed u*
shows *closed u'*
⟨*proof*⟩

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

lemma *matchesI*[*intro*]: *subst t env* = *u* ⟹ *t* ≲ *u*

⟨*proof*⟩

lemma *matchesE*[*elim*]:
assumes $t \lesssim u$
obtains *env* **where** *subst t env* = *u*
⟨*proof*⟩

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

lemma *overlapping-refl*: *overlapping t t*
⟨*proof*⟩

lemma *overlapping-sym*: *overlapping t u* $\Longrightarrow$ *overlapping u t*
⟨*proof*⟩

lemma *overlappingI*[*intro*]: $s \lesssim u \Longrightarrow t \lesssim u \Longrightarrow$ *overlapping s t*
⟨*proof*⟩

lemma *overlappingE*[*elim*]:
assumes *overlapping s t*
obtains *u* **where** $s \lesssim u$ $t \lesssim u$
⟨*proof*⟩

abbreviation *non-overlapping s t* ≡ ¬ *overlapping s t*

corollary *non-overlapping-implies-neq*: *non-overlapping t u* $\Longrightarrow$ $t \neq u$
⟨*proof*⟩

end

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

code-pred (*modes*: *i* ⇒ *i* ⇒ *o* ⇒ *bool*) *rewrite-first* ⟨*proof*⟩

lemma *rewrite-firstE*:
assumes *rewrite-first cs t t'*
obtains *pat rhs env* **where** (*pat*, *rhs*) ∈ *set cs match pat t* = *Some env t'* = *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 *find-match cs t* = *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 (⋀u_1 u_2. *t* = *app* u_1 u_2 ⟹ *if-app1* u_1 u_2 = *if-app2* u_1 u_2)
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

⟨*proof*⟩

lemma *strip-comb-simps*[*simp*]:
strip-comb (*app t u*) = (*let* (*f*, *args*) = *strip-comb t in* (*f*, *args* @ [*u*]))
unapp t = *None* ⟹ *strip-comb t* = (*t*, [])
⟨*proof*⟩

lemma *strip-comb-induct*[*case-names app no-app*]:
assumes ⋀*x y*. *P x* ⟹ *P* (*app x y*)
assumes ⋀*t*. *unapp t* = *None* ⟹ *P t*
shows *P t*
⟨*proof*⟩

lemma *strip-comb-size*: *t'* ∈ *set* (*snd* (*strip-comb t*)) ⟹ *size t'* < *size t*
⟨*proof*⟩

lemma *sstrip-comb-termination*[*termination-simp*]:
(*f*, *ts*) = *strip-comb t* ⟹ *t'* ∈ *set ts* ⟹ *size t'* < *size t*
⟨*proof*⟩

lemma *strip-comb-empty*: *snd* (*strip-comb t*) = [] ⟹ *fst* (*strip-comb t*) = *t*
⟨*proof*⟩

lemma *strip-comb-app*: *fst* (*strip-comb* (*app t u*)) = *fst* (*strip-comb t*)
⟨*proof*⟩

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

lemma *list-comb-app*[*simp*]: *list-comb f* (*xs* @ *ys*) = *list-comb* (*list-comb f xs*) *ys*
⟨*proof*⟩

corollary *list-comb-snoc*: *app* (*list-comb f xs*) *y* = *list-comb f* (*xs* @ [*y*])
⟨*proof*⟩

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

lemma *subst-list-comb*: *subst* (*list-comb f xs*) *env* = *list-comb* (*subst f env*) (*map* (λ*t*. *subst t env*) *xs*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *strip-list-comb*: *strip-comb* (*list-comb f ys*) = (*fst* (*strip-comb f*), *snd* (*strip-comb f*) @ *ys*)
⟨*proof*⟩

