A Verified Code Generator from Isabelle/HOL to CakeML

Lars Hupel

July 11, 2019
## Contents

1 **Terms**  
   1.1 Additional material over the *Higher-Order-Terms* AFP entry  
   1.2 Reflecting HOL datatype definitions  
   1.3 Constructor information  
   1.4 Special constants  
   1.5 Term algebra extended with wellformedness  
   1.6 Terms with sequential pattern matching  
   1.7 Terms with explicit pattern matching  
   1.8 Irreducible terms (values)  
   1.9 Viewing *term* as values  
   1.10 A dedicated value type  

2 **A smaller version of CakeML: CupCakeML**  
   2.1 CupCake environments  
   2.2 CupCake semantics  

3 **Term rewriting**  
   3.1 Higher-order term rewriting using de-Bruijn indices  
   3.1.1 Matching and rewriting  
   3.1.2 Wellformedness  
   3.2 Higher-order term rewriting using explicit bound variable names  
   3.2.1 Definitions  
   3.2.2 Matching and rewriting  
   3.2.3 Translation from *Term-Class.term* to *nterm*  
   3.2.4 Correctness of translation  
   3.2.5 Completeness of translation  
   3.2.6 Splitting into constants  
   3.2.7 Computability  
   3.3 Higher-order term rewriting with explicit pattern matching  
   3.3.1 Intermediate rule sets  
   3.3.2 Pure pattern matching rule sets  
   3.4 Sequential pattern matching  
   3.5 Big-step semantics
Chapter 1

Terms

theory Doc-Terms
imports Main
begin

end

1.1 Additional material over the Higher-Order-Terms AFP entry

theory Terms-Extras
imports
  ../Utils/Compiler-Utils
  Higher-Order-Terms.Pats
  Dict-Construction.Dict-Construction
begin

no-notation Mpat-Antiquot.mpaq-App (infixl $\infty$ 900)

ML-file hol-term.ML

primrec basic-rule :: - $\Rightarrow$ bool where
  basic-rule (lhs, rhs) $\Longleftrightarrow$
  linear lhs $\land$
  is-const (fst (strip-comb lhs)) $\land$
  $\neg$ is-const lhs $\land$
  frees rhs $\subseteq$ frees lhs

lemma basic-ruleI[intro]:
  assumes linear lhs
  assumes is-const (fst (strip-comb lhs))
  assumes $\neg$ is-const lhs
  assumes frees rhs $\subseteq$ frees lhs
  shows basic-rule (lhs, rhs)
using assms by simp

primrec split-rule :: \((\text{term} \times \text{a}) \Rightarrow (\text{name} \times (\text{term list} \times \text{a}))\) where
split-rule \((\text{lhs}, \text{rhs}) = (\text{let (name, \text{args}} = \text{strip-comb} \ \text{lhs} \ \text{in (const-name name,} \ \text{(args, rhs))})\)

fun unsplit-rule :: \((\text{name} \times (\text{term list} \times \text{a})) \Rightarrow (\text{term} \times \text{a})\) where
unsplit-rule \((\text{name, (args, rhs}}) = (\text{name} \text{## args, rhs})\)

lemma split-unsplit: \(\text{split-rule (unsplit-rule t)} = t\)
by (induct t rule: unsplit-rule.induct) (simp add: strip-list-comb const-name-def)

lemma unsplit-split:
assumes basic-rule \(r\)
shows unsplit-rule \((\text{split-rule r)} = r\)
using assms
by (cases r) (simp add: split-beta)

datatype \(\text{pat} = \text{Patvar name} | \text{Patconstr name pat list}\)

fun mk-pat :: \(\text{term} \Rightarrow \text{pat}\) where
\(\text{mk-pat pat} = (\text{case strip-comb pat of (Const s, args} = \text{Patconstr s (map mk-pat args) | (Free s, [])} = \text{Patvar s)})\)

declare mk-pat.simps[simp del]

lemma mk-pat-simps[simp];
\(\text{mk-pat (name ## args) = Patconstr name (map mk-pat args)}\)
\(\text{mk-pat (Free name)} = \text{Patvar name}\)
apply (auto simp: mk-pat.simps strip-list-comb-const)
apply (simp add: const-term-def)
done

primrec patvars :: \(\text{pat} \Rightarrow \text{name fset}\) where
\(\text{patvars (Patvar name)} = \{ | \text{name} | \} | \text{patvars (Patconstr - ps)} = \text{ffUnion (fset-of-list (map patvars ps))}\)

lemma mk-pat-frees:
assumes linear \(p\)
shows \(\text{patvars (mk-pat p)} = \text{frees p}\)
using assms proof (induction \(p\) rule: linear-pat-induct)
case (comb name args)

have \(\text{map (patvars o mk-pat) args = map frees args}\)
using comb by force

hence \(\text{fset-of-list (map (patvars o mk-pat) args)} = \text{fset-of-list (map frees args)}\)
by metis
thus \(?case\)
by (simp add: freess-def)

5
This definition might seem a little counter-intuitive. Assume we have two defining equations of a function, e.g. map: map f [] = [] map f (x # xs) = f x # map f xs The pattern "matrix" is compiled right-to-left. Equal patterns are grouped together. This definition is needed to avoid the following situation: map f [] = [] map g (x # xs) = g x # map g xs While this is logically the same as above, the problem is that f and g are overlapping but distinct patterns. Hence, instead of grouping them together, they stay separate. This leads to overlapping patterns in the target language which will produce wrong results. One way to deal with this is to rename problematic variables before invoking the compiler.

fun pattern-compatible :: term ⇒ term ⇒ bool where
pattern-compatible (t₁ § t₂) (u₁ § u₂) ←→ pattern-compatible t₁ u₁ ∧ (t₁ = u₁ → pattern-compatible t₂ u₂) |
pattern-compatible t u ←→ t = u ∨ non-overlapping t u

lemmas pattern-compatible-simps[simp] =
pattern-compatible.simps[folded app-term-def]

lemmas pattern-compatible-induct = pattern-compatible.induct[case-names app-app]

lemma pattern-compatible-refl[intro!?]: pattern-compatible t t
by (induct t) auto

corollary pattern-compatible-reflP[intro!]: reflp pattern-compatible
by (auto intro: pattern-compatible-refl reflpI)

lemma pattern-compatible-cases[consumes 1]:
assumes pattern-compatible t u
obtains (eq) t = u
| (non-overlapping) non-overlapping t u
using assms proof (induction arbitrary: thesis rule: pattern-compatible-induct)
case (app-app t₁ t₂ u₁ u₂)

show ?case
proof (cases t₁ = u₁ ∧ t₂ = u₂)
  case True
  with app-app show thesis
  by simp
next
  case False
  from app-app have pattern-compatible t₁ t₁ t₁ = u₁ ⇒ pattern-compatible t₂ u₂
  by auto
  with False have non-overlapping (t₁ § t₂) (u₁ § u₂)
  using app-app by (metis non-overlapping-appI1 non-overlapping-appI2)
thus thesis
by (rule app-app.prems(2))
qed
qed auto

inductive rev-accum-rel :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ bool for R
where
  nil: rev-accum-rel R [] [] |
  snoc: rev-accum-rel R xs ys ⇒ (xs = ys ⇒ R x y) ⇒ rev-accum-rel R (xs @ [x]) (ys @ [y])

lemma rev-accum-rel-refl[intro]: reflp R ⇒ rev-accum-rel R xs xs
unfolding reflp-def
by (induction xs rule: rev-induct) (auto intro: rev-accum-rel.intros)

lemma rev-accum-rel-length:
  assumes rev-accum-rel R xs ys
  shows length xs = length ys
using assms
by (induct auto)

context begin

private inductive-cases rev-accum-relE[consumes 1, case-names nil snoc]: rev-accum-rel P xs ys

lemma rev-accum-rel-butlast[intro]:
  assumes rev-accum-rel P xs ys
  shows rev-accum-rel P (butlast xs) (butlast ys)
using assms by (cases rule: rev-accum-relE) (auto intro: rev-accum-rel.intros)

lemma rev-accum-rel-snoc-eqE: rev-accum-rel P (xs @ [a]) (xs @ [b]) ⇒ P a b
by (auto elim: rev-accum-relE)
end

abbreviation patterns-compatible :: term list ⇒ term list ⇒ bool where
patterns-compatible ≡ rev-accum-rel pattern-compatible

abbreviation patterns-compatibles :: (term list × 'a) fset ⇒ bool where
patterns-compatibles ≡ fpairwise (λ(pats₁, -) (pats₂, -). patterns-compatible pats₁ pats₂)

lemma pattern-compatible-combD:
  assumes length xs = length ys pattern-compatible (list-comb f xs) (list-comb f ys)
  shows patterns-compatible xs ys
using assms by (induction xs ys rule: rev-induct2) (auto intro: rev-accum-rel.intros)

lemma pattern-compatible-combI[intro]:
  assumes patterns-compatible xs ys pattern-compatible f g
shows pattern-compatible \((\text{list-comb } f \, xs) \, (\text{list-comb } g \, ys)\) using assms

proof (induction rule: rev-accum-rel.induct)
case (\text{snoc } xs \, ys \, x \, y)

then have pattern-compatible \((\text{list-comb } f \, xs) \, (\text{list-comb } g \, ys)\) by auto

moreover have pattern-compatible \(x \, y\) if \(\text{list-comb } f \, xs = \text{list-comb } g \, ys\)

proof (rule snoc, rule list-comb-semi-inj)
  show \(\text{length } xs = \text{length } ys\)
    using snoc by (auto dest: rev-accum-rel-length)

qed fact

ultimately show \(?case\)
  by auto

qed (auto intro: rev-accum-rel.intros)

experiment begin

— The above example can be made concrete here. In general, the following identity does not hold:

lemma pattern-compatible \(t \, u\) \leftarrow \(t = u \lor \text{non-overlapping } t \, u\)

apply rule
  apply (erule pattern-compatible-cases; simp)
  apply (erule disjE)
  apply (metis pattern-compatible-refl)

oops

— The counterexample:

definition \(\text{pats1} = [\text{Free } (\text{Name }'f')\), \text{Const } (\text{Name }'\text{nil}')\])
definition \(\text{pats2} = [\text{Free } (\text{Name }'g')\), \text{Const } (\text{Name }'\text{cons}') \$ \text{Free } (\text{Name }'x')\)

\$ \text{Free } (\text{Name }'\text{xs}')\]

proposition non-overlapping \((\text{list-comb } c \, \text{pats1}) \, (\text{list-comb } c \, \text{pats2})\)

unfolding \(\text{pats1-def} \, \text{pats2-def}\)

apply (simp add: app-term-def)
apply (rule non-overlapping-appI2)
apply (rule non-overlapping-const-appI)

done

proposition \(\neg \text{patterns-compatible } \text{pats1} \, \text{pats2}\)

unfolding \(\text{pats1-def} \, \text{pats2-def}\)

apply rule
apply (erule rev-accum-rel.cases)
apply simp
apply auto
apply (erule rev-accum-rel.cases)
apply auto
apply (erule rev-accum-rel.cases)
apply auto
apply (metis overlapping-var1I)
done

end

abbreviation pattern-compatibles :: (term × 'a) fset ⇒ bool where
pattern-compatibles ≡ fpairwise (λ(lhs1, -) (lhs2, -). pattern-compatible lhs1 lhs2)
corollary match-compatible-pat-eq:
  assumes pattern-compatible t1 t2 linear t1 linear t2
  assumes match t1 u = Some env1 match t2 u = Some env2
  shows t1 = t2
  using assms by (metis pattern-compatible-cases match-overlapping)
corollary match-compatible-env-eq:
  assumes pattern-compatible t1 t2 linear t1 linear t2
  assumes match t1 u = Some env1 match t2 u = Some env2
  shows env1 = env2
  using assms by (metis match-compatible-pat-eq option.inject)
corollary match-compatible-eq:
  assumes patterns-compatible ts1 ts2 linears ts1 linears ts2
  assumes matchs ts1 us = Some env1 matchs ts2 us = Some env2
  shows ts1 = ts2 env1 = env2
proof –
  fix name
  have match (name $$ ts1) (name $$ us) = Some env1 match (name $$ ts2) (name $$ us) = Some env2
  using assms by auto
  moreover have length ts1 = length ts2
  using assms by (metis matchs-some-eq-length)
  ultimately have pattern-compatible (name $$ ts1) (name $$ ts2)
  using assms by (auto simp: const-term-def)
  moreover have linear (name $$ ts1) linear (name $$ ts2)
  using assms by (auto intro: linear-list-comb)
note * = calculation
from * have name $$ ts1 = name $$ ts2
  by (rule match-compatible-pat-eq) fact+
thus ts1 = ts2
  by (meson list-comb-inj-second injD)
from * show env1 = env2
  by (rule match-compatible-env-eq) fact+
qed
lemma compatible-find-match:
  assumes pattern-compatibles (fset-of-list cs) list-all (linear o fst) cs is-fmap (fset-of-list cs)
  assumes match pat t = Some env (pat, rhs) ∈ set cs
  shows find-match cs t = Some (env, pat, rhs)
using assms proof (induction cs arbitrary: pat rhs)
case (Cons c cs)
  then obtain [simp]: c = (pat', rhs')
  by force
have find-match ((pat', rhs') # cs) t = Some (env, pat, rhs)
proof (cases match pat' t)
case None
  have pattern-compatibles (fset-of-list cs)
  using Cons by (force simp: fpairwise-alt-def)
  have list-all (linear o fst) cs
  using Cons by (auto simp: list-all-iff)
  have is-fmap (fset-of-list cs)
  using Cons by (meson fset-of-list-subset is-fmap-subset set-subset-Cons)
show ?thesis
  apply (simp add: None)
  apply (rule Cons)
  apply fact+
  using Cons None by force
next
  case (Some env')
  have linear pat linear pat'
  using Cons apply (metis Ball-set comp-apply fst-conv)
  using Cons by simp
moreover from Cons have pattern-compatible pat pat'
  apply (cases pat = pat')
  apply (simp add: pattern-compatible-refl)
  unfolding fpairwise-alt-def
  by (force simp: fset-of-list-elem)
ultimately have env' = env pat' = pat
  using match-compatible-env-eq match-compatible-pat-eq
  using Cons Some
  by blast+
with Cons have rhs' = rhs
  using is-fmapD
  by (metis ⟨c = (pat', rhs')⟩ fset-of-list-elem list.set-intros(1))
show ?thesis
  apply (simp add: Some)
  apply (intro conjI)
  by fact+
qed
thus case
  unfolding \( c = \neg \).
qed auto

context term begin

definition arity-compatible :: \( 'a \Rightarrow 'a \Rightarrow \) bool where
arity-compatible \( t_1 \; t_2 \equiv (\)
  
  let
  \((\text{head}_1, \text{pats}_1) = \text{strip-comb} \; t_1;\)
  \((\text{head}_2, \text{pats}_2) = \text{strip-comb} \; t_2\)
  \nin \text{head}_1 = \text{head}_2 \; \rightarrow \; \text{length} \; \text{pats}_1 = \text{length} \; \text{pats}_2
\)

abbreviation arity-compatibles :: \( (\'a \times 'b) \) \( \) fset \( \Rightarrow \) bool where
arity-compatibles \equiv fpairwise (\( \lambda(\text{lhs}_1, \cdot) \; (\text{lhs}_2, \cdot). \) arity-compatible \( \text{lhs}_1 \; \text{lhs}_2\))

definition head :: \( 'a \Rightarrow \) name where
head \( t \equiv \) const-name (\( \text{fst} \; (\text{strip-comb} \; t)\))

abbreviation heads-of :: (term \( \times 'a\)) \( \) fset \( \Rightarrow \) name fset where
heads-of \( rs \equiv (\text{head} \; \circ \; \text{fst}) \; |^\cdot | \; rs\)

end

definition arity :: \( (\'a \; \text{list} \times 'b) \) \( \) fset \( \Rightarrow \) nat where
arity \( rs = fthe-elem'((\text{length} \; \circ \; \text{fst}) \; |^\cdot | \; rs)\)

lemma arityI:
  assumes \( fBall \; rs \; (\lambda(pats, \cdot). \; \text{length} \; \text{pats} = n)\) \( rs \neq \{||\}\)
  shows arity \( rs = n\)
proof
  have \( (\text{length} \; \circ \; \text{fst}) \; |^\cdot | \; rs = \{\{n\}\}\)
    using \( \text{assms by force}\)
  thus \text{thesis}
    unfolding arity-def fthe-elem'-eq by simp
qed

end

1.2 Reflecting HOL datatype definitions

theory HOL-Datatype
imports
  Terms-Extras
  HOL-Library.Datatype-Records
  HOL-Library.Finite-Map
  Higher-Order-Terms.Name
begin
datatype typ =
  TVar name 
  TApp name typ list

datatype-compat typ

context begin

qualified definition tapp-0 where
tapp-0 tc = TApp tc []

qualified definition tapp-1 where
tapp-1 tc t1 = TApp tc [t1]

qualified definition tapp-2 where
tapp-2 tc t1 t2 = TApp tc [t1, t2]

end

quickcheck-generator typ
  constructors:
  TVar,
  HOL-Datatype.tapp-0,
  HOL-Datatype.tapp-1,
  HOL-Datatype.tapp-2

datatype-record dt-def =
  tparams :: name list
  constructors :: (name, typ list) fmap