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

lemma *frees-list-comb*[*simp*]: *frees* (*list-comb t xs*) = *frees t* |∪| *freess xs*
⟨*proof*⟩

lemma *consts-list-comb*: *consts* (*list-comb f xs*) = *consts f* |∪| *ffUnion* (*fset-of-list* (*map consts xs*))
⟨*proof*⟩

lemma *ids-list-comb*: *ids* (*list-comb f xs*) = *ids f* |∪| *ffUnion* (*fset-of-list* (*map ids xs*))
⟨*proof*⟩

lemma *frees-strip-comb*: *frees t* = *frees* (*fst* (*strip-comb t*)) |∪| *freess* (*snd* (*strip-comb t*))
⟨*proof*⟩

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

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*
⟨*proof*⟩

end

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

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

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

lemma *list-comb-cond-inj*:
  assumes *list-comb f xs* = *list-comb g ys left-nesting f* = *left-nesting g*

shows $xs = ys\ f = g$
⟨*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*) (λ*t u*. *no-abs t* ∧ *no-abs u*)
False t

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

lemma *no-abs-induct*[*consumes 1*, *case-names free const app*, *induct pred*: *no-abs*]:
assumes *no-abs t*
assumes ⋀*name*. *P* (*free name*)
assumes ⋀*name*. *P* (*const name*)
assumes ⋀$t_1\ t_2$. *P* t_1 ⟹ *no-abs* t_1 ⟹ *P* t_2 ⟹ *no-abs* t_2 ⟹ *P* (*app* $t_1\ t_2$)
shows *P t*
⟨*proof*⟩

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*) $t_1\ t_2$ **where** *t* = *app* $t_1\ t_2$ *no-abs* t_1 *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*::*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*) $\longleftrightarrow$ P f $\wedge$ *list-all* P *xs*
⟨*proof*⟩

corollary *list-combE*:
  assumes P (*list-comb* f *xs*)
  shows P f $x \in$ *set xs* $\Longrightarrow$ P x
⟨*proof*⟩

lemma *match*:
  assumes *match pat t* = *Some env P t*
  shows *fmpred* (λ-. *P*) *env*
⟨*proof*⟩

lemma *matchs*:
  assumes *matchs pats ts* = *Some env list-all P ts*
  shows *fmpred* (λ-. *P*) *env*
⟨*proof*⟩

end

end

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

lemma *rewrite-step*:
  assumes (*lhs*, *rhs*) ⊢ *t* → *t*′ *P t P rhs*
  shows P *t*′
⟨*proof*⟩

end

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

context
  notes *app*[*simp*]
begin

lemma *list-comb*: P (*list-comb* f *xs*) $\longleftrightarrow$ P f ∨ *list-ex* P *xs*
⟨*proof*⟩

lemma *match*:
  assumes *match pat t* = *Some env* ¬ *P t*
  shows *fmpred* (λ- *t*. ¬ *P t*) *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* $\lambda t.$ *closed-except* $t\ S$ **for** S
⟨*proof*⟩

global-interpretation *closed*: *subst-syntactic-and closed*
⟨*proof*⟩

corollary *closed-list-comb*: *closed* (*name* $\$\$$ *args*) $\longleftrightarrow$ *list-all closed args*
⟨*proof*⟩

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

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

lemma *related-match*:
  assumes *match x u* = *Some env P t u*
  obtains *env'* **where** *match x t* = *Some env' P-env env' env*
⟨*proof*⟩

lemma *list-combI*:
  assumes *list-all2 P* $us_1\ us_2\ P\ t_1\ t_2$
  shows P (*list-comb* $t_1\ us_1$) (*list-comb* $t_2\ us_2$)
⟨*proof*⟩

lemma *list-combE*:
  assumes $P\ t$ (*name* $\$\$$ *args*)
  obtains *args'* **where** t = *name* $\$\$$ *args' list-all2 P args' args*
⟨*proof*⟩

end

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

lemma *unconst-rel*: $P\ t\ u \Longrightarrow unconst\ t = unconst\ u$
⟨*proof*⟩

lemma *unapp-rel*: $P\ t\ u \Longrightarrow rel\text{-}option\ (rel\text{-}prod\ P\ P)\ (unapp\ t)\ (unapp\ u)$
⟨*proof*⟩