ML-file hol-datatype.ML

end

1.3 Constructor information

type-synonym C-info = (name, dt-def) fmap

locale constructors =
  fixes C-info :: C-info
begin
definition flat-C-info :: (string × nat × string) list where
flat-C-info = do {
  (tname, Cs) ← sorted-list-of-fmap C-info;
  (C, params) ← sorted-list-of-fmap (constructors Cs);
  [(as-string C, (length params, as-string tname))]
}
definition all-tdefs :: name fset where
all-tdefs = fdom C-info
definition C :: name fset where
C = ffUnion (fdom |\*| constructors |\*| fmrn C-info)
definition all-constructors :: name list where
all-constructors =
  concat (map (λ(-, Cs). map fst (sorted-list-of-fmap (constructors Cs))) (sorted-list-of-fmap C-info))
end

declare constructors.C-def [code]
declare constructors.flat-C-info-def [code]
declare constructors.all-constructors-def [code]
export-code
  constructors.C constructors.flat-C-info constructors.all-constructors
  checking Scala
end

1.4 Special constants

theory Consts
imports
  Constructors
  Higher-Order-Terms.Nterm
begin
locale special-constants = constructors
locale pre-constants = special-constants +
  fixes heads :: name fset
begin
definition all-consts :: name fset where
all-consts = heads |\cup| C
abbreviation welldefined :: `'a::term ⇒ bool where
welldefined t ≡ consts t ⊆| allconsts

sublocale welldefined: simple-syntactic-and welldefined
by standard auto
end

declare pre-constants.all-consts-def[code]

locale constants = pre-constants +
  assumes disjnt: fdisjnt heads C
  — Conceptually the following assumptions should belong into constructors, but I
  prefer to keep that one assumption-free.
  assumes distinct-ctr: distinct all-constructors
begin

lemma distinct-ctr: distinct (map as-string all-constructors)
unfolding distinct-map
using distinct-ctr
by (auto intro: inj-onI dest: name.expand)

end

end

1.5 Term algebra extended with wellformedness

theory Strong-Term
imports Consts
begin

class pre-strong-term = term +
  fixes wellformed :: `'a ⇒ bool
  fixes all-frees :: `'a ⇒ name fset
  assumes wellformed-const[simp]: wellformed (const name)
  assumes wellformed-free[simp]: wellformed (free name)
  assumes wellformed-app[simp]: wellformed (app u_1 u_2) ⇐⇒ wellformed u_1 ∧
wellformed u_2
  assumes all-frees-const[simp]: all-frees (const name) = fempty
  assumes all-frees-free[simp]: all-frees (free name) = { | name |}
  assumes all-frees-app[simp]: all-frees (app u_1 u_2) = all-frees u_1 |∪| all-frees u_2
begin

abbreviation wellformed-env :: (name, `'a) fmap ⇒ bool where
wellformed-env ≡ fmpred (λ- wellformed)
end
context pre-constants begin

definition shadows-consts :: 'a::pre-strong-term ⇒ bool where
  shadows-consts t ⟷ ¬ fdisjnt all-consts (all-frees t)

sublocale shadows: simple-syntactic-or shadows-consts
  by standard (auto simp: shadows-consts-def fdisjnt-alt-def)

abbreviation not-shadows-consts-env :: (name, 'a::pre-strong-term) fmap ⇒ bool
  where
  not-shadows-consts-env ≡ fmpred (λ s. ¬ shadows-consts s)

end

declare pre-constants.shadows-consts-def[code]

class strong-term = pre-strong-term +
  assumes raw-frees-all-frees: abs-pred (λt. frees t |⊆| all-frees t) t
  assumes raw-subst-wellformed: abs-pred (λt. wellformed t −→ (∀ env. wellformed-env
    env −→ wellformed (subst t env))) t

begin

lemma frees-all-frees: frees t |⊆| all-frees t
  proof (induction t rule: raw-induct)
    case (abs t)
    show ?case
      by (rule raw-frees-all-frees)
  qed auto

lemma subst-wellformed: wellformed t −→ wellformed-env env −→ wellformed
  (subst t env)
  proof (induction t arbitrary: env rule: raw-induct)
    case (abs t)
    show ?case
      by (rule raw-subst-wellformed)
  qed (auto split: option.splits)

end

global-interpretation wellformed: subst-syntactic-and wellformed :: 'a::strong-term
⇒ bool
  by standard (auto simp: subst-wellformed)

instantiation term :: strong-term begin

fun all-frees-term :: term ⇒ name fset where
  all-frees-term (Free x) = { | x | }
  all-frees-term (t₁ $ t₂) = all-frees-term t₁ |∪| all-frees-term t₂ |


all-frees-term (\Lambda t) = all-frees-term t | all-frees-term - = {||}

lemma frees-all-frees-term[simp]: all-frees t = frees (t::term)
by (induction t) auto

definition wellformed-term :: term \Rightarrow bool where
[simp]: wellformed-term t \Leftarrow Term.wellformed t

instance proof (standard, goal-cases)
case 8

have *: abs-pred P t if P t for P and t :: term
  unfolding abs-pred-term-def using that
  by auto

show ?case
  apply (rule *)
  unfolding wellformed-term-def
  by (auto simp: Term.subst-wellformed)
qed (auto simp: const-term-def free-term-def app-term-def abs-pred-term-def)

end

instantiation nterm :: strong-term begin

definition wellformed-nterm :: nterm \Rightarrow bool where
[simp]: wellformed-nterm t \Leftarrow True

fun all-frees-nterm :: nterm \Rightarrow name fset where
all-frees-nterm (Nvar x) = {[x]} | all-frees-nterm (t1 \sqcup n \ t2) = all-frees-nterm t1 | all-frees-nterm t2 | all-frees-nterm (\Lambda n x. t) = finsert x (all-frees-nterm t) | all-frees-nterm (Nconst -) = {||}

instance proof (standard, goal-cases)
case (7 t)
  unfolding abs-pred-nterm-def
  by auto
qed (auto simp: const-nterm-def free-nterm-def app-nterm-def abs-pred-nterm-def)

end

lemma (in pre-constants) shadows-consts-frees:
  fixes t :: 'a::strong-term
  shows \sim shadows-consts t \Leftarrow fdisjnt all-consts (frees t)
unfolding fdisjnt-all-def shadows-consts-def
using frees-all-frees
by auto

abbreviation wellformed-clauses :: - ⇒ bool where
wellformed-clauses cs ≡ list-all (λ(pat, t). linear pat ∧ wellformed t) cs ∧ distinct (map fst cs) ∧ cs ≠ []
end

1.6 Terms with sequential pattern matching

theory Sterm
imports Strong-Term
begin

datatype sterm =
  Sconst name |
  Svar name |
  Sabs (clauses: (term × sterm) list) |
  Sapp sterm sterm (infixl $s 70)

datatype-compat sterm

derive linorder sterm

abbreviation Sabs-single (Λs - [0, 50] 50) where
Sabs-single x rhs ≡ Sabs [((Free x), rhs)]

type-synonym sclauses = (term × sterm) list

lemma sterm-induct[case-names Sconst Svar Sabs Sapp]:
  assumes \(∀x. P (Sconst x)\)
  assumes \(∀x. P (Svar x)\)
  assumes \(∀cs. (\forall pat t. (pat, t) ∈ set cs ⇒ P t) ⇒ P (Sabs cs)\)
  assumes \(∀t u. P t ⇒ P u ⇒ P (t \ s u)\)
  shows \(P t\)
using assms
  apply induction-schema
  apply pat-completeness
  apply lexicographic-order
  done

instantiation sterm :: pre-term begin

definition app-sterm where
app-sterm t u = t $s u

fun unapp-sterm where
unapp-sterm (t $s u) = Some (t, u) |
\( \text{unapp-sterm} \, \text{=} \, \text{None} \)

**definition const-sterm where**
\( \text{const-sterm} \, \text{=} \, \text{Sconst} \)

**fun unconst-sterm where**
\( \text{unconst-sterm} \, (\text{Sconst name}) \, \text{=} \, \text{Some name} \)
\( \text{unconst-sterm} \, \text{=} \, \text{None} \)

**fun unfree-sterm where**
\( \text{unfree-sterm} \, (\text{Svar name}) \, \text{=} \, \text{Some name} \)
\( \text{unfree-sterm} \, \text{=} \, \text{None} \)

**definition free-sterm where**
\( \text{free-sterm} \, \text{=} \, \text{Svar} \)

**fun frees-sterm where**
\( \text{frees-sterm} \, (\text{Svar name}) \, \text{=} \, \{ \text{name} \} \)
\( \text{frees-sterm} \, (\text{Sconst -}) \, \text{=} \, \{ \} \)
\( \text{frees-sterm} \, (\text{Sabs cs}) \, \text{=} \, \text{ffUnion} \, (\text{fset-of-list} \, (\text{map} \, (\lambda(\text{pat}, \text{rhs}). \, \text{frees-sterm} \, \text{rhs} \, - \, \text{frees} \, \text{pat}) \, \text{cs}))\)
\( \text{frees-sterm} \, (\text{t} \, \text{\$} \, \text{s} \, \text{u}) \, \text{=} \, \text{frees-sterm} \, \text{t} \, | \cup | \, \text{frees-sterm} \, \text{u} \)

**fun subst-sterm where**
\( \text{subst-sterm} \, (\text{Svar s}) \, \text{env} \, \text{=} \, (\text{case} \, \text{fmlookup} \, \text{env} \, \text{s} \, \text{of} \, \text{Some t} \, \Rightarrow \, \text{t} \, | \, \text{None} \, \Rightarrow \, \text{Svar s}) \)
\( \text{subst-sterm} \, (\text{t} \, \text{\$} \, \text{s} \, \text{t} \, \text{2}) \, \text{env} \, \text{=} \, \text{subst-sterm} \, \text{t} \, \text{1} \, \text{env} \, \text{\$} \, \text{s} \, \text{subst-sterm} \, \text{t} \, \text{2} \, \text{env} \)
\( \text{subst-sterm} \, (\text{Sabs cs}) \, \text{env} \, \text{=} \, \text{Sabs} \, (\text{map} \, (\lambda(\text{pat}, \text{rhs}). \, (\text{pat}, \text{subst-sterm} \, \text{rhs} \, (\text{fmdrop-fset} \, \text{frees} \, \text{pat}) \, \text{env})) \, \text{cs}) \)
\( \text{subst-sterm} \, \text{t} \, \text{env} \, \text{=} \, \text{t} \)

**fun consts-sterm :: sterm \Rightarrow name fset where**
\( \text{consts-sterm} \, (\text{Svar -}) \, \text{=} \, \{ \} \)
\( \text{consts-sterm} \, (\text{Sconst name}) \, \text{=} \, \{ \text{name} \} \)
\( \text{consts-sterm} \, (\text{Sabs cs}) \, \text{=} \, \text{ffUnion} \, (\text{fset-of-list} \, (\text{map} \, (\lambda(\text{-, rhs}). \, \text{consts-sterm} \, \text{rhs}) \, \text{cs})) \)
\( \text{consts-sterm} \, (\text{t} \, \text{\$} \, \text{s} \, \text{u}) \, \text{=} \, \text{consts-sterm} \, \text{t} \, | \cup | \, \text{consts-sterm} \, \text{u} \)

**instance**
**by standard**
\( \text{auto} \)
\( \text{simp: app-sterm-def const-sterm-def free-sterm-def} \)
\( \text{elim: unapp-sterm.elims unconst-sterm.elims unfree-sterm.elims} \)
\( \text{split: option.splits} \)

**end**

**instantiation sterm :: term begin**
definition abs-pred-sterm :: (sterm ⇒ bool) ⇒ sterm ⇒ bool where
[code def]: abs-pred P t ≐ (∀ cs. t = Sabs cs → (∀ pat t. (pat, t) ∈ set cs → P t) → P t)

lemma abs-pred-stermI[intro]:
assumes (∀ cs. (∀ pat t. (pat, t) ∈ set cs → P t) → P (Sabs cs))
shows abs-pred P t
using assms unfolding abs-pred-sterm-def by auto

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

next

next

next

next

next

next
apply clarsimp
subgoal premises prems[rule-format]
  apply (rule prems(1)[OF prems(4)])
subgoal using prems by auto
subgoal using prems unfolding fdisjnt-alt-def by auto
done
done
next
case 5
show ?case
proof (intro abs-pred-sternI allI impI, goal-cases)
case (1 cs env)
show ?case
proof safe
  fix name
  assume name ∈ frees (subst (Sabs cs) env)
  then obtain pat rhs
    where (pat, rhs) ∈ set cs
    and name ∈ frees (subst rhs (fmdrop-fset (frees pat) env))
    and name ∉ frees pat
    by (auto simp: fset-of-list-elem case-prod-twice comp-def ffUnion-alt-def)
  hence name ∈ frees rhs − fmdom (fmdrop-fset (frees pat) env)
    using 1 by (simp add: fmpred-drop-fset)
  hence name ∈ frees rhs − frees pat
    using ⟨name ∉ frees pat⟩ by blast
  show name ∈ frees (Sabs cs)
    apply (simp add: ffUnion-alt-def)
    apply (rule fBexI[where x = (pat, rhs)])
    unfolding prod.case
    apply (fact ⟨name ∈ frees rhs − frees pat⟩)
    unfolding fset-of-list-elem
    by fact
  assume name ∈ fmdom env
  thus False
    using ⟨name ∈ frees rhs − fmdom (fmdrop-fset (frees pat) env)⟩ ⟨name ∉ frees pat⟩
    by fastforce
next
fix name
assume name ∈ frees (Sabs cs) name ∉ fmdom env
then obtain pat rhs
  where (pat, rhs) ∈ set cs name ∈ frees rhs name ∉ frees pat
  by (auto simp: fset-of-list-elem ffUnion-alt-def)
moreover hence \( \text{name} \mid \in \mid \text{frees} \ \text{rhs} \mid \neg \mid \text{fmdom} \ (\text{fmdrop-fset} \ (\text{frees} \ \text{pat})) \) 

\text{env} \mid \neg \mid \text{frees} \ \text{pat} 

\text{using} \ (\text{name} \mid \notin \mid \text{fmdom} \ \text{env}) \text{ by fastforce}

ultimately have \( \text{name} \mid \in \mid \text{frees} \ (\text{subst} \ \text{rhs} \ (\text{fmdrop-fset} \ (\text{frees} \ \text{pat}) \ \text{env})) \) 

\neg \mid \text{frees} \ \text{pat} 

\text{using} \ (1) \text{ by} \ (\text{simp add: fmpred-drop-fset})

show \( \text{name} \mid \in \mid \text{frees} \ (\text{subst} \ (\text{Sabs} \ \text{cs}) \ \text{env}) \)

\text{apply} \ (\text{simp add: case-prod-twice comp-def})

\text{unfolding} \ \text{ffUnion-alt-def}

\text{apply} \ (\text{rule fBexI})

\text{apply} \ (\text{fact} \ (\text{name} \mid \in \mid \text{frees} \ (\text{subst} \ \text{rhs} \ (\text{fmdrop-fset} \ (\text{frees} \ \text{pat}) \ \text{env})) \mid \neg \mid \text{frees} \ \text{pat})))

\text{apply} \ (\text{subst fimage-iff})

\text{apply} \ (\text{rule fBexI[where} \ x = (\text{pat}, \ \text{rhs}]))

\text{apply} \ \text{simp}

\text{using} \ ((\text{pat}, \ \text{rhs}) \in \text{set} \ \text{cs})

\text{by} \ (\text{auto simp: fset-of-list-elem})

\text{qed}

\text{Qed}

next

case 6

show \ ?case

proof \ (\text{intro abs-pred-stermI allI impl, goal-cases})

case \ (1 \ \text{cs} \ \text{env})

— some property on various operations that is only useful in here

have \(*\) : \text{fbind} \ (\text{fnimage} \ m \ (\text{fbind} \ A \ g)) \ f = \text{fbind} \ A \ (\lambda x. \text{fbind} \ (\text{fnimage} \ m \ (g \ x)) \ f)

for \ m \ A \ f \ g

including \ fset.lifting fnmap.lifting

by \ transfer’ force

have \ consnts \ (\text{subst} \ (\text{Sabs} \ \text{cs}) \ \text{env}) = \text{fbind} \ (\text{fset-of-list} \ \text{cs}) \ (\lambda (\text{pat}, \ \text{rhs}). \ consnts \ \text{rhs} \ \mid \cup \mid \text{ffUnion} \ (\text{consnts} \ \mid \mid \text{fnimage} \ (\text{fmdrop-fset} \ (\text{frees} \ \text{pat}) \ \text{env}) \ (\text{frees} \ \text{rhs}))))

apply \ (\text{simp add: fnunion-image-bind-eq})

apply \ (\text{rule fnbind-cong[OF refl]})

apply \ (\text{clarsimp split: prod.splits})

apply \ (\text{subst} \ (1))

apply \ (\text{clarsimp (asm)} \ fset-of-list-elem, \ assumption)

apply \ \text{simp}

by \ (\text{simp add: fnunion-image-bind-eq})

also have \ \ldots = \text{fbind} \ (\text{fset-of-list} \ \text{cs}) \ (\text{consnts} \ o \ \text{snd}) \ \cup \ \text{fbind} \ (\text{fset-of-list} \ \text{cs}) \ (\lambda (\text{pat}, \ \text{rhs}). \ \text{ffUnion} \ (\text{consnts} \ \mid \mid \text{fnimage} \ (\text{fmdrop-fset} \ (\text{frees} \ \text{pat}) \ \text{env}) \ (\text{frees} \ \text{rhs}))))

apply \ (\text{subst} \ fnbind-fun-funion[symmetric])

apply \ (\text{rule fnbind-cong[OF refl]})

by \ \text{auto}
also have \ldots = \text{consts} (\text{Sabs} \; \text{cs} |\cup| \text{fbind} (\text{fset-of-list} \; \text{cs}) (\lambda (\text{pat}, \; \text{rhs}). \text{ffUnion} (\text{consts} |\cdot| \text{fnimage} (\text{fndrop-fset} (\text{frees} \; \text{pat}) \; \text{env}) (\text{frees} \; \text{rhs})))

\begin{itemize}
\item apply (rule cong [OF cong, OF refl - refl, where \text{f1} = \text{funion}])
\item apply (subst funion-image-bind-eq [symmetric])
\item unfolding \text{consts-sterm}. \text{simps}
\item apply (rule arg-cong [where \text{f} = \text{ffUnion}])
\item apply (subst \text{fset-of-list-map})
\item apply (rule \text{fset.map-cong} [OF \text{refl}])
\end{itemize}
by \text{auto}

also have \ldots = \text{consts} (\text{Sabs} \; \text{cs} |\cup| \text{ffUnion} (\text{consts} |\cdot| \text{fnimage} \; \text{env} (\text{fbind} (\text{fset-of-list} \; \text{cs}) (\lambda (\text{pat}, \; \text{rhs}). \text{frees} \; \text{rhs} |\cdot| \text{frees} \; \text{pat}))) \; \text{consts}

\begin{itemize}
\item apply (subst funion-image-bind-eq)
\item apply (subst \text{fnimage-drop-fset})
\item apply (rule cong [OF cong, OF \text{refl} \; \text{refl}, where \text{f1} = \text{funion}])
\item apply (subst \text{*})
\item apply (rule \text{fbind-cong} [OF \text{refl}])
\end{itemize}
by \text{auto}

also have \ldots = \text{consts} (\text{Sabs} \; \text{cs} |\cup| \text{ffUnion} (\text{consts} |\cdot| \text{fnimage} \; \text{env} (\text{frees} (\text{Sabs} \; \text{cs}))))

by (simp only: frees-sterm. \text{simps} \text{fset-of-list-map} \text{fnimage-Union} \text{funion-image-bind-eq})

finally show ?case.
\text{qed}
\text{qed (auto simp: \text{abs-pred-sterm-def})}

end

lemma \text{no-abs-abs[simp]}: \neg \text{no-abs} (\text{Sabs} \; \text{cs})

by (subst \text{no-abs}. \text{simps}) (auto simp: \text{term-cases-def})

\begin{itemize}
\item \text{closed-except-simps:}
\item closed-except (Svar \; x) \; \text{S} \longleftrightarrow x \; |\in| \; \text{S}
\item closed-except (t_1 \; \& t_2) \; \text{S} \longleftrightarrow closed-except \; t_1 \; \text{S} \; \land \; closed-except \; t_2 \; \text{S}
\item closed-except (\text{Sabs} \; \text{cs}) \; \text{S} \longleftrightarrow \text{list-all} (\lambda (\text{pat}, \; \text{t}). \; \text{closed-except} \; \text{t} \; (\text{S} \; |\cup| \; \text{frees} \; \text{pat})) \; \text{cs}
\item closed-except (\text{Sconst name}) \; \text{S} \longleftrightarrow \text{True}
\end{itemize}

\text{proof (goal-cases)}
\text{case 3}
\text{show ?case}
\text{proof (standard, goal-cases)}
\text{case 1}
\text{then show ?case}
\item apply (auto simp: list-all-iff \text{ffUnion-alt-def} \text{fset-of-list-elem} \text{closed-except-def})
\item apply (drule \text{ffUnion-least-rev})
\item apply \text{auto}
\item by (\text{smt case-prod-conv fbspec fimageI fminusI fset-of-list-elem fset-rev-mp})
\text{next}
\text{case 2}
\text{then show ?case}

22
by (fastforce simp: list-all-iff fFUnion-alt-def fset-of-list-elem closed-except-def)

qed

qed (auto simp: fFUnion-alt-def closed-except-def)

lemma closed-except-sabs:
  assumes closed (Sabs cs) (pat, rhs) ∈ set cs
  shows closed-except rhs (frees pat)
using assms unfolding closed-except-def
apply auto
by (metis bot.extremum-uniqueI fempty-iff fFUnion-subset-elem fimageI fminusI fset-of-list-elem old.prod.case)

instantiation sterm :: strong-term begin

fun wellformed-sterm :: sterm ⇒ bool where
  wellformed-sterm (t₁ $ₗ s₂) ←→ wellformed-sterm t₁ ∧ wellformed-sterm t₂ |
  wellformed-sterm (Sabs cs) ←→ list-all (λ(pat, t). linear pat ∧ wellformed-sterm t) cs ∧ distinct (map fst cs) ∧ cs ≠ [] |
  wellformed-sterm - ←→ True

primrec all-frees-sterm :: sterm ⇒ name fset where
  all-frees-sterm (Svar x) = {{x}} |
  all-frees-sterm (t₁ $ₗ s₂) = all-frees-sterm t₁ ∪ all-frees-sterm t₂ |
  all-frees-sterm (Sabs cs) = fFUnion (fset-of-list (map (λ(P, T). P ∪ T) (map (map-prod frees all-frees-sterm) cs))) |
  all-frees-sterm (Sconst -) = {{}}

instance proof (standard, goal-cases)
  case (7 t)
  show ?case
    apply (intro abs-pred-stermI allI impl)
    apply simp
    apply (rule fFUnion-least)
    apply (rule fBallI)
    apply auto
    apply (subst fFUnion-alt-def)
    apply simp
    apply (rule-tac x = (a, b) in fBexI)
    by (auto simp: fset-of-list-elem)
next
  case (8 t)
  show ?case
    apply (intro abs-pred-stermI allI impl)
    apply (simp add: list.pred-map comp-deq case-prod-twice, safe)
    subgoal
      apply (subst list-all-iff)
      apply (rule ballI)
      apply safe[1]
      apply (fastforce simp: list-all-iff)

23
subgoal premises prems[rule-format]
  apply (rule prems)
  apply (fact prems)
  using prems apply (fastforce simp: list-all-iff)
  using prems by force
done
subgoal
  apply (subst map-cong[OF refl])
  by auto
done
qed (auto simp: const-sterm-def free-sterm-def app-sterm-def)

end

lemma match-sabs[simp]: ¬ is-free t ⟹ match t (Sabs cs) = None
  by (cases t) auto

context pre-constants begin

lemma welldefined-sabs: welldefined (Sabs cs) ⟷ list-all (λ(_, t). welldefined t) cs
  apply (auto simp: list-all-iff ffUnion-alt-def dest: ffUnion-least-rev)
  apply (subst (asm) list-all-iff-fset[symmetric])
  apply (auto simp: list-all-iff fset-of-list-elem)
  done

lemma shadows-consts-sterm-simps[simp]:
  shadows-consts (t1 $ t2) ⟷ shadows-consts t1 ∨ shadows-consts t2
  shadows-consts (Svar name) ⟷ name ∈ all-consts
  shadows-consts (Sabs cs) ⟷ list-ex (λ(pat, t). ¬ fdisjnt all-consts (frees pat) ∨ shadows-consts t) cs
  shadows-consts (Sconst name) ⟷ False
proof (goal-cases)
  case 3
  unfolding shadows-consts-def list-ex-iff
  apply rule
  subgoal
    by (force simp: ffUnion-alt-def fset-of-list-elem fdisjnt-alt-def elim!: ballE)
  subgoal
    apply (auto simp: fset-of-list-elem fdisjnt-alt-def)
    by (auto simp: fset-eq-empty-iff ffUnion-alt-def fset-of-list-elem elim!: allE fBallE)
  done
qed (auto simp: shadows-consts-def fdisjnt-alt-def)

lemma subst-shadows:
  assumes ¬ shadows-consts (t::sterm) not-shadows-consts-env Γ
shows ¬ shadows-cons (subst t Γ)
using assms proof (induction t arbitrary: Γ rule: sterm-induct)
case (Sabs cs)
show ?case
  apply (simp add: list-ex-iff case-prod-twice)
  apply (rule ballI)
subgoal for c
  apply (cases c, hypsubst-thin, simp)
  apply (rule conjI)
subgoal using Sabs(2) by (fastforce simp: list-ex-iff)
subgoal using Sabs(1) by force
subgoal using Sabs(3) by force
done
done
qed (auto split: option.splits)

end

end

1.7 Terms with explicit pattern matching

theory Pterm
imports
  ../Utils/Compiler-Utils
  Consts

  Sterm — Inclusion of this theory might seem a bit strange. Indeed, it is only for technical reasons: to allow for a quickcheck setup.

begin

datatype pterm =
  Pconst name |
  Pvar name |
  Pabs (term × pterm) fset |
  Papp pterm pterm (infixl $p 70)

primrec sterm-to-pterms :: sterm ⇒ pterm where
  sterm-to-pterms (Sconst name) = Pconst name |
  sterm-to-pterms (Svar name) = Pvar name |
  sterm-to-pterms (t $p u) = sterm-to-pterms t $p, sterm-to-pterms u |
  sterm-to-pterms (Sabs cs) = Pabs (fset-of-list (map (map-prod id sterm-to-pterms) cs))

quickcheck-generator pterm
— will print some fishy “constructor” names, but at least it works
constructors: sterm-to-pterms

25
lemma stem-to-pterm-total:
  obtains $t'$ where $t = \text{stem-to-pterm} \ t'$
proof (induction $t$ arbitrary: thesis)
  case (Pconst $x$)
  then show $?case$
    by (metis stem-to-pterm.simps)
next
  case (Pvar $x$)
  then show $?case$
    by (metis stem-to-pterm.simps)
next
  case (Pabs $cs$)
  from Pabs.IH obtain $cs'$ where $cs = fset-of-list \ (map \ (map-prod \ id \ stem-to-pterm) \ cs')$
  apply atomize-elem
proof (induction $cs$)
  case empty
  show $?case$
    apply (rule exI [where $x = []$])
    by simp
next
  case (insert $c \ cs$)
  obtain $pat \ rhs$ where $c = (pat, rhs)$ by (cases $c$) auto
  have $\exists \ cs'. \ cs = fset-of-list \ (map \ (map-prod \ id \ stem-to-pterm) \ cs')$
    apply (rule insert.prems) unfolding finsert.rep-eq
    by blast
  then obtain $cs'$ where $cs = fset-of-list \ (map \ (map-prod \ id \ stem-to-pterm) \ cs')$
    by blast
  obtain $rhs'$ where $rhs = \text{stem-to-pterm} \ rhs'$
    apply (rule insert.prems[of $(pat, rhs)$ rhs])
    unfolding $\langle c = - \rangle$ by simp+
  show $?case$
    apply (rule exI[where $x = (pat, rhs') \ # \ cs']$)
    unfolding $\langle c = - \rangle \ (cs = -) \ (rhs = -)$
    by (simp add: id-def)
  qed
  hence Pabs $cs = \text{stem-to-pterm} \ (Sabs \ cs')$
    by simp
  then show $?case$
    using Pabs by metis
next
  case (Papp $t1 \ t2$)
  then obtain $t1' \ t2'$ where $t1 = \text{stem-to-pterm} \ t1' \ t2 = \text{stem-to-pterm} \ t2'$
    by metis
then have \( t_1 \$_p t_2 = \text{st term-to-pterm} \ (t_1' \$_p t_2') \)
  by simp
with \( \text{Papp} \) show \( \text{?case} \)
  by metis
qed

lemma \( \text{pterm-induct} \) [case-names \( \text{Pconst} \ \text{Pvar} \ \text{Pabs} \ \text{Papp} \)]:
  assumes \( \exists x. \ P \ (\text{Pconst} \ x) \)
  assumes \( \exists x. \ P \ (\text{Pvar} \ x) \)
  assumes \( \exists cs. \ (\exists \text{pat} \ t. \ (\text{pat}, \ t) |\in| cs \implies P \ t) \implies P \ (\text{Pabs} \ cs) \)
  assumes \( \exists t \ u. \ P \ t \implies P \ u \implies P \ (t \$_p u) \)
  shows \( P \ t \)
proof (rule \( \text{pterm-induct} \), goal-cases)
  case \( (\exists cs) \)
  show \( \text{?case} \)
    apply (rule assms)
    using \( 3 \)
    apply (subst \( \text{asm} \) \( \text{fm member.rep-eq}[\text{symmetric}] \))
    by auto
qed \( \text{fact+} \)

instantiation \( \text{pterm} :: \text{pre-term} \) begin

definition \( \text{app-pterm} \) where
\( \text{app-pterm} \ t \ u = t \$_p u \)

fun \( \text{unapp-pterm} \) where
\( \text{unapp-pterm} \ (t \$_p u) = \text{Some} \ (t, \ u) \ | \)
\( \text{unapp-pterm} \ - = \text{None} \)

definition \( \text{const-pterm} \) where
\( \text{const-pterm} = \text{Pconst} \)

fun \( \text{unconst-pterm} \) where
\( \text{unconst-pterm} \ (\text{Pconst} \ \text{name}) = \text{Some} \ \text{name} \ | \)
\( \text{unconst-pterm} \ - = \text{None} \)

definition \( \text{free-pterm} \) where
\( \text{free-pterm} = \text{Pvar} \)

fun \( \text{unfree-pterm} \) where
\( \text{unfree-pterm} \ (\text{Pvar} \ \text{name}) = \text{Some} \ \text{name} \ | \)
\( \text{unfree-pterm} \ - = \text{None} \)

function (\( \text{sequential} \)) \( \text{subst-pterm} \) where
\( \text{subst-pterm} \ (\text{Pvar} \ \text{s}) \ \text{env} = (\text{case \ fm lookup \ env \ \text{s} \ of \ Some \ \text{t} \ \Rightarrow \ \text{t} | \ None \ \Rightarrow \ \text{Pvar} \ \text{s}) \ | \)
\( \text{subst-pterm} \ (t_1 \$_p t_2) \ \text{env} = \text{subst-pterm} \ t_1 \ \text{env} \$_p \ \text{subst-pterm} \ t_2 \ \text{env} \ | \)
\( \text{subst-pterm} \ (\text{Pabs} \ \text{cs}) \ \text{env} = \text{Pabs} \ ((\lambda (\text{pat}, \ \text{rhs}). \ (\text{pat}, \ \text{subst-pterm} \ \text{rhs} \ \text{findrop-fset})) \ \text{cs}) \ | \)
\( \text{subst-pterm} \ (\text{Pconst} \ \text{name}) \ \text{env} = \text{Pconst} \ \text{name} \ | \)
\( \text{subst-pterm} \ - = \text{None} \)

end
(frees pat) env)) \mid \cdot \mid cs \mid

subst-pterm t \cdot = t

by pat-completeness auto

termination

proof (relation measure (size o fst), goal-cases)

case 4

then show ?case

apply auto

including fset.lifting apply transfer

apply (rule le-imp-less-Suc)

apply (rule sum-nat-le-single[where y = (a, (b, size b)) for a b])