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

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

end

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

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

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

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

lemma *convert-term-inj*:
  assumes *no-abs t no-abs t' convert-term t* = *convert-term t'*
  shows $t = t'$
⟨*proof*⟩

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

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

instance ⟨*proof*⟩

end

lemma *is-const-free*[*simp*]: ¬ *is-const* (*Free name*)
⟨*proof*⟩

lemma *is-free-app*[*simp*]: ¬ *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*) ⟹ *is-free f*
⟨*proof*⟩

lemma *const-list-comb-free*[*simp*]: ¬ *is-free* (*name* $$ *args*)
⟨*proof*⟩

corollary *const-list-comb-neq-free*[*simp*]: *name* $$ *args* ≠ *free name'*
⟨*proof*⟩

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

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

lemma *matchs-match-list-comb*[*simp*]: *match* (*name* $$ *xs*) (*name* $$ *ys*) = *matchs xs ys*
⟨*proof*⟩

fun *bounds* :: *term* ⇒ *nat fset* **where**
bounds (*Bound i*) = {| *i* |} |
bounds (t_1 $ t_2) = *bounds* t_1 |∪| *bounds* t_2 |
bounds (Λ *t*) = (λ*i*. *i* − *1*) |‘| (*bounds t* − {| *0* |}) |
bounds - = {||}

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

fun *incr-bounds* :: *int* ⇒ *nat* ⇒ *term* ⇒ *term* **where**
incr-bounds inc lev (*Bound i*) = (*if i* ≥ *lev then Bound* (*shift-nat i inc*) *else Bound i*) |
incr-bounds inc lev (Λ *u*) = Λ *incr-bounds inc* (*lev* + *1*) *u* |
incr-bounds inc lev (t_1 $ t_2) = *incr-bounds inc lev* t_1 $ *incr-bounds inc lev* t_2 |
incr-bounds - - *t* = *t*

lemma *incr-bounds-frees*[*simp*]: *frees* (*incr-bounds n k t*) = *frees t*
⟨*proof*⟩

lemma *incr-bounds-zero*[*simp*]: *incr-bounds 0 i t* = *t*
⟨*proof*⟩

fun *replace-bound* :: *nat* ⇒ *term* ⇒ *term* ⇒ *term* **where**
replace-bound lev (*Bound i*) *t* = (*if i* < *lev then Bound i else if i* = *lev then incr-bounds* (*int lev*) *0 t else Bound* (*i* − *1*)) |
replace-bound lev (t_1 $ t_2) *t* = *replace-bound lev* t_1 *t* $ *replace-bound lev* t_2 *t* |
replace-bound lev (Λ *u*) *t* = Λ *replace-bound* (*lev* + *1*) *u t* |
replace-bound - *t* - = *t*

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

lemma *replace-bound-frees*: *frees* (*replace-bound n t t′*) |⊆| *frees t* |∪| *frees t′*
⟨*proof*⟩

lemma *replace-bound-eq*:
assumes $i \mid\notin\mid$ *bounds* t
shows *replace-bound* $i\ t\ t' =$ *incr-bounds* $(-1)\ (i + 1)\ t$
⟨*proof*⟩

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

lemma *wellformed-inc*:
assumes *wellformed'* $k\ t\ k \leq n$
shows *wellformed'* $n\ t$
⟨*proof*⟩