by auto

qed auto

primrec consts-pterm :: pterm \Rightarrow name fset

consts-pterm (Pconst x) = \{\{x\}\}\mid

consts-pterm (t_1 \cdot p \cdot t_2) = constsp-term t_1 \mid\mid constsp-term t_2 \mid

consts-pterm (Pabs cs) = \bigcup_{i=\cdot} \map-prod id consts-pterm \mid \cdot \mid cs \mid

consts-pterm (Pvar \cdot) = \{\}\mid

primrec frees-pterm :: pterm \Rightarrow name fset

frees-pterm (Pvar x) = \{\{x\}\}\mid

frees-pterm (t_1 \cdot p \cdot t_2) = frees-p-term t_1 \mid\mid frees-p-term t_2 \mid

frees-p-term (Pabs cs) = \bigcup_{i=\cdot} \map-prod id frees-p-term \mid \cdot \mid cs \mid

frees-p-term (Pconst \cdot) = \{\}\mid

instance

by standard

(auto

  simp: app-p-term-def const-p-term-def free-p-term-def

  elim: unapp-p-term.elims unconst-p-term.elims unfree-p-term.elims

  split: option.splits)

end

corollary subst-pabs-id:

  assumes \text{\textsf{\textbar} pat rhs. (pat, rhs) \in\mid cs \Rightarrow subst rhs (fmdrop-fset (frees pat) env)} \text{\textsf{\textequiv} rhs}

  shows subst (Pabs cs) env = Pabs cs

apply (subst subst-pterm.simps)

apply (rule arg-cong[where f = Pabs])

apply (rule fset-map-snd-id)

apply (rule assms)

apply (subst (asm) fnmember.rep-eq[symmetric])

apply assumption

done

28
corollary frees-pabs-alt-def:
\[ \text{frees} (\text{Pabs} \ cs) = \text{ffUnion} ((\lambda (\text{pat}, \text{rhs}). \text{frees} \ \text{rhs} - \text{frees} \ \text{pat}) \ |\ | \ cs) \]
apply simp
apply (rule arg-cong[where \( f = \text{ffUnion} \])
apply (rule fset.map-cong[\( \text{OF} \ \text{refl} \])
by auto

lemma sterm-to-pterm-frees[simp]: frees (sterm-to-pterm \( t \)) = frees \( t \)
proof (induction \( t \))
case (Sabs \( cs \))
show ?case
  apply simp
  apply (rule arg-cong[where \( f = \text{ffUnion} \])
  apply (rule fimage-cong[\( \text{OF} \ \text{refl} \])
  apply clarsimp
  apply (subst Sabs)
  by (auto simp: fset-of-list-elem snds.simps)
qed auto

lemma sterm-to-pterm-consts[simp]: consts (sterm-to-pterm \( t \)) = consts \( t \)
proof (induction \( t \))
case (Sabs \( cs \))
show ?case
  apply simp
  apply (rule arg-cong[where \( f = \text{ffUnion} \])
  apply (rule fimage-cong[\( \text{OF} \ \text{refl} \])
  apply clarsimp
  apply (subst Sabs)
  by (auto simp: fset-of-list-elem snds.simps)
qed auto

lemma subst-sterm-to-pterm:
\[ \text{subst} (\text{sterm-to-pterm} \( t \)) (\text{fmmap} \text{sterm-to-pterm} \text{env}) = \text{sterm-to-pterm} (\text{subst} \( t \) \text{env}) \]
proof (induction \( t \) arbitrary: \text{env} rule: sterm-induct)
case (Sabs \( cs \))
show ?case
  apply simp
  apply (rule fset.map-cong[\( \text{OF} \text{refl} \])
  apply (auto split: prod.splits)
  apply (rule Sabs)
  by (auto simp: fset-of-list-elem rep-eq)
qed (auto split: option.splits)

instantiation pterm :: term begin

definition abs-pred-pterms :: (pterm \( \Rightarrow \) bool) \( \Rightarrow \) pterm \( \Rightarrow \) bool where
[\( \text{code del} \): \( \text{abs-pred} \ P \ t \leftarrow\rightarrow (\forall \text{cs} \ t = \text{Pabs} \ cs \rightarrow (\forall \text{pat} t. (\text{pat}, t) |\in| \text{cs} \rightarrow P t) \rightarrow P t) \]
context begin

private lemma abs-pred-trivI0: P t \implies abs-pred P (t::pterm)
unfolding abs-pred-ptermd-def by auto

instance proof (standard, goal-cases)
case (1 P t)
then show ?case
  by (induction t rule: pterm-induct)
     (auto simp: const-ptermd-def free-ptermd-def app-ptermd-def abs-pred-ptermd-def)

next

case (2 t)
show ?case
  unfolding abs-pred-ptermd-def
  apply clarify
  apply (rule subst-pabs-id)
  by blast

next

case (3 x t)
show ?case
  unfolding abs-pred-ptermd-def
  apply clarsimp
  apply (rule fset.map-cong0)
  apply (rename-tac c)
  apply (case-tac c; hypsubst-thin)
  apply simp
  subgoal for cs env pat rhs
    apply (cases x [\in] frees pat)
    subgoal
      apply (rule arg-cong[where f = subst rhs])
      by (auto intro: fmap-ext)
    subgoal premises prems[rule-format]
      apply (subst (2) prems(1)[symmetric, where pat = pat])
    subgoal
      by (subst fmember.rep-eq) fact
    subgoal
      using prems unfolding fUnion-alt-def
      by (auto simp: fmember.rep-eq fset-of-list.rep-eq elim!: fBallE)
    subgoal
      apply (rule arg-cong[where f = subst rhs])
      by (auto intro: fmap-ext)
done
done
done
next
case (4 t)
show ?case
unfolding abs-pred-pterm-def
apply clarsimp
apply (rule fset.map-cong0)
apply clarsimp
subgoal premises prems[rule-format] for cs env₁ env₂ a b
apply (rule prems(2)[unfolded fmember.rep-eq, OF prems(5)])
using prems unfolding fdisjnt-alt-def by auto
done
next
case 5
show ?case
proof (rule abs-pred-trivI0, clarify)
fix t :: pterm
fix env :: (name, pterm) fmap
obtain t' where t = sterm-to-pterm t'
  by (rule sterm-to-pterm-total)
obtain env' where env = fmmap sterm-to-pterm env'
  by (metis fmmap-total sterm-to-pterm-total)
show frees (subst t env) = frees t - fmdom env if fmpred (λ-. closed) env
  unfolding (t = '-' (env = '-'))
  apply simp
  apply (rule subst-stterm-to-pterm)
  apply simp
  apply (rule subst-frees)
  using that unfolding (env = '-')
  apply simp
  apply (rule fmpred-mono-strong; assumption?)
  unfolding closed-except-def by simp
qed
next
case 6
show ?case
proof (rule abs-pred-trivI0, clarify)
fix t :: pterm
fix env :: (name, pterm) fmap
obtain t' where t = sterm-to-pterm t'
  by (rule sterm-to-pterm-total)
obtain env' where env = fmmap sterm-to-pterm env'
  by (metis fmmap-total sterm-to-pterm-total)
show consts (subst t env) = consts t |∪| ffUnion (consts |'| fmimage env (frees t))
unfolding \( t = \lambda \) (env = \_)
apply simp
apply (subst comp-def)
apply simp
apply (subst subst-sterm-to-pterms)
apply simp
apply (rule subst-consts')
done
qed
qed (rule abs-pred-trivI0)
end
end

lemma no-abs-abs[simp]; \( \neg \) no-abs (Pabs cs)
by (subst no-abs.simps) (auto simp: term-cases-def)

lemma sterm-to-pterms:
  assumes no-abs t
  shows sterm-to-pterms t = convert-term t
using assms proof induction
  case (free name)
  show ?case
    apply simp
    apply (simp add: free-sterm-def free-pterm-def)
done
next
  case (const name)
  show ?case
    apply simp
    apply (simp add: const-sterm-def const-pterm-def)
done
next
  case (app t_1 t_2)
  then show ?case
    apply simp
    apply (simp add: app-sterm-def app-pterm-def)
done
qed

abbreviation Pabs-single (\( \Lambda \_ p \_ - [0, 50] 50 \)) where
Pabs-single x rhs \( \equiv \) Pabs \( \{ | \) (Free x, rhs) \} \)

lemma closed-except-simps:
closed-except (Pvar x) S \( \leftrightarrow \) x \( | \in | \) S
closed-except (t_1 \_ t_2) S \( \leftrightarrow \) closed-except t_1 S \( \land \) closed-except t_2 S
closed-except (Pabs cs) S \( \leftrightarrow \) fBall cs (\( \lambda \) (pat, t). closed-except t \( \{ | \cup | \) frees pat))
proof goal-cases
  case 3
  show ?case
    proof (standard, goal-cases)
      case 1
      then show ?case
        apply (auto simp: ffUnion-alt-def closed-except-def)
        apply (drule ffUnion-least-rev)
        apply auto
        by (smt case-prod-conv fBall-alt-def fminus-iff fset-rev-mp id-apply map-prod-simp)
    next
    case 2
    then show ?case
    by (fastforce simp: ffUnion-alt-def closed-except-def)
  qed
qed (auto simp: ffUnion-alt-def closed-except-def)

instantiation pterm :: pre-strong-term begin

function (sequential) wellformed-pterms :: pterm ⇒ bool where
  wellformed-pterms (t1, t2) ←→ wellformed-pterms t1 ∧ wellformed-pterms t2 |
  wellformed-pterms (Pabs cs) ←→ fBall cs (λ(pat, t). linear pat ∧ wellformed-pterms t) ∧ is-fmap cs ∧ pattern-compatibles cs ∧ cs ≠ {||} |
  wellformed-pterms - ←→ True
by pat-completeness auto

termination
proof (relation measure size, goal-cases)
  case 4, 
  then show ?case
  apply auto
  including fset.lifting apply transfer
  apply (rule le-imp-less-Suc)
  apply (rule sum-nat-le-single|where y = (a, (b, size b)) for a b)
  by auto
  qed auto

primrec all-frees-pterms :: pterm ⇒ name fset where
  all-frees-pterms (Pvar x) = {{x}} |
  all-frees-pterms (t1, t2) = all-frees-pterms t1 | all-frees-pterms t2 |
  all-frees-pterms (Pabs cs) = ffUnion ((λ(P, T). all-frees-pterms P |∪| T) |^| map-prod-frees all-frees-pterms |^| cs) |
  all-frees-pterms (Pconst -) = {||}

instance
  by standard (auto simp: const-pterms-def free-pterms-def app-pterms-def)

end
lemma sterm-to-pterms-all-frees[simp]: all-frees (stem-to-pterms t) = all-frees t
proof (induction t)
case (Sabs cs)
show ?case
apply simp
apply (rule arg-cong[where f = ffUnion])
apply (rule fmap-cong[OF refl])
apply clarsimp
apply (subst Sabs)
by (auto simp: fset-of-list-elem snds.simps)
qed auto

instance pterm :: strong-term proof (standard, goal-cases)
case (1 t)
obtain t’ where t = stem-to-pterms t’
  by (metis stem-to-pterms-total)
show ?case
apply (rule abs-pred-trivI)
unfolding sterm-to-pterms-all-frees sterm-to-pterms-frees
by (rule frees-all-frees)
next
case (2 t)
show ?case
unfolding abs-pred-pterms-def
apply (intro allI impI)
apply (simp add: case-prod-twice, intro conjI)
subgoal by blast
subgoal by (auto intro: is-fmap-image)
subgoal
  unfolding fpairwise-image fpairwise-alt-def
  by (auto elim!: fBallE)
done
qed

lemma wellformed-PabsI:
  assumes is-fmap cs pattern-compatibles cs cs \neq \{||\}
  assumes \bigwedge pat t. \langle pat, t \rangle \in cs \Longrightarrow linear pat
  assumes \bigwedge pat t. \langle pat, t \rangle \in cs \Longrightarrow wellformed t
  shows wellformed (Pabs cs)
using assms by auto

corollary subst-closed-pabs:
  assumes (pat, rhs) \in cs closed (Pabs cs)
  shows subst rhs (fmdrop-fset (frees pat) env) = rhs
using assms by (subt subst-closed-except-id) (auto simp: fdisjnt-alt-def closed-except-simps)

lemma (in constants) shadows-consts-pterms-simps[simp]:
  shadows-consts (t_1 $p, t_2) \longleftrightarrow shadows-consts t_1 \lor shadows-consts t_2
shadows-consts (Pvar name) $\leftrightarrow$ name $\in$ allconsts
shadows-consts (Pabs cs) $\leftrightarrow$ fBex cs ($\lambda$(pat, t). shadows-consts pat $\lor$ shadows-consts t)
shadows-consts (Pconst name) $\leftrightarrow$ False

proof goal-cases
  case 3

  show ?case
  unfolding shadows-consts-def
  apply rule
  subgoal
    by (force simp: fUnion-alt-def fset-of-list-elem fdisjnt-alt-def elim!: ballE)
  subgoal
    apply (auto simp: fset-of-list-elem fdisjnt-alt-def)
    by (auto simp: fset-eq-empty-iff fUnion-alt-def fset-of-list-elem elim!: allE)
  qed (auto simp: shadows-consts-def fdisjnt-alt-def)

end

1.8 Irreducible terms (values)

theory Term-as-Value
imports Sterm
begin

1.9 Viewing sterm as values

declare list_pred_mono[mono]

current context constructors begin

inductive is-value :: sterm $\Rightarrow$ bool where
abs: is-value (Sabs cs) |
constr: list-all is-value vs $\Rightarrow$ name $\in$ C $\Rightarrow$ is-value (name $\forall$ vs)

lemma value-distinct:
Sabs cs $\ne$ name $\forall$ ts (is ?P)
name $\forall$ ts $\ne$ Sabs cs (is ?Q)

proof -
  show ?P
    apply (rule list-comb-cases [where f = const name and cs = ts])
    apply (auto simp: const-sterm-def is-app-def elim: unapp-sterm.elims)
  done
  thus ?Q
    by simp

end

35
qed

abbreviation value-env :: (name, sterm) fmap ⇒ bool where
value-env ≡ fmap \ (λ. is-value)

lemma svar-value[simp]; ¬ is-value (Svar name)
proof
  assume is-value (Svar name)
  thus False
  apply (cases rule: is-value.cases)
  apply (fold free-sterm-def)
  by simp
qed

lemma value-cases:
  obtains (comb) name vs where list-all is-value vs t = name $$ vs name \in C
         | (abs) cs where t = Sabs cs
         | (nonvalue) ¬ is-value t
proof (cases t)
  case Svar
  thus thesis using nonvalue by simp
next
  case Sabs
  thus thesis using abs by (auto intro: is-value.abs)
next
  case (Sconst name)
  have list-all is-value [] by simp
  have t = name $$ [] unfolding Sconst by (simp add: const-sterm-def)
  show thesis
    using comb is-value.cases abs nonvalue by blast
next
  case Sapp
  show thesis
  proof (cases is-value t)
    case False
    thus thesis using nonvalue by simp
next
    case True
    then obtain name vs where list-all is-value vs t = name $$ vs name \in C
    unfolding Sapp
    by cases auto
    thus thesis using comb by simp
  qed
qed

end
fun \text{smatch}' :: \text{pat} \Rightarrow \text{term} \Rightarrow (\text{name}, \text{term}) \\text{fmap} \\text{option} \quad \text{where}
\begin{align*}
\text{smatch}' (\text{Patvar name}) \ t &= \text{Some (fmap-of-list [(name, t)])} \\
\text{smatch}' (\text{Patconstr name ps}) \ t &=
\begin{cases}
(\text{case strip-comb} \ t \ \text{of}) & \\
(\text{Sconst name}', \ vs) & \\
(\text{if name} = \text{name}' \land \text{length} \ ps = \text{length} \ vs \ \text{then}) & \\
\text{map-option} (\text{foldl} (+f) \ \text{fmempty}) (\text{those (map2 smatch' ps vs)}) & \\
\text{else} & \\
\text{None} & \\
\end{cases} \\
\end{align*}

\text{lemmas} \ \text{smatch'}-\text{induct} = \text{smatch}'.\text{induct}[\text{case-names var constr}]

\text{context} \ \text{constructors} \ \text{begin}
\text{context} \ \text{begin}

\text{private lemma} \ \text{smatch-list-comb-is-value}:
\begin{align*}
\text{assumes} & \quad \text{is-value} \ t \\
\text{shows} & \quad \text{match} (\text{name} \ $$ \ ps) \ t = (\text{case strip-comb} \ t \ \text{of}) \\
& \quad (\text{Sconst name}', \ vs) & \\
& \quad (\text{if name} = \text{name}' \land \text{length} \ ps = \text{length} \ vs \ \text{then}) & \\
& \quad \text{map-option} (\text{foldl} (+f) \ \text{fmempty}) (\text{those (map2 smatch' ps vs)}) & \\
& \quad \text{else} & \\
& \quad \text{None} & \\
& \ | - \Rightarrow \text{None} \\
& \end{align*}