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

lemma *wellformed'-replace-bound-eq*:
assumes *wellformed'* $n\ t\ k \geq n$
shows *replace-bound* $k\ t\ u = t$
⟨*proof*⟩

lemma *wellformed-replace-bound-eq*: *wellformed* $t \Longrightarrow$ *replace-bound* $k\ t\ u = t$
⟨*proof*⟩

lemma *incr-bounds-eq*: $n \geq k \Longrightarrow$ *wellformed'* $k\ t \Longrightarrow$ *incr-bounds* $i\ n\ t = t$
⟨*proof*⟩

lemma *incr-bounds-subst*:
assumes $\bigwedge t.\ t \in$ *fmran'* $env \Longrightarrow$ *wellformed* t
shows *incr-bounds* $i\ n$ (*subst* $t\ env$) = *subst* (*incr-bounds* $i\ n\ t$) env
⟨*proof*⟩

lemma *incr-bounds-wellformed*:
assumes *wellformed'* $m\ u$
shows *wellformed'* $(k + m)$ (*incr-bounds* (*int* k) $n\ u$)
⟨*proof*⟩

lemma *replace-bound-wellformed*:
assumes *wellformed* u *wellformed'* (*Suc* k) $t\ i \leq k$
shows *wellformed'* k (*replace-bound* $i\ t\ u$)
⟨*proof*⟩

lemma *subst-wellformed*:
assumes *wellformed'* $n\ t$ *fmpred* (λ-. *wellformed*) env
shows *wellformed'* n (*subst* $t\ env$)

⟨*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-matchs*:
  assumes *matchs ps* ts_2 = *Some* env_2 *list-all2 P* ts_1 ts_2

obtains env_1 **where** *matchs ps* ts_1 = *Some* env_1 *P-env* env_1 env_2
⟨*proof*⟩

end

end

Chapter 4

Wellformedness of patterns

theory *Pats*
imports *Term*
begin

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

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

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

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

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

lemma *linear-implies-no-abs*: *linear t* ⟹ *no-abs t*
⟨*proof*⟩

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

lemma *linears-butlastI*[*intro*]: *linears ts* $\Longrightarrow$ *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* $\Longrightarrow$ *t* $\in$ *set ts* $\Longrightarrow$ *linear t*
⟨*proof*⟩

lemma *linears-singleI*[*intro*]: *linear t* $\Longrightarrow$ *linears* [*t*]
⟨*proof*⟩

lemma *linear-strip-comb*: *linear t* $\Longrightarrow$ *linear* (*fst* (*strip-comb t*))
⟨*proof*⟩

lemma *linears-strip-comb*: *linear t* $\Longrightarrow$ *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*) ¬ *is-free f*
  shows *linear* (*list-comb f xs*)
⟨*proof*⟩

corollary *linear-list-comb′*: *linears xs* $\Longrightarrow$ *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* $\Longrightarrow$ *wellformed′ n t*
⟨*proof*⟩

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

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

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

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

context begin

private lemma *match-subst-correctness0*:
assumes *linear t*
shows *case match t u of*
None ⇒ (∀ *env*. *subst* (*convert-term t*) *env* ≠ *u*) |
Some env ⇒ *subst* (*convert-term t*) *env* = *u*
⟨*proof*⟩

lemma *match-subst-some*[*simp*]:
match t u = *Some env* ⟹ *linear t* ⟹ *subst* (*convert-term t*) *env* = *u*
⟨*proof*⟩

lemma *match-subst-none*:
match t u = *None* ⟹ *linear t* ⟹ *subst* (*convert-term t*) *env* = *u* ⟹ *False*
⟨*proof*⟩

end

lemma *match-matches*: *match t u* = *Some env* ⟹ *linear t* ⟹ *t* ≲ *u*
⟨*proof*⟩

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

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

lemma *non-overlapping-appI1*: *non-overlapping* t_1 u_1 ⟹ *non-overlapping* (t_1 $ t_2) (u_1 $ u_2)
⟨*proof*⟩

lemma *non-overlapping-appI2*: *non-overlapping* t_2 u_2 ⟹ *non-overlapping* (t_1 $ t_2) (u_1 $ u_2)

⟨*proof*⟩

lemma *non-overlapping-app-constI*: *non-overlapping* (t_1 \$ t_2) (*Const name*)
⟨*proof*⟩

lemma *non-overlapping-const-appI*: *non-overlapping* (*Const name*) (t_1 \$ t_2)
⟨*proof*⟩

lemma *non-overlapping-const-constI*: $x \neq y \Longrightarrow$ *non-overlapping* (*Const x*) (*Const y*)
⟨*proof*⟩

lemma *match-overlapping*:
  assumes *linear* t_1 *linear* t_2
  assumes *match* t_1 u = *Some* env_1 *match* t_2 u = *Some* env_2
  shows *overlapping* t_1 t_2
⟨*proof*⟩

end

Chapter 5

Terms with explicit bound variable names

theory *Nterm*
imports *Term-Class*
begin

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

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

derive *linorder nterm*

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

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

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

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

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

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

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

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

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

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

instance
⟨*proof*⟩

end

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

definition *abs-pred-nterm* :: (*nterm* ⇒ *bool*) ⇒ *nterm* ⇒ *bool* **where**
[*code del*]: *abs-pred P t* ⟷ (∀ *t′ x*. *t* = (Λ_n *x*. *t′*) ⟶ *P t′* ⟶ *P t*)

instance ⟨*proof*⟩

end

lemma *no-abs-abs*[*simp*]: ¬ *no-abs* (Λ_n *x*. *t*)
⟨*proof*⟩

end

Chapter 6

Converting between *term*s and *nterm*s

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

6.1 α-equivalence

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

code-pred *alpha-equiv* ⟨*proof*⟩

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

lemma *alpha-equiv-refl*[*intro?*]:
assumes *fmpred* (=) Γ
shows *alpha-equiv* Γ *t t*
⟨*proof*⟩

corollary *alpha-eq-refl*: *alpha-eq t t*

⟨*proof*⟩

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

fun *term-to-nterm* :: *name list* ⇒ *term* ⇒ (*name*, *nterm*) *state* **where**
term-to-nterm - (Const name) = *State-Monad.return (Nconst name)* |
term-to-nterm - (Free name) = *State-Monad.return (Nvar name)* |
term-to-nterm Γ *(Bound n)* = *State-Monad.return (Nvar (*Γ ! *n))* |
term-to-nterm Γ *(*Λ *t)* = *do* {
  n ← *fresh-create*;
  e ← *term-to-nterm (n #* Γ*) t*;
  *State-Monad.return (*Λ$_n$ *n. e)*
} |
term-to-nterm Γ *(*t_1 $ t_2*)* = *do* {
  e_1 ← *term-to-nterm* Γ t_1;
  e_2 ← *term-to-nterm* Γ t_2;
  State-Monad.return (e_1 $_n e_2*)*
}

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

lemma *term-to-nterm*:
  assumes *no-abs t*
  shows *fst (run-state (term-to-nterm* Γ *t) x)* = *convert-term t*
⟨*proof*⟩

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

lemma *term-to-nterm-mono*: *mono-state (term-to-nterm* Γ *x)*
⟨*proof*⟩

lemma *term-to-nterm-vars0*:
  assumes *wellformed' (length* Γ*) t*
  shows *frees (fst (run-state (term-to-nterm* Γ *t) s)) |*⊆*| frees t |*∪*| fset-of-list* Γ
⟨*proof*⟩
    including *fset.lifting* ⟨*proof*⟩

corollary *term-to-nterm-vars*:
  assumes *wellformed t*
  shows *frees (fresh-frun (term-to-nterm* [] *t) F) |*⊆*| frees t*
⟨*proof*⟩

corollary *term-to-nterm-closed*: *closed t* ⟹ *wellformed t* ⟹ *closed (term-to-nterm' t)*
⟨*proof*⟩

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

Γ *t*)
⟨*proof*⟩

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

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

lemma *nterm-to-term*:
assumes *no-abs t fdisjnt* (*fset-of-list* Γ) (*frees t*)
shows *nterm-to-term* Γ *t* = *convert-term t*
⟨*proof*⟩