\text{using} \ \text{assms}
\text{apply} \ \text{cases}
\text{apply} \ (\text{auto simp: strip-list-comb split: option.splits})
\text{apply} \ (\text{subst (2) const-sterm-def})
\text{apply} \ (\text{auto simp: matchs-alt-def})
\text{done}

\text{lemma} \ \text{smatch-smatch'}-eq:
\begin{align*}
\text{assumes} & \quad \text{linear pat is-value} \ t \\
\text{shows} & \quad \text{match} \ pat \ t = \text{smatch'} (\text{mk-pat pat}) \ t \\
& \text{using} \ \text{assms}
\text{proof} \ (\text{induction pat arbitrary: t rule: linear-pat-induct})
\text{case} \ (\text{comb name args})
\begin{align*}
\text{show} & \quad ?\text{case} \\
& \text{using} \ (\text{is-value} \ t) \\
& \text{proof} \ (\text{cases rule: is-value.cases})
& \quad \text{case} \ (\text{abs cs}) \\
& \quad \text{thus} \ ?\text{thesis} \\
& \quad \text{by} \ (\text{force simp: strip-list-comb-const})
& \text{next}
& \quad \text{case} \ (\text{constr args'} \ \text{name}')
\end{align*}
\end{align*}

37
have \( \text{map}^2 \text{match} \text{args} \text{args}' = \text{map}^2 \text{smatch}' (\text{map} \text{mk-pat} \text{args}) \text{args}' \) if length \text{args} = length \text{args}' using that constr(2) comb(2) by (induct \text{args} \text{args}' rule: list-induct2) auto

thus \(?\)thesis using constr apply (auto simp: smatch-list-comb-is-value strip-list-comb map-option-case strip-list-comb-const intro: is-value.intros) apply (subst (2) const-sterm-def) apply (auto simp: matches-alt-def) done qed simp

end

end

end

1.10 A dedicated value type

theory Value imports Term-as-Value begin
datatype value =
  is-Vconstr: Vconstr name value list |
  Vabs sclauses (name, value) fmap |
  Vrecabs (name, sclauses) fmap name (name, value) fmap
type-synonym vrule = name \times value
setup (Sign.mandatory-path quickcheck)
datatype value =
  Vconstr name value list |
  Vabs sclauses (name \times value) list |
  Vrecabs (name \times sclauses) list name (name \times value) list
primrec Value :: quickcheck.value ⇒ value where
Value (quickcheck.Vconstr s vs) = Vconstr s (Value vs) |
Value (quickcheck.Vabs cs Γ) = Vabs cs (fmap-of-list (map (map-prod id Value) Γ)) |
Value (quickcheck.Vrecabs css name Γ) = Vrecabs (fmap-of-list css) name (fmap-of-list
(map (map-prod id Value) Γ))

setup (Sign.parent-path)

quickcheck-generator value
constructors: quickcheck.Value

fun vmatch :: pat ⇒ value ⇒ (name, value) fmap option where
vmatch (Patvar name) v = Some (fmap-list [(name, v)]) |
vmatch (Patconst name ps) (Vconstr name' vs) =
  (if name = name' ∧ length ps = length vs then
   map-option (foldl (++) fmempty) (those (map2 vmatch ps vs))
  else
   None) |
vmatch - - = None

lemmas vmatch-induct = vmatch.induct[case-names var constr]

locale value-pred =
fixes P :: (name, value) fmap ⇒ sclauses ⇒ bool
fixes Q :: name ⇒ bool
fixes R :: name fset ⇒ bool
begin

primrec pred :: value ⇒ bool where
  pred (Vconstr name vs) ←→ Q name ∧ list-all id (map pred vs) |
  pred (Vabs cs Γ) ←→ pred-fmap id (fmmap pred Γ) ∧ P Γ cs |
  pred (Vrecabs css name Γ) ←→
    pred-fmap id (fmmap pred Γ) ∧
    pred-fmap (P Γ) css ∧
    name |∈| fmdom css ∧
    R (fmdom css)

declare pred.simps[simp def]

lemma pred-alt-def[simp, code]:
  pred (Vconstr name vs) ←→ Q name ∧ list-all pred vs
  pred (Vabs cs Γ) ←→ fmpred (λ-. pred) Γ ∧ P Γ cs
  pred (Vrecabs css name Γ) ←→ fmpred (λ-. pred) Γ ∧ pred-fmap (P Γ) css ∧
    name |∈| fmdom css ∧ R (fmdom css)
  by (auto simp: list-all-iff pred.simps)

For technical reasons, we don’t introduce an abbreviation for fmpred (λ-.
pred) env here. This locale is supposed to be interpreted with global-interpretation
(or sublocale and a defines clause. However, this does not affect abbreviations: the abbreviation would still refer to the locale constant, not the
constant introduced by the interpretation.

lemma vmatch-env:
  assumes vmatch pat v = Some env pred v
shows \textsf{fmpred} (\lambda \cdot \text{pred}) \textsf{env}

using \textsf{assms} \textbf{proof} (induction \textit{pat} \textit{v} arbitrary; \textit{env} rule: \textsf{vmatch-induct})
\begin{itemize}
  \item case (\textit{constr} \textit{name} \textit{ps} \textit{name}' \textit{vs})
  \item hence
  \begin{itemize}
    \item map-option \((\textit{foldl} (++) \textsf{fmempty}) \textsf{those} (\textit{map2} \textsf{vmatch} \textit{ps} \textit{vs})) = \textsf{Some} \textsf{env}
    \item name = \textit{name}' \ length \textit{ps} = \textit{length} \textit{vs}
    \item by \textit{(auto split: if-splits)}
  \end{itemize}
  \item then obtain \textit{envs} where \textit{env} = \textsf{foldl} (++) \textsf{fmempty} \textsf{envs} \textsf{map2} \textsf{vmatch} \textit{ps} \textit{vs}
  = \textsf{map} \textsf{Some} \textsf{envs}
  \item by \textit{(blast dest: those-someD)}
\end{itemize}

moreover have \textsf{fmpred} (\lambda \cdot \text{pred}) \textsf{env} if \textsf{env} \in \textsf{set} \textit{envs} for \textit{env}
\begin{itemize}
  \item proof –
  \begin{itemize}
    \item from that have \textsf{Some} \textsf{env} \in \textsf{set} (\textit{map2} \textsf{vmatch} \textit{ps} \textit{vs})
    \item unfolding \textsf{(map2 - - - = -)} by \textit{simp}
    \item then obtain \textit{p} \textit{v} where \textit{p} \in \textsf{set} \textit{ps} \textit{v} \in \textsf{set} \textit{vs} \textit{vmatch} \textit{p} \textit{v} = \textsf{Some} \textsf{env}
    \item apply \textit{(rule map2-elemE)}
    \item by \textit{auto}
    \item hence \textsf{pred} \textsf{v}
    \item using \textit{constr} by \textit{(simp add: list-all-iff)}
    \item show \textit{?thesis}
      \item by \textit{(rule constr; safe?) fact+}
  \end{itemize}
  \item qed
\end{itemize}

ultimately show \textit{?case}
\begin{itemize}
  \item by \textit{auto}
  \item qed \textit{auto}
\end{itemize}

end

\begin{primrec}
\textit{value-to-sterm} :: \textit{value} \Rightarrow \textit{sterm}
\begin{align*}
\textit{value-to-sterm} \ (\textit{Vconstr} \textit{name} \textit{vs}) &= \textit{name} \map \textit{value-to-sterm} \textit{vs} \\
\textit{value-to-sterm} \ (\textit{Vabs} \textit{cs} \textit{\Gamma}) &= \textit{Sabs} \ \text{(map \ (\lambda \textit{(pat}, \textit{t}). \ (\textit{pat}, \textit{subst} \textit{t} \ (\textit{fmdrop-fset} \ (\textit{frees} \textit{pat}) \ (\textit{fmmap} \textit{value-to-sterm} \textit{\Gamma})))) \textit{cs}}) \\
\textit{value-to-sterm} \ (\textit{Vrecabs} \textit{css} \textit{name} \textit{\Gamma}) &= \textit{Sabs} \ \text{(map \ (\lambda \textit{(pat}, \textit{t}). \ (\textit{pat}, \textit{subst} \textit{t} \ (\textit{fmdrop-fset} \ (\textit{frees} \textit{pat}) \ (\textit{fmmap} \textit{value-to-sterm} \textit{\Gamma})))) \ (\textit{the \ (\textit{fmlookup} \textit{css} \textit{name})))}
\end{align*}
\end{primrec}

This locale establishes a connection between a predicate on values with the corresponding predicate on stterms, by means of \textit{value-to-sterm}.

\textbf{locale} \textit{pre-value-sterm-pred} = \textit{value-pred} +
\begin{itemize}
  \item fixes \textit{S}
  \item assumes \textit{value-to-sterm}: \textit{pred} \textit{v} \Rightarrow \textit{S} \ (\textit{value-to-sterm} \textit{v})
\end{itemize}
\begin{begin}
\textbf{corollary} \textit{value-to-sterm-env}:
\begin{itemize}
  \item assumes \textit{fmpred} (\lambda \cdot \textit{pred}) \textit{\Gamma}
  \item shows \textit{fmpred} (\lambda \cdot \textit{S}) (\textit{fmmap} \textit{value-to-sterm} \textit{\Gamma})
\end{itemize}
\textbf{unfolding} \textit{fmpred-map} \textbf{proof}

40
fix name v
assume fnlookup Γ name = Some v
with assms have pred v by (metis fmpredD)
thus S (value-to-sterm v) by (rule value-to-sterm)
qed

end

locale value-sterm-pred = value-pred + S: simple-syntactic-and S for S +
assumes const: \(\forall\) name. Q name \(\Rightarrow\) S (const name)
assumes abs: \(\forall\) Γ cs.
(\(\forall\) n v. fnlookup Γ n = Some v \(\Rightarrow\) pred v \(\Rightarrow\) S (value-to-sterm v)) \(\Rightarrow\)
fmpred (\(\lambda\)- pred) Γ \(\Rightarrow\)
P Γ cs \(\Rightarrow\)
S (Sabs (map (\(\lambda\) (pat, t). (pat, subst t (fmmap value-to-sterm (fndrop-fset (frees pat) Γ)))) cs))
begin
sublocale pre-value-sterm-pred
proof
fix v
assume pred v
then show S (value-to-sterm v)
proof (induction v)
case (Vconstr x1 x2)
show ?case
apply simp
unfolding S.list-comb
apply rule
apply (rule const)
using Vconstr by (auto simp: list-all-iff)
next
case (Vabs x1 x2)
show ?case
apply auto
apply (rule abs)
using Vabs
by (auto intro: fmran'1)
next
case (Vrecabs x1 x2 x3)
show ?case
apply auto
apply (rule abs)
using Vrecabs
by (auto simp: fnlookup-dom-iff fmpred-iff intro: fmran'1)
qed
qed
end
global-interpretation vwellformed:
value-stern-pred
  λ·. wellformed-clauses
  λ·. True
  λ·. True
wellformed
defines vwellformed = vwellformed.pred
proof (standard, goal-cases)
case (2 Γ cs)
hence cs ≠ []
  by simp

moreover have wellformed (subst rhs (fmdrop-fset (frees pat) (fmmap value-to-sterm Γ))))
  if (pat, rhs) ∈ set cs for pat rhs
  proof –
    show ?thesis
      apply (rule subst-wellformed)
      subgoal using 2 that by (force simp: list-all-iff)
      apply (rule fmpred-drop-fset)
      using 2 by auto
    qed

moreover have distinct (map (fst o (λ(pat, t). (pat, subst t (fmmap value-to-sterm (fmdrop-fset (frees pat) Γ)))))) cs)
  apply (subst map-cong[OF refl, where g = fst])
  using 2 by auto

ultimately show ?case
  using 2 by (auto simp: list-all-iff)
qed (auto simp: const-sterm-def)

abbreviation wellformed-venv ≡ fmpred (λ·. vwellformed)

global-interpretation vclosed:
value-stern-pred
  λΓ cs. list-all (λ(pat, t). closed-except t (fdom Γ ∪ | frees pat)) cs
  λ·. True
  λ·. True
closed
defines vclosed = vclosed.pred
proof (standard, goal-cases)
case (2 Γ cs)
  show ?case
    apply (simp add: list-all-iff case-prod-two Stem.closed-except-simps)
    apply safe
    apply (subst closed-except-def)
    apply (subst subst-frees)
apply simp
subgoal
  apply (rule fmpred-drop-fset)
  apply (rule fmpredI)
  apply (rule 2)
  apply assumption
  using 2 by auto
subgoal
  using 2 by (auto simp: list-all-iff closed-except-def)
done
qed simp

abbreviation closed-venv ≡ fmpred (λ-. vclosed)

context pre-constants begin

sublocale vwelldefined:
  value-sterm-pred
    λ- cs. list-all (λ(-, t). welldefined t) cs
  λname. name |∈| C
  λdom. dom |⊆| heads
  welldefined
defines vwelldefined = vwelldefined.pred
proof (standard, goal-cases)
  case (2 Γ cs)
  note fset-of-list-map[simp del]
  show ?case
    apply simp
    apply (rule ffUnion-least)
    apply (rule fBallI)
    apply (subst (asm) fset-of-list-elem)
    apply simp
    apply (erule imageE)
    apply (simp add: case-prod-twice)
    subgoal for - x
      apply (cases x)
      apply simp
      apply (rule substconsts)
    subgoal
      using 2 by (fastforce simp: list-all-iff)
    subgoal
      apply simp
      apply (rule fmpred-drop-fset)
      unfolding fmpred-map
      apply (rule fmpredI)
      using 2 by auto
    done
  done

43
lemmas vwelldefined-alt-def = vwelldefined.pred-alt-def
end

declare pre-constants.vwelldefined-alt-def[code]

context constructors begin

sublocale vconstructor-value:
  pre-value-sterm-pred
  λ_. True
  λname. name |∈| C
  λ_. True
  is-value

defines vconstructor-value = vconstructor-value.pred

proof
  fix v
  assume value-pred.pred (λ_. True) (λname. name |∈| C) (λ_. True) v
  then show is-value (value-to-sterm v)
    proof (induction v)
      case (Vconstr name vs)
      hence list-all is-value (map value-to-sterm vs)
        by (fastforce simp: list-all-iff value-pred.pred-alt-def)
      show ?case
        unfolding value-to-sterm.simps
        apply (rule is-value.constr)
        apply fact
        using Vconstr by (simp add: value-pred.pred-alt-def)
    qed (auto simp: disjnt-def intro: is-value.intros)
  qed

lemmas vconstructor-value-alt-def = vconstructor-value.pred-alt-def

abbreviation vconstructor-value-env ≡ fnpred (λ_. vconstructor-value)

definition vconstructor-value-rs :: vrule list ⇒ bool where
  vconstructor-value-rs rs ←→
  list-all (λ(_, rhs). vconstructor-value rhs) rs ∧
  fdisjnt (fset-of-list (map fst rs)) C

end

declare constructors.vconstructor-value-alt-def[code]
declare constructors.vconstructor-value-rs-def[code]

context pre-constants begin
sublocale not-shadows-vconsts:
value-sterm-pred
λ- cs. list-all (λ(pat, t). fdissjnt all-consts (frees pat) ∧ ¬ shadows-consts t) cs
λ. True
λ. True
λt. ¬ shadows-consts t
defines not-shadows-vconsts = not-shadows-vconsts.pred
proof (standard, goal-cases)
case (2 Γ cs)
show ?case
apply (simp add: list-all-iff list-ex-iff case-prod-twice)
apply (rule ballI)
subgoal for x
apply (cases x, simp)
apply (rule conjI)
subgoal
using 2 by (force simp: list-all-iff)
apply (rule subst-shadows)
subgoal
using 2 by (force simp: list-all-iff)
apply simp
apply (rule fnmpred-drop-fset)
apply (rule fnmpredI)
using 2 by auto
done
qed (auto simp: const-sterm-def app-sterm-def)

lemmas not-shadows-vconsts-alt-def = not-shadows-vconsts.pred-alt-def

abbreviation not-shadows-vconsts-env ≡ fnmpred (λ- s. not-shadows-vconsts s)

end

declare pre-constants.not-shadows-vconsts-alt-def[code]