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

lemma *nterm-to-term′*: *no-abs t* ⟹ *nterm-to-term′ t* = *convert-term t*
⟨*proof*⟩

lemma *nterm-to-term-frees*[*simp*]: *frees* (*nterm-to-term* Γ *t*) = *frees t* − *fset-of-list* Γ
⟨*proof*⟩
including *fset.lifting*
⟨*proof*⟩
including *fset.lifting*
⟨*proof*⟩

6.4 Correctness

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

lemma *term-to-nterm-nterm-to-term0*:
assumes *wellformed′* (*length* Γ) *t fdisjnt* (*fset-of-list* Γ) (*frees t*) *distinct* Γ
assumes *fBall* (*frees t* |∪| *fset-of-list* Γ) (λ*x*. *x* ≤ *s*)
shows *nterm-to-term* Γ (*fst* (*run-state* (*term-to-nterm* Γ *t*) *s*)) = *t*
⟨*proof*⟩
including *fset.lifting* ⟨*proof*⟩

lemma *term-to-nterm-nterm-to-term*:
assumes *wellformed t frees t* |⊆| *S*
shows *nterm-to-term′* (*frun-fresh* (*term-to-nterm* [] *t*) (*S* |∪| *Q*)) = *t*
⟨*proof*⟩

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

shows *nterm-to-term′* (*term-to-nterm′ t*) = *t*
⟨*proof*⟩

lemma *nterm-to-term-eq*:
assumes *frees u* |⊆| *fset-of-list* (*common-prefix* Γ Γ′)
shows *nterm-to-term* Γ *u* = *nterm-to-term* Γ′ *u*
⟨*proof*⟩
including *fset.lifting*
⟨*proof*⟩

corollary *nterm-to-term-eq-closed*: *closed t* ⟹ *nterm-to-term* Γ *t* = *nterm-to-term* Γ′ *t*
⟨*proof*⟩

lemma *nterm-to-term-wellformed*: *wellformed′* (*length* Γ) (*nterm-to-term* Γ *t*)
⟨*proof*⟩

corollary *nterm-to-term-closed-wellformed*: *closed t* ⟹ *wellformed* (*nterm-to-term* Γ *t*)
⟨*proof*⟩

lemma *nterm-to-term-term-to-nterm*:
assumes *frees t* |⊆| *fset-of-list* Γ *length* Γ = *length* Γ′
shows *alpha-equiv* (*fmap-of-list* (*zip* Γ Γ′)) *t* (*fst* (*run-state* (*term-to-nterm* Γ′ (*nterm-to-term* Γ *t*)) *s*))
⟨*proof*⟩

corollary *nterm-to-term-term-to-nterm′*: *closed t* ⟹ *t* ≈$_\alpha$ *term-to-nterm′* (*nterm-to-term′ t*)
⟨*proof*⟩

context begin

private lemma *term-to-nterm-alpha-equiv0*:
length Γ*1* = *length* Γ*2* ⟹ *distinct* Γ*1* ⟹ *distinct* Γ*2* ⟹ *wellformed′* (*length* Γ*1*) *t1* ⟹
fresh-fin (*frees t1* |∪| *fset-of-list* Γ*1*) *s1* ⟹ *fdisjnt* (*fset-of-list* Γ*1*) (*frees t1*) ⟹
fresh-fin (*frees t1* |∪| *fset-of-list* Γ*2*) *s2* ⟹ *fdisjnt* (*fset-of-list* Γ*2*) (*frees t1*) ⟹
alpha-equiv (*fmap-of-list* (*zip* Γ*1* Γ*2*)) (*fst*(*run-state* (*term-to-nterm* Γ*1 t1*) *s1*)) (*fst* (*run-state* (*term-to-nterm* Γ*2 t1*) *s2*))
⟨*proof*⟩

lemma *term-to-nterm-alpha-equiv*:
assumes *length* Γ*1* = *length* Γ*2 distinct* Γ*1 distinct* Γ*2 closed t*
assumes *wellformed′* (*length* Γ*1*) *t*
assumes *fresh-fin* (*fset-of-list* Γ*1*) *s1 fresh-fin* (*fset-of-list* Γ*2*) *s2*
shows *alpha-equiv* (*fmap-of-list* (*zip* Γ*1* Γ*2*)) (*fst* (*run-state* (*term-to-nterm* Γ*1*

t) *s1*)) (*fst* (*run-state* (*term-to-nterm* Γ*2* *t*) *s2*))
— An instantiated version of this lemma with Γ*1* = [] and Γ*2* = [] would not make sense because then it would just be a special case of *alpha-eq-refl*.
⟨*proof*⟩

end

global-interpretation *nrelated*: *term-struct-rel-strong* (λ*t* *n*. *t* = *nterm-to-term* Γ *n*) **for** Γ
⟨*proof*⟩

lemma *env-nrelated-closed*:
assumes *nrelated.P-env* Γ *env nenv closed-env nenv*
shows *nrelated.P-env* Γ′ *env nenv*
⟨*proof*⟩

lemma *nrelated-subst*:
assumes *nrelated.P-env* Γ *env nenv closed-env nenv fdisjnt* (*fset-of-list* Γ) (*fmdom* *nenv*)
shows *subst* (*nterm-to-term* Γ *t*) *env* = *nterm-to-term* Γ (*subst* *t* *nenv*)
⟨*proof*⟩
including *fset.lifting* ⟨*proof*⟩

lemma *nterm-to-term-insert-dupl*:
assumes *y* ∈ *set* (*take* *n* Γ) *n* ≤ *length* Γ
shows *nterm-to-term* Γ *t* = *incr-bounds* (− *1*) (*Suc* *n*) (*nterm-to-term* (*insert-nth* *n* *y* Γ) *t*)
⟨*proof*⟩

lemma *nterm-to-term-bounds-dupl*:
assumes *i* < *length* Γ *j* < *length* Γ *i* < *j*
assumes Γ ! *i* = Γ ! *j*
shows *j* |∉| *bounds* (*nterm-to-term* Γ *t*)
⟨*proof*⟩

fun *subst-single* :: *nterm* ⇒ *name* ⇒ *nterm* ⇒ *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-single* *t*$_1$ *s′* *t′* $_n$ *subst-single* *t*$_2$ *s′* *t′* |
subst-single (Λ$_n$ *x*. *t*) *s′* *t′* = (Λ$_n$ *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-of-list* [(*s*, *t′*)])
⟨*proof*⟩

lemma *nterm-to-term-subst-replace-bound*:
assumes *closed* *u′* *n* ≤ *length* Γ *x* ∉ *set* (*take* *n* Γ)
shows *nterm-to-term* Γ (*subst-single* *u* *x* *u′*) = *replace-bound* *n* (*nterm-to-term* (*insert-nth* *n* *x* Γ) *u*) (*nterm-to-term* Γ *u′*)
⟨*proof*⟩

corollary *nterm-to-term-subst-β*:
assumes *closed u′*
shows *nterm-to-term* Γ (*subst u* (*fmap-of-list* [(*x*, *u′*)])) = *nterm-to-term* (*x* # Γ) *u* [*nterm-to-term* Γ *u′*]$_\beta$
⟨*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* ⇒ *'a*
assumes *bij*: *bij of-name*
begin

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

lemma *to-of-name*[*simp*]: *to-name* (*of-name a*) = *a*
⟨*proof*⟩

lemma *of-to-name*[*simp*]: *of-name* (*to-name a*) = *a*
⟨*proof*⟩

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

⟨*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** λ*t*. *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 fmlookup env* (*to-name v*) *of Some v'* ⇒ *v'* | - ⇒ *Var v*) |
subst-trm (*trm.Const c*) - = *trm.Const c* |
subst-trm (*M* · *N*) *env* = *subst-trm M env* · *subst-trm N env*

instance
⟨*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* ⇒ *Some* (*t*, *u*) | - ⇒ *None*)

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

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

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

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

context
  includes *fset.lifting*
begin

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

⟨*proof*⟩

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