fun term-to-value :: sterm ⇒ value where
term-to-value t =
(case strip-comb t of
  (Sconst name, args) ⇒ Vconstr name (map term-to-value args)
  | (Sabs cs, []) ⇒ Vabs cs fmempty)

lemma (in constructors) term-to-value-to-sterm:
  assumes is-value t
  shows value-to-sterm (term-to-value t) = t
  using assms proof induction
case (constr vs name)
  have map (value-to-sterm o term-to-value) vs = map id vs
  proof (rule list.map-cong0, unfold comp-apply id-apply)
fix v
assume v ∈ set vs
with constr show value-to-term (term-to-value v) = v
  by (simp add: list-all-iff)
qed
thus ?case
  apply (simp add: strip-list-comb-const)
  apply (subst const-sterm-def)
  by simp
qed simp

lemma vmatch-dom:
  assumes vmatch pat v = Some env
  shows fmdom env = patvars pat
using assms proof (induction pat v arbitrary: env rule: vmatch-induct)
case (constr name ps name' vs)
hence
  map-option (foldl (++) fnempty) (those (map2 vmatch ps vs)) = Some env
  name = name' length ps = length vs
  by (auto split: if-splits)
then obtain envs where
  env = foldl (++) fnempty envs map2 vmatch ps vs
  = map Some envs
  by (blast dest: those-someD)
moreover have fset-of-list (map fmdom envs) = fset-of-list (map patvars ps)
proof safe
fix names
  assume names ∈ fset-of-list (map fmdom envs)
hence names ∈ set (map fmdom envs)
  unfolding fset-of-list-elem .
then obtain env where
  env ∈ set envs names = fmdom env
  by auto
hence Some env ∈ set (map2 vmatch ps vs)
  unfolding (map2 - - - = -) by simp
then obtain p v where
  p ∈ set ps v ∈ set vs vmatch p v = Some env
  by (auto elim: map2-elemE)
moreover hence fmdom env = patvars p
  using constr by fastforce
ultimately have names ∈ set (map patvars ps)
  unfolding (names = -) by simp
thus names ∈ fset-of-list (map patvars ps)
  unfolding fset-of-list-elem .
next
fix names
  assume names ∈ fset-of-list (map patvars ps)
hence names ∈ set (map patvars ps)
  unfolding fset-of-list-elem .
then obtain p where
  p ∈ set ps names = patvars p
  by auto
then obtain $v$ where $v \in \text{set } v_{\text{match}}$ $p$ $v \in \text{set } (\text{map2 } v_{\text{match}} \ p \ vs)$
using $\langle \text{length } ps = \text{length } vs \rangle$ by (auto elim! : map2-elemE1)
then obtain $\text{env}$ where $\text{env} \in \text{set } \text{envs } v_{\text{match}}$ $p$ $v$
unfolding $\langle \text{map2 } v_{\text{match}} \ p \ vs = \emptyset \rangle$ by auto
moreover hence $\text{fdom } \text{env} = \text{patvars } p$
using constr $\langle \text{name } = \text{name’} \rangle$ $\langle \text{length } ps = \text{length } vs \rangle$ $\langle p \in \text{set } ps \rangle$ $\langle v \in \text{set } vs \rangle$
by fastforce
ultimately have $\text{names } \in \text{set } (\text{map } \text{fdom } \text{envs})$
unfolding $\langle \text{names } = \emptyset \rangle$ by auto
thus $\text{names } \in \text{fset-of-list } (\text{map } \text{fdom } \text{envs})$
unfolding fset-of-list-elem .
qed

ultimately show $\text{?case}$
by (auto simp: fmdom-foldl-add)
qed auto

fun $v_{\text{find-match}} : \text{sclauses } \Rightarrow \text{value } \Rightarrow ((\text{name }, \text{value}) \ \text{fnmap } \times \ \text{term } \times \ \text{terrm})$
option where
$v_{\text{find-match}} \ [ ] = \text{None}$ | $v_{\text{find-match}} \ ((\text{pat }, \text{rhs}) \ # \ cs) =$
(case $v_{\text{match}} (\text{mk-pat } \text{pat}) \ t$ of
  Some $\text{env} \Rightarrow \text{Some } (\text{env}, \text{pat}, \text{rhs})$
| None $\Rightarrow v_{\text{find-match}} \ cs \ t$)

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

inductive $\text{veq-structure } : \text{value } \Rightarrow \text{value } \Rightarrow \text{bool}$ where
abs-abs: $\text{veq-structure } (\text{Vabs } - -) (\text{Vabs } - -)$ |
recabs-recabs: $\text{veq-structure } (\text{Vrecabs } - - -) (\text{Vrecabs } - - -)$ |
constr-constr: $\text{list-all2 } \text{veq-structure } \text{ts } \text{us } \Rightarrow \text{veq-structure } (\text{Vconstr } \text{name } \text{ts})$
  $(\text{Vconstr } \text{name } \text{us})$

lemma $\text{veq-structure}$-simps[code, simp]:
  $\text{veq-structure } (\text{Vabs } \text{cs}_1 \ \Gamma_1) (\text{Vabs } \text{cs}_2 \ \Gamma_2)$ |
  $\text{veq-structure } (\text{Vrecabs } \text{css}_1 \ \text{name}_1 \ \Gamma_1) (\text{Vrecabs } \text{css}_2 \ \text{name}_2 \ \Gamma_2)$ |
  $\text{veq-structure } (\text{Vconstr } \text{name}_1 \ \text{ts}) (\text{Vconstr } \text{name}_2 \ \text{us}) \ \Rightarrow \ \text{name}_1 = \text{name}_2 \ \land$
  $\text{list-all2 } \text{veq-structure } \text{ts } \text{us}$
by (auto intro: veq-structure.intros elim: veq-structure.cases)

lemma $\text{veq-structure}$-refl[simp]: $\text{veq-structure } \ t \ \text{t}$
by (induction $t$) (auto simp: list.rel-refl-strong)

global-interpretation $\text{vno-abs} : \text{value-pred } \lambda - . \ \text{False } \lambda -. \ \text{True } \lambda -. \ \text{False}$
defines \( \text{vno-abs} = \text{vno-abs.pred} \).

lemma \( \text{veq-structure-eq-left} \):
  assumes \( \text{veq-structure} \ t \ u \ \text{vno-abs} \ t \)
  shows \( t = u \)
  using assms proof (induction rule: \( \text{veq-structure.induct} \))
  case \( (\text{constr-constr} \ ts \ us \ \text{name}) \)
  have \( ts = us \) if \( \text{list-all} \ \text{vno-abs} \ ts \)
    using constr-constr.IH that
    by induction auto
  with \( \text{constr-constr} \) show \( ?\text{case} \)
    by auto
qed auto

lemma \( \text{veq-structure-eq-right} \):
  assumes \( \text{veq-structure} \ t \ u \ \text{vno-abs} \ u \)
  shows \( t = u \)
  using assms proof (induction rule: \( \text{veq-structure.induct} \))
  case \( (\text{constr-constr} \ ts \ us \ \text{name}) \)
  have \( ts = us \) if \( \text{list-all} \ \text{vno-abs} \ us \)
    using constr-constr.IH that
    by induction auto
  with \( \text{constr-constr} \) show \( ?\text{case} \)
    by auto
qed auto

fun \( \text{vmatch} \ ' :: \ \text{pat} \Rightarrow \ \text{value} \Rightarrow (\text{name}, \ \text{value}) \ \text{fmap} \ \text{option} \ \text{where} \)
\( \text{vmatch} \ '|(\text{Patvar} \ \text{name}) \ v = \text{Some} \ (\text{fmap-of-list} \ \{(\text{name}, \ v)\}) \ |
\text{vmatch} \ '|(\text{Patconstr} \ \text{name} \ \text{ps}) \ v =
  \text{(case} \ v \ \text{of}\ 
  V\text{constr} \ \text{name}' \ \text{vs} \Rightarrow
  \ (\text{if} \ \text{name} = \text{name}' \ \land \ \text{length} \ \text{ps} = \text{length} \ \text{vs} \ \text{then}
    \ \text{map-option} \ (\text{foldl} \ (\++f) \ \text{fmempty}) \ (\text{those} \ (\text{map2} \ \text{vmatch} \ ')' \ \text{ps} \ \text{vs}))
  \ \text{else}
  \ \text{None})
  | - \Rightarrow \text{None})

lemma \( \text{vmatch-vmatch}'-eq: \text{vmatch} \ p \ v = \text{vmatch} \ ' \ p \ v \)
proof (induction rule: \( \text{vmatch.induct} \))
  case \( (2 \ \text{name} \ \text{ps} \ \text{name}' \ \text{vs}) \)
  then show \( ?\text{case} \)
    apply auto
    apply (rule map-option-cong\([\text{OF - refl}]\))
    apply (rule arg-cong\([\text{where} \ f = \text{those}]\))
    apply (rule map2-cong\([\text{OF refl refl}]\))
    apply blast
    done
qed auto
locale value-struct-rel =
  fixes Q :: value ⇒ value ⇒ bool
  assumes Q-impl-struct: Q t1 t2 ⇒ veq-structure t1 t2
  assumes Q-def[simp]: Q (Vconstr name ts) (Vconstr name’ us) ←→ name = name’ ∧ list-all2 Q ts us
begin

lemma eq-left: Q t u ⇒ vno-abs t ⇒ t = u
using Q-impl-struct by (metis veq-structure-eq-left)

lemma eq-right: Q t u ⇒ vno-abs u ⇒ t = u
using Q-impl-struct by (metis veq-structure-eq-right)

context begin

private lemma vmatch'-rel:
  assumes Q t1 t2
  shows rel-option (fmrel Q) (vmatch’ p t1) (vmatch’ p t2)
using assms(1) proof (induction p arbitrary: t1 t2)
case (Patconstr name ps)
with Q-impl-struct have veq-structure t1 t2
  by blast
thus ?case
proof (cases rule: veq-structure.cases)
case (constr-constr ts us name’)
 {
  assume length ps = length ts

  have list-all2 (rel-option (fmrel Q)) (map2 vmatch’ ps ts) (map2 vmatch’ ps us)
  using ⟨list-all2 veq-structure ts us⟩ Patconstr ⟨length ps = length ts⟩
  unfolding ⟨t1 = ⊥⟩ ⟨t2 = ⊥⟩
  proof (induction arbitrary: ps)
  case (Cons t ts u us ps0)
  then obtain p ps where ps0 = p # ps
  by (cases ps0) auto

  have length ts = length us
  using Cons by (auto dest: list-all2-lengthD)
  hence Q t u
  using ⟨Q (Vconstr name’ (t # ts)) (Vconstr name’ (u # us))⟩
  by (simp add: list-all-iff)
  hence rel-option (fmrel Q) (vmatch’ p t) (vmatch’ p u)
  using Cons unfolding ⟨ps0 = ⊥⟩ by simp

  moreover have list-all2 (rel-option (fmrel Q)) (map2 vmatch’ ps ts)

moreover
apply (rule Cons)
subgoal
apply (rule Cons)
unfolding \( ps0 = \cdot \) apply simp
by assumption
subgoal
using \( \langle Q \ (V\text{constr name'} (t \# ts)) \ (V\text{constr name'} (u \# us)) \rangle \langle \text{length } ts = \text{length } us \rangle \)
by (simp add: list-all-iff)
subgoal
using \( \langle \text{length } ps0 = \text{length } (t \# ts) \rangle \)
unfolding \( ps0 = \cdot \) by simp
done
ultimately show ?case
unfolding \( ps0 = \cdot \)
by auto
qed auto

hence rel-option \( \langle \text{list-all2 } (fmrel Q) \rangle \langle \text{those } (map2 \ vmatch' ps ts) \rangle \langle \text{those } (map2 \ vmatch' ps us) \rangle \)
by (rule rel-funD[OF those-transfer])

have
  rel-option \( fmrel Q \)
  \( \langle \text{map-option } (foldl (++\ f) fnempty) \langle \text{those } (map2 \ vmatch' ps ts) \rangle \rangle \)
  \( \langle \text{map-option } (foldl (++\ f) fnempty) \langle \text{those } (map2 \ vmatch' ps us) \rangle \rangle \)
apply (rule rel-funD[OF rel-funD[OF option.map-transfer]])
apply transfer-prover
by fact
}

note * = this

have length ts = length us
using constr-constr by (auto dest: list-all2-lengthD)

thus ?thesis
unfolding \( t_1 = \cdot \) \( t_2 = \cdot \)
apply auto
apply (rule *)
by simp
qed auto

lemma vmatch-rel: \( Q \ t_1 \ t_2 \implies \text{rel-option } (fmrel Q) \ (vmatch p t_1) \ (vmatch p t_2) \)
unfolding vmatch-vmatch'-eq by (rule vmatch'-rel)

lemma vfind-match-rel:
assumes list-all2 (rel-prod (=) R) cs1 cs2
assumes Q t1 t2
shows rel-option (rel-prod (fmrel Q) (rel-prod (=) R)) (vfind-match cs1 t1) (vfind-match cs2 t2)
using assms(1) proof induction
case (Cons c1 c2 cs)
moreover obtain pat1 rhs1 where c1 = (pat1, rhs1) by fastforce
moreover obtain pat2 rhs2 where c2 = (pat2, rhs2) by fastforce
ultimately have pat1 = pat2 R rhs1 rhs2
  by auto
have rel-option (fmrel Q) (vmatch (mk-pat pat1) t1) (vmatch (mk-pat pat1) t2)
  by (rule vmatch-rel) fact
thus ?case
proof cases
  case None
  thus ?thesis
    unfolding ⟨c1 = ⊥⟩ ⟨c2 = ⊥⟩ ⟨pat1 = ⊥⟩
    using Cons by auto
next
  case (Some Γ1 Γ2)
  thus ?thesis
    unfolding ⟨c1 = ⊥⟩ ⟨c2 = ⊥⟩ ⟨pat1 = ⊥⟩
    using ⟨R rhs1 rhs2⟩
    by auto
qed
qed simp

lemmas vfind-match-rel' =
vfind-match-rel[ where R = (=) and cs1 = cs and cs2 = cs for cs, unfolded prod.rel-eq,
  OF list.rel-refl, OF refl]

end end

hide-fact vmatch-vmatch'-eq
hide-const vmatch'

global-interpretation veq-structure: value-struct-rel veq-structure
by standard auto

abbreviation env-eq where
env-eq ≡ fmrel (λv t. t = value-to-sterm v)

lemma env-eq-eq:
  assumes env-eq venv senv
  shows senv = fmmap value-to-sterm venv
proof (rule fmap-ext, unfold fmlookup-map)

fix name
from assms have rel-option (λv t. t = value-to-sterm v) (fmlookup venv name)
  (fmlookup senv name)
  by auto
thus fmlookup senv name = map-option value-to-sterm (fmlookup venv name)
  by cases auto
qed

context constructors begin

context begin

private lemma vmatch-eq0: rel-option env-eq (vmatch p v) (smatch′ p (value-to-sterm v))
proof (induction p v rule: vmatch-induct)
case (constr name ps name′ vs)
  have rel-option env-eq
    (map-option (foldl (++) Γ) (those (map2 vmatch ps vs)))
    (map-option (foldl (++) Γ′) (those (map2 smatch′ ps (map value-to-sterm vs))))
    if length ps = length vs and name = name′ and env-eq Γ Γ′ for Γ Γ′
    using that constr
  proof (induction arbitrary: Γ Γ′ rule: list-induct2)
case (Cons p ps v vs)
  hence rel-option env-eq (vmatch p v) (smatch′ p (value-to-sterm v))
    by auto
  thus ?case
  proof cases
    case (Some Γ1 Γ2)
    thus ?thesis
    apply (simp add: option.map-comp comp-def)
    apply (rule Cons)
    using Cons by auto
  qed simp
  qed fastforce
thus ?case
apply (auto simp: strip-list-comb-const)
apply (subst const-sterm-def, simp)+
done
qed auto

corollary vmatch-eq:
  assumes linear p vconstructor-value v
  shows rel-option env-eq (vmatch (mk-pat p) v) (match p (value-to-sterm v))
  using assms
by (metis smatch-smatch'eq vmatch-eq0 vconstructor-value.value-to-sterm)

end

end

abbreviation match-related where
match-related ≡ (λ(Γ₁, pat₁, rhs₁) (Γ₂, pat₂, rhs₂). rhs₁ = rhs₂ ∧ pat₁ = pat₂ ∧ env-eq Γ₁ Γ₂)

lemma (in constructors) find-match-eq:
assumes list-all (linear o fst) cs vconstructor-value v
shows rel-option match-related (vfind-match cs v) (find-match cs (value-to-sterm v))
using assms proof (induct cs)
case (Cons c cs)
then obtain p t where c = (p, t) by fastforce
hence rel-option env-eq (vmatch (mk-pat p) v) (match p (value-to-sterm v))
using Cons by (fastforce intro: vmatch-eq)
thus ?case
using Cons unfolding (c = -)
by cases auto
qed auto

inductive erelated :: value ⇒ value ⇒ bool (- ≈e -) where
constr: list-all2 erelated ts us ⇒ Vconstr name ts ≈e Vconstr name us |
abs: fnrel-on-fset (ids (Sabs cs)) erelated Γ₁ Γ₂ ⇒ Vabs cs Γ₁ ≈e Vabs cs Γ₂ |
rec-abs:
  pred-fmap (λcs. fnrel-on-fset (ids (Sabs cs)) erelated Γ₁ Γ₂) css ⇒
    Vrecabs css name Γ₁ ≈e Vrecabs css name Γ₂

code-pred erelated .

global-interpretation erelated: value-struct-rel erelated
proof
  fix v₁ v₂
  assume v₁ ≈e v₂
  thus veq-structure v₁ v₂
    by induction (auto intro: list.rel-mono-strong)
next
  fix name name' and ts us :: value list
  show Vconstr name ts ≈e Vconstr name' us ←→ (name = name' ∧ list-all2 erelated ts us)
    by (auto intro: erelated.intros elim: erelated.cases)
qed

lemma erelated-refl[intro]: t ≈e t
proof (induction t)
case Vrecabs

53
thus \( ?\text{case} \)

\textbf{apply} (\textit{auto intro: erelated.intros fmpredI fmrel-on-fset-refl-strong})
\textbf{apply} (\textit{auto intro: fmran’I})
\textbf{done}

\textbf{qed} (\textit{auto intro: erelated.intros list.rel-refl-strong fmrel-on-fset-refl-strong fmran’I})

\textbf{export-code}

\textit{value-to-term vmatch vwellformed vclosed erelated-i-i pre-constants.vwelldefined constructors.vconstructor-value-rs pre-constants.not-shadows-vconsts term-to-value vfind-match veq-structure vno-abs}

\textbf{checking Scala}

\textbf{end}
Chapter 2

A smaller version of CakeML: *CupCakeML*

theory Doc-CupCake
imports Main
begin

end

2.1 CupCake environments

theory CupCake-Env
imports ../Utils/CakeML-Utils
begin

fun cake-no-abs :: v ⇒ bool where
cake-no-abs (Conv - vs) ←→ list-all cake-no-abs vs |
cake-no-abs - ←→ False

fun is-cupcake-pat :: Ast.pat ⇒ bool where
is-cupcake-pat (Ast.Pvar -) ←→ True |
is-cupcake-pat (Ast.Pcon (Some (Short -)) xs) ←→ list-all is-cupcake-pat xs |
is-cupcake-pat - ←→ False

fun is-cupcake-exp :: exp ⇒ bool where
is-cupcake-exp (Ast.Var (Short -)) ←→ True |
is-cupcake-exp (Ast.App oper es) ←→ oper = Ast.Opapp ∧ list-all is-cupcake-exp es |
is-cupcake-exp (Ast.Con (Some (Short -)) es) ←→ list-all is-cupcake-exp es |
is-cupcake-exp (Ast.Fun - e) ←→ is-cupcake-exp e |
is-cupcake-exp (Ast.Mut e es) ←→ is-cupcake-exp e ∧ list-all (λ(p, e). is-cupcake-pat p ∧ is-cupcake-exp e) es ∧ cake-linear-clauses es |
is-cupcake-exp - ←→ False
abbreviation cupcake-clauses :: (Ast.pat × exp) list ⇒ bool where
  cupcake-clauses ≡ list-all (λ(p, e). is-cupcake-pat p ∧ is-cupcake-exp e)

fun cupcake-c-ns :: c-ns ⇒ bool where
cupcake-c-ns (Bind cs mods) ←→
 .mods = [] ∧ list-all (λ(_, _, tid). case tid of TypeId (Short _) ⇒ True | _ ⇒ False) cs

locale cakeml-static-env =
  fixes static-cenv :: c-ns
  assumes static-cenv: cupcake-c-ns static-cenv
begin

definition empty-sem-env :: v sem-env where
  empty-sem-env = (| sem-env.v = nsEmpty, sem-env.c = static-cenv |

lemma v-of-empty-sem-env[simp]: sem-env.v empty-sem-env = nsEmpty
unfolding empty-sem-env-def by simp

lemma c-of-empty-sem-env[simp]: c empty-sem-env = static-cenv
unfolding empty-sem-env-def by simp

fun is-cupcake-value :: SemanticPrimitives.v ⇒ bool and
  is-cupcake-all-env :: all-env ⇒ bool where
  is-cupcake-value (Conv (Some (-, TypeId (Short -))) vs) ←→ list-all is-cupcake-value vs |
  is-cupcake-value (Closure env - e) ←→ is-cupcake-exp e ∧ is-cupcake-all-env env |
  is-cupcake-value (Recclosure env vs -) ←→ list-all (λ(-, -, e). is-cupcake-exp e) es
  ∧ is-cupcake-all-env env |
  is-cupcake-value - ←→ False |
  is-cupcake-all-env (| sem-env.v = Bind v0 [], sem-env.c = c0 |) ←→ c0 = static-cenv
  ∧ list-all (is-cupcake-value ○ snd) v0 |
  is-cupcake-all-env - ←→ False

lemma is-cupcake-all-envE:
  assumes is-cupcake-all-env env
  obtains v c where env = (| sem-env.v = Bind v [], sem-env.c = c |) c = static-cenv list-all (is-cupcake-value ○ snd) v
  using assms
  by (auto elim!: is-cupcake-all-env.elims)

fun is-cupcake-ns :: v-ns ⇒ bool where
  is-cupcake-ns (Bind v0 []) ←→ list-all (is-cupcake-value ○ snd) v0 |
  is-cupcake-ns - ←→ False

lemma is-cupcake-nsE:
  assumes is-cupcake-ns ns
  obtains v where ns = Bind v [] list-all (is-cupcake-value ○ snd) v
  using assms by (rule is-cupcake-ns.elims)
lemma is-cupcake-all-envD:
  assumes is-cupcake-all-env env
  shows is-cupcake-ns (sem-env.v env) cupcake-c-ns (c env)
  using assms static-cenv by (auto elim!: is-cupcake-all-envE)

lemma is-cupcake-all-envI:
  assumes is-cupcake-ns (sem-env.v env) sem-env.c env = static-cenv
  shows is-cupcake-all-env env
  using assms static-cenv
  apply (cases env)
  apply simp
  subgoal for v c
    apply (cases v)
    apply simp
    subgoal for x1 x2
      by (cases x2) auto
  done
done

end

end

2.2 CupCake semantics

theory CupCake-Semantics
imports
  CupCake-Env
  CakeML.Matching
  CakeML.Big-Step-Unclocked-Single
begin

fun cupcake-nsLookup :: (′m,′n,′v)namespace ⇒ ′n ⇒ ′v option
  where
  cupcake-nsLookup (Bind v1 -) n = map-of v1 n

lemma cupcake-nsLookup-eq[simp]: nsLookup ns (Short n) = cupcake-nsLookup ns n
  by (cases ns) auto

fun cupcake-pmatch :: ((string),(string),(nat+tid-or-exn))namespace ⇒ pat ⇒ v
  ⇒(string+v)list ⇒((string+v)list)match-result
  where
  cupcake-pmatch cenv (Pvar x) v0 env = Match ((x, v0)# env) |
  cupcake-pmatch cenv (Pcon (Some (Short n)) ps) (Conv (Some (n', t')) vs) env
  =
    (case cupcake-nsLookup cenv n of
     Some (l, t) =>>
       if same-tid t t' /
         (List.length ps = l)
       then
       if same-ctor (n, t) (n', t') then

57
Matchingfold2 \((\lambda p v m. \text{case } m \text{ of})\)
\[ \begin{align*}
  \text{Match env } &\Rightarrow \text{cupcake-pmatch cenv } p \ v \ \text{env} \\
  | \ m &\Rightarrow m
\end{align*} \]
else
No-match
else
Match-type-error
| \ - - \Rightarrow Match-type-error \)
cupcake-pmatch cenv \ - - - = Match-type-error

fun cupcake-match-result :: - \Rightarrow v \Rightarrow (pat \ exp) \ list \Rightarrow v \Rightarrow (\exp \times pat \times (char \ list \times v)) \ list \Rightarrow v \Rightarrow (\exp \times pat \times (char \ list \times v)) \ list \Rightarrow v \Rightarrow (\exp \times pat \times (char \ list \times v)) \ list
where
cupcake-match-result - - [] err-v = Rerr (Rraise err-v) |
cupcake-match-result cenv v0 ((p, e) \# pes) err-v =
(if distinct (pat-bindings p []) then
  (case cupcake-pmatch cenv p v0 [] of
    Match env' \Rightarrow Rval (e, p, env') |
    No-match \Rightarrow \text{cupcake-match-result cenv } v0 \ pes \ err-v |
    Match-type-error \Rightarrow Rerr (Rabort Rtype-error))
else
  Rerr (Rabort Rtype-error))

lemma cupcake-match-resultE:
assumes cupcake-match-result cenv v0 pes err-v = Rval (e, p, env')
obtains init rest
where pes = init @ (p, e) \# rest
   and distinct (pat-bindings p [])
   and list-all \((\lambda (p, e)). \text{cupcake-pmatch cenv } p \ v0 [] = \text{No-match} \land \text{distinct} (pat-bindings p [])\)) init
   and cupcake-pmatch cenv p v0 [] = Match env'
using assms
proof (induction pes)
case (Cons pe pes)
obtain p0 e0 where pe = (p0, e0)
by fastforce

show thesis
proof (cases distinct (pat-bindings p0 []))
case True
thus ?thesis
proof (cases cupcake-pmatch cenv p0 v0 [])
case No-match
show ?thesis
proof (rule Cons)
fix init rest
assume pes = init @ (p, e) \# rest
assume list-all \((\lambda (p, e)). \text{cupcake-pmatch cenv } p \ v0 [] = \text{No-match} \land \text{distinct} (pat-bindings p [])\)) init
assume distinct (pat-bindings p [])
assume \( \text{cupcake-pmatch } cenv \ p \ v0 \ [] = \text{Match } env' \)

moreover have \( pe \# \ pes = ((p0, e0) \# \ init) \otimes (p, e) \# \ rest \)
unfolding \( \langle pes \rightarrow pe \rightarrow \rangle \) by simp

moreover have \( \text{list-all } (\lambda(p, e). \text{cupcake-pmatch } cenv \ p \ v0 \ []) = \)
\( \text{No-match } \land \text{distinct } (\text{pat-bindings } p \ []) \)\
apply auto
subgoal using True by simp
subgoal using \( \langle \text{list-all } - \rangle \) by simp
done

moreover have \( \text{distinct } (\text{pat-bindings } p \ []) \)
by fact

ultimately show \( \text{thesis} \)
using Cons by blast

next
show \( \text{cupcake-match-result } cenv \ v0 \ pes \ err-v = \text{Rval } (e, p, env') \)
using Cons No-match True unfolding \( \langle pe \rightarrow \rangle \) by auto
qed

next
case \( \text{Match} \)
with Cons show \( \text{?thesis} \)
using True unfolding \( \langle pe \rightarrow \rangle \) by force

next
case \( \text{Match-type-error} \)
with Cons show \( \text{?thesis} \)
using True unfolding \( \langle pe \rightarrow \rangle \) by force
qed

next
case False
hence False
using Cons unfolding \( \langle pe \rightarrow \rangle \) by force
thus \( \text{?thesis} .. \)
qed

qed simp

lemma \( \text{cupcake-pmatch-eq:} \)
\( \text{is-cupcake-pat } pat \Rightarrow \text{pmatch-single } envC \ s \ pat \ v0 \ env = \text{cupcake-pmatch } envC \)
\( \text{pat } v0 \ env \)
proof (induct rule: \text{pmatch-single.induct})
case 4
from \( \text{is-cupcake-pat.elims(2)][OF 4(2)]} \) show \( \text{?case} \)
proof cases
case 2
then show \( \text{?thesis} \)
using \( \langle 1 \rangle \) apply –
apply simp
apply (auto split: option.splits match-result.splits)
apply (rule Matching.fold2-cong)
  apply (auto simp: fun-eq-iff split: match-result.splits)
apply (metis in-set-conv-decomp-last list.pred-inject(2) list.all-append)
done
qed simp
qed auto

lemma cupcake-match-result-eq:
  cupcake-clauses pes \Rightarrow
  match-result env s v pes err-v =
  map-result (\lambda(e, -, env'). (e, env')) id (cupcake-match-result (c env) v pes err-v)
by (induction pes) (auto split: match-result.splits simp: cupcake-pmatch-eq pmatch-single-equiv)

context cakeml-static-env begin

lemma cupcake-nsBind-preserve:
  is-cupcake-ns ns \Rightarrow
  is-cupcake-value v0 \Rightarrow
  is-cupcake-ns (nsBind k v0 ns)
by (cases ns) (auto elim: is-cupcake-ns.elims)

lemma cupcake-build-rec-preserve:
  assumes
  is-cupcake-all-env cl-env
  is-cupcake-ns env
  list-all (\lambda(-, -, e)). is-cupcake-exp e)
  fs
  shows
  is-cupcake-ns (build-rec-env fs cl-env env)
proof –
  have
  is-cupcake-ns (foldr (\lambda(f, -) env'. nsBind f (Recclosure cl-env fs0 f) env'))
  fs env)
  if
  list-all (\lambda(-, -, e)). is-cupcake-exp e) fs0
  for
  fs0
  using
  (is-cupcake-ns env)
proof
  (induction fs arbitrary: env)
  case (Cons f fs)
  show ?case
  apply (cases f, simp)
  apply (rule cupcake-nsBind-preserve)
  apply (rule Cons.IH)
  apply (rule Cons)
  using that
  auto
  thus
  thesis
  unfolding
  build-rec-env-def
  using
  assms
  by (simp add: cond-case-prod-eta)
qed

lemma cupcake-v-update-preserve:
  assumes
  is-cupcake-all-env env
  is-cupcake-ns (f (sem-env.v env))

60
shows is-cupcake-all-env (sem-env.update-v f env)
using assms
by (metis is-cupcake-all-envE is-cupcake-nsE sem-env.collapse sem-env.record-simps(1) sem-env.record-simps(2) sem-env.sel(2))

lemma cupcake-nsAppend-preserve: is-cupcake-ns ns1 \implies is-cupcake-ns ns2 \implies is-cupcake-ns (nsAppend ns1 ns2)
by (auto elim!: is-cupcake-nsE is-cupcake-nsE)

lemma cupcake-alist-to-ns-preserve: list-all (is-cupcake-value ◦ snd) env \implies is-cupcake-ns (alist-to-ns env)
unfolding alist-to-ns-def by simp

lemma cupcake-opapp-preserve:
  assumes do-opapp vs = Some (env, e) list-all is-cupcake-value vs
  shows is-cupcake-all-env env is-cupcake-exp e
proof –
  obtain cl v0 where vs = [cl, v0]
  using assms
  by (cases vs rule: do-opapp.cases) auto
with assms have is-cupcake-value cl is-cupcake-value v0
  by auto

have is-cupcake-all-env env ∧ is-cupcake-exp e
using (do-opapp vs = _) proof (cases rule: do-opapp.cases)
case (closure env' n arg)
then show ?thesis
  using (is-cupcake-value cl) (is-cupcake-value v0): (vs = [cl, v0])
by (auto intro: cupcake-v-update-preserve cupcake-nsBind-preserve dest:is-cupcake-all-envD(1))
next
case (recclosure env' funs name n)
hence is-cupcake-all-env env'
  using (is-cupcake-value cl) (vs = [cl, v0]) by simp
have (name, n, e) ∈ set funs
  using recclosure by (auto dest: map-of-SomeD)
hence is-cupcake-exp e
  using (is-cupcake-value cl) (vs = [cl, v0]) recclosure
by (auto simp: list-all-iff)
thus ?thesis
  using (is-cupcake-all-env env'5 (is-cupcake-value cl) (is-cupcake-value v0) vs = [cl, v0]) reclosure
  unfolding (env = _)
  using cupcake-build-rec-preserve cupcake-nsBind-preserve cupcake-v-update-preserve
is-cupcake-all-envD(1)
  by auto
qed

thus is-cupcake-all-env env is-cupcake-exp e
context begin

lemma cup-pmatch-list-length-neq:
  \begin{align*}
  \text{length } vs \neq \text{length } ps \implies \text{Matching} \cdot \text{fold2}(\lambda p \ v \ m. \ \text{case } m \ of}
  & \quad \text{Match } env \Rightarrow \text{cupcake-pmatch } cenv \ p \ v \ env \\
  & \quad \mid \ m \Rightarrow m) \ \text{Match-type-error } ps \ vs \ m = \text{Match-type-error}
  \end{align*}
by (induction ps vs arbitrary; rule:List.list-induct2') auto

lemma cup-pmatch-list-nomatch:
  \begin{align*}
  \text{length } vs = \text{length } ps \implies \text{Matching} \cdot \text{fold2}(\lambda p \ v \ m. \ \text{case } m \ of}
  & \quad \text{Match } env \Rightarrow \text{cupcake-pmatch } cenv \ p \ v \ env \\
  & \quad \mid \ m \Rightarrow m) \ \text{Match-type-error } ps \ vs \ \text{No-match} = \text{No-match}
  \end{align*}
by (induction ps vs rule:List.list-induct2') auto

lemma cup-pmatch-list-typerr:
  \begin{align*}
  \text{length } vs = \text{length } ps \implies \text{Matching} \cdot \text{fold2}(\lambda p \ v \ m. \ \text{case } m \ of}
  & \quad \text{Match } env \Rightarrow \text{cupcake-pmatch } cenv \ p \ v \ env \\
  & \quad \mid \ m \Rightarrow m) \ \text{Match-type-error } ps \ vs \ \text{Match-type-error} = \text{Match-type-error}
  \end{align*}
by (induction ps vs rule:List.list-induct2') auto

private lemma cupcake-pmatch-list-preserve:
  \begin{align*}
  \text{assumes } \forall p \ v \ env. \ p \in \text{set } ps \land v \in \text{set } vs \implies \text{list-all} \ (\text{is-cupcake-value} \circ \text{snd}) \ env \\
  \text{shows if-match} \ (\text{list-all} \ (\lambda a. \ \text{is-cupcake-value} \circ \text{snd} \ a)) \ (\text{Matching.} \text{fold2} \\
  (\lambda p \ v \ m. \ \text{case } m \ of}
  & \quad \text{Match } env \Rightarrow \text{cupcake-pmatch } cenv \ p \ v \ env \\
  & \quad \mid \ m \Rightarrow m) \ \text{Match-type-error } ps \ vs \ (\text{Match } env))
  \end{align*}
using assms proof (induction ps vs arbitrary; env rule:List.list-induct2')

apply (cases cupcake-pmatch cenv p v env)

by (cases cupcake-pmatch cenv p v env)

then show ?thesis

by (cases length ps = length vs) (auto simp:cup-pmatch-list-nomatch cup-pmatch-list-length-neq)

next

case Match-type-error

then show ?thesis

by (cases length ps = length vs) (auto simp:cup-pmatch-list-typerr cup-pmatch-list-length-neq)

next

case (Match env')

then have env': list-all (is-cupcake-value o snd) env'

using 4 by fastforce

then show ?thesis

apply (cases length ps = length vs)
using 4 Match by fastforce+

qed

qed (auto simp: comp-def)

private lemma cupcake-pmatch-preserve0:
  is-cupcake-pat pat \implies
  is-cupcake-value v0 \implies
  list-all (is-cupcake-value o snd) env \implies
  cupcake-c-ns envC \implies
  if-match (list-all (is-cupcake-value o snd)) (cupcake-pmatch envC pat v0 env)

proof (induction rule: cupcake-pmatch.induct)
  case (2 cenv n ps n' t' vs env)
  have p: p \in set ps \implies is-cupcake-pat p for p
    using 2 by (metis Ball-set is-cupcake-pat.simps(2))
  have v: v \in set vs \implies is-cupcake-value v for v
    using 2 by (metis Ball-set is-cupcake-value.elims(2) v.distinct(11) v.distinct(13) v.inject(2))
  show ?case
    by (auto intro!: cupcake-pmatch-list-preserve split:if-splits option.splits) (metis 2 p v)+

qed (auto split: option.splits if-splits elim: is-cupcake-pat.elims is-cupcake-value.elims)

lemma cupcake-pmatch-preserve:
  is-cupcake-pat pat \implies
  is-cupcake-value v0 \implies
  list-all (is-cupcake-value o snd) env \implies
  cupcake-c-ns envC \implies
  cupcake-pmatch envC pat v0 env = Match env' \implies
  list-all (is-cupcake-value o snd) env'
  by (metis if-match.simps(1) cupcake-pmatch-preserve0)+

end

lemma cupcake-match-result-preserve:
  cupcake-c-ns envC \implies
  cupcake-clauses pes \implies
  is-cupcake-value v \implies
  \text{if-rval} (\lambda(e, p, env'). is-cupcake-pat p \land is-cupcake-exp e \land list-all (is-cupcake-value o snd) env') env'
  (cupcake-match-result envC v pes err-v)
  apply (induction pes)
  apply (auto split: match-result.splits)
  apply (rule cupcake-pmatch-preserve)
  apply auto
  done

lemma static-cenv-lookup:
  assumes cupcake-nsLookup static-cenv i = Some (len, b)
  obtains name where b = TypeId (Short name)
using assms static-cenv
apply (cases static-cenv; cases b)
apply (auto simp: list-all-iff split: prod.splits tid-or-exn.splits id0.splits dest!: map-of-SomeD elim!: ballE allE)
using static-cenv
apply (auto simp: list-all-iff split: prod.splits tid-or-exn.splits id0.splits dest!: map-of-SomeD elim!: ballE allE)
done

lemma cupcake-build-conv-preserve:
  fixes v
  assumes list-all is-cupcake-value vs build-conv static-cenv (Some (Short i)) vs = Some v
  shows is-cupcake-value v
using assms
by (auto simp: build-conv.simps split: option.splits elim: static-cenv-lookup)

lemma cupcake-nsLookup-preserve:
  assumes is-cupcake-ns ns nsLookup ns n = Some v0
  shows is-cupcake-value v0
proof –
  obtain vs where list-all (is-cupcake-value ◦ snd) vs ns = Bind vs [] using assms
  by (auto elim: is-cupcake-ns.elims)
show ?thesis
proof (cases n)
  case (Short id)
  hence (id, v0) ∈ set vs using assms
  unfolding (ns = -) by (auto dest: map-of-SomeD)
  thus ?thesis using :list-all (is-cupcake-value ◦ snd) vs
  by (auto simp: list-all-iff)
next
  case Long
  hence nsLookup ns n = None unfolding (ns = -) by simp
  thus ?thesis using assms by auto
qed

Qed

corollary match-all-preserve:
  assumes cupcake-match-result cenv v0 pes err_v = Rval (e, p, env') cupcake-c-ns cenv
  assumes is-cupcake-value v0 cupcake-clauses pes
  shows list-all (is-cupcake-value ◦ snd) env' is-cupcake-exp e is-cupcake-pat p
proof –
  from assms obtain init rest
  where pes = init @ (p, e) ≠ rest and cupcake-pmatch cenv p v0 [] = Match
env' 
  by (elim cupcake-match-resultE)

hence \((p, e) \in \text{set pes}\)
  by simp

thus \(\text{is-cupcake-exp e is-cupcake-pat p}\)
  using assms by (auto simp: list-all-iff)

show \(\text{list-all (is-cupcake-value \circ \text{snd}) env'}\)
  by (rule cupcake-pmatch-preserve[where \(\text{env} = []\)] (fact | simp)+

qed end

fun list-all2-shortcircuit where
list-all2-shortcircuit \(P (x \neq xs) (y \neq ys) \iff (\text{case y of Rval - } \Rightarrow P x y \land \text{list-all2-shortcircuit P xs ys} | \text{Rerr - } \Rightarrow P x y) | \)
list-all2-shortcircuit \(P \ [ ] \ [ ] \iff \text{True |} \)
list-all2-shortcircuit \(P - - \iff \text{False} \)

lemma list-all2-shortcircuit-induct[consumes 1, case-names nil cons-val cons-err]:
  assumes list-all2-shortcircuit \(P xs ys\)
  assumes \(R \ [ ] \ [ ]\)
  assumes \(\land x xs y ys. P x (\text{Rval y}) \Rightarrow \text{list-all2-shortcircuit P xs ys} \Rightarrow R xs ys \Rightarrow R (x \neq xs) (\text{Rval y} \neq ys)\)
  assumes \(\land x xs y ys. P x (\text{Rerr y}) \Rightarrow R (x \neq xs) (\text{Rerr y} \neq ys)\)
  shows \(R xs ys\)
  using assms
  proof (induction \(P xs ys\) rule: list-all2-shortcircuit.induct)
    case (1 \(P x xs y ys\)
    thus \(?case\)
      by (cases y) auto
  qed auto

lemma list-all2-shortcircuit-mono[mono]:
  assumes \(R \leq Q\)
  shows \(\text{list-all2-shortcircuit R} \leq \text{list-all2-shortcircuit Q}\)

proof
  fix \(xs ys\)
  assume list-all2-shortcircuit \(R xs ys\)
  thus list-all2-shortcircuit \(Q xs ys\)
    using assms by (induction \(xs ys\) rule: list-all2-shortcircuit-induct) auto

qed

lemma list-all2-shortcircuit-weaken: list-all2-shortcircuit \(P xs ys \Rightarrow (\land xs ys. P xs ys \Rightarrow Q xs ys) \Rightarrow \text{list-all2-shortcircuit Q xs ys}\)
  by (metis list-all2-shortcircuit-mono predicate2I rev-predicate2D)

lemma list-all2-shortcircuit-rval[simp]:
  list-all2-shortcircuit \(P xs (\text{map Rval ys}) \iff \text{list-all2 (}\lambda x y. P x (\text{Rval y})\) xs ys\)

65
proof
  assume \( \text{lhs} \) thus \( \text{rhs} \)
  by (induction map \( \text{Rval \, ys}::\langle 'b, 'c \rangle \) result arbitrary: \( \text{ys \, rule: \, list-all2-shortcircuit-induct} \))
auto
next
assume \( \text{rhs} \) thus \( \text{lhs} \)
  by (induction rule: \( \text{list-all2-induct} \))
auto
qed

inductive \( \text{cupcake-evaluate-single :: \, all-env \Rightarrow \, exp \Rightarrow (v, v) \, result \Rightarrow \, bool \, where} \)

con1:
  do-con-check \( (c \, \text{env} \, cn \, (\text{length \, es})) \Rightarrow \)
  list-all2-shortcircuit \( (\text{cupcake-evaluate-single \, env}) \) (\( \text{rev \, es} \)) \( \Rightarrow \)
  sequence-result \( \text{rs} = \text{Rval \, vs} \Rightarrow \)
  build-conv \( (c \, \text{env}) \, cn \, (\text{rev \, vs}) = \text{Some \, v0} \Rightarrow \)
  cupcake-evaluate-single \( \text{env} \) \( (\text{Con \, cn \, es}) \) (\( \text{Rval \, v0} \)) |

con2:
  \(!\, \text{do-con-check} \, (c \, \text{env} \, cn \, (\text{List.length \, es}))) \Rightarrow \)
  cupcake-evaluate-single \( \text{env} \) \( (\text{Con \, cn \, es}) \) (\( \text{Rerr \, (Rabort \, Rtype-error)} \)) |

con3:
  do-con-check \( (c \, \text{env} \, cn \, (\text{List.length \, es}))) \Rightarrow \)
  list-all2-shortcircuit \( (\text{cupcake-evaluate-single \, env}) \) (\( \text{rev \, es} \)) \Rightarrow \)
  sequence-result \( \text{rs} = \text{Rerr \, err} \Rightarrow \)
  cupcake-evaluate-single \( \text{env} \) \( (\text{Con \, cn \, es}) \) (\( \text{Rerr \, err} \)) |

var1:
  \( \text{nsLookup} \, (\text{sem-env} \, v \, \text{env}) \, n = \text{Some \, v0} \Rightarrow \text{cupcake-evaluate-single \, env} \) \( (\text{Var \, n}) \) (\( \text{Rval \, v0} \)) |

var2:
  \( \text{nsLookup} \, (\text{sem-env} \, v \, \text{env}) \, n = \text{None} \Rightarrow \text{cupcake-evaluate-single \, env} \) \( (\text{Var \, n}) \) (\( \text{Rerr \, (Rabort \, Rtype-error)} \)) |

fn:
  \( \text{cupcake-evaluate-single \, env} \) \( (\text{Fun \, n \, e}) \) (\( \text{Rval \, (Closure \, \text{env} \, n \, e)} \)) |

app1:
  list-all2-shortcircuit \( (\text{cupcake-evaluate-single \, env}) \) (\( \text{rev \, es} \)) \Rightarrow \)
  sequence-result \( \text{rs} = \text{Rval \, vs} \Rightarrow \)
  do-opapp \( (\text{rev \, vs}) = \text{Some \, (\text{env}', e)} \Rightarrow \)
  cupcake-evaluate-single \( \text{env}' \) \( e \) bv \Rightarrow \)
  cupcake-evaluate-single \( \text{env} \) \( (\text{App \, Opapp \, es}) \) bv |

app3:
  list-all2-shortcircuit \( (\text{cupcake-evaluate-single \, env}) \) (\( \text{rev \, es} \)) \Rightarrow \)
  sequence-result \( \text{rs} = \text{Rerr \, err} \Rightarrow \)
  do-opapp \( (\text{rev \, vs}) = \text{None} \Rightarrow \)
  cupcake-evaluate-single \( \text{env} \) \( (\text{App \, Opapp \, es}) \) (\( \text{Rerr \, (Rabort \, Rtype-error)} \)) |

app6:
  list-all2-shortcircuit \( (\text{cupcake-evaluate-single \, env}) \) (\( \text{rev \, es} \)) \Rightarrow \)
  sequence-result \( \text{rs} = \text{Rerr \, err} \Rightarrow \)
  cupcake-evaluate-single \( \text{env} \) \( (\text{App \, op0 \, es}) \) (\( \text{Rerr \, err} \)) |

mat1:


cupcake-evaluate-single env e (Rval v0) \Rightarrow 
cupcake-match-result (c env) v0 pes Bindv = Rval (e', -, env') \Rightarrow 
cupcake-evaluate-single (env [] sem-env.v := nsAppend (alist-to-ns env') (sem-env.v env) |)) e' bv \Rightarrow 
cupcake-evaluate-single env (Mat e pes) bv |

mat1error:
cupcake-evaluate-single env e (Rval v0) \Rightarrow 
cupcake-match-result (c env) v0 pes Bindv = Rerr err \Rightarrow 
cupcake-evaluate-single env (Mat e pes) (Rerr err) |

mat2:
cupcake-evaluate-single env e (Rerr err) \Rightarrow 
cupcake-evaluate-single env (Mat e pes) (Rerr err)

context cakeml-static-env begin

context begin

private lemma cupcake-list-preserve0:
list-all2-shortcircuit (Ae r. cupcake-evaluate-single env e r \land (is-cupcake-all-env env \rightarrow is-cupcake-exp e \rightarrow if-rval is-cupcake-value r)) es rs \Rightarrow 
is-cupcake-all-env env \Rightarrow list-all is-cupcake-exp es \Rightarrow sequence-result rs = Rval vs \Rightarrow list-all is-cupcake-value vs
proof (induction es rs arbitrary: vs rule:list-all2-shortcircuit-induct)
case (cons-val - - - rs)
  thus ?case by (cases sequence-result rs) auto
qed auto

private lemma cupcake-single-preserve0:
cupcake-evaluate-single env e res \Rightarrow is-cupcake-all-env env \Rightarrow is-cupcake-exp e \Rightarrow if-rval is-cupcake-value res
proof (induction rule:cupcake-evaluate-single.induct)
case (con1 env cn es rs vs v0)
  then obtain tid where cn: cn = Some (Short tid) and list-all is-cupcake-exp (rev es)
    by (cases rule: is-cupcake-exp.cases [where x = Con cn es]) auto
  hence list-all is-cupcake-exp (rev vs) and c env = static-env
    using cupcake-list-preserve0 con1
    by (fastforce elim: is-cupcake-all-envE)+
then show ?case
  using cupcake-build-conv-preserve con1 cn
  by fastforce
next
case (app1 env es rs vs env' e bv)
hence list-all is-cupcake-exp (rev es)
  by fastforce
hence list-all is-cupcake-value (rev vs)


using app1 cupcake-list-preserve0 by force
hence is-cupcake-exp e and is-cupcake-all-env env'
using app1 cupcake-opapp-preserve by blast+
then show ?case
using app1 by blast
next
case (mat1 env e v0 pes e' uu env' be)
hence cupcake-c-ns (c env) cupcake-clauses pes is-cupcake-value v0
by (auto dest: is-cupcake-all-envD)
hence list-all (is-cupcake-value o snd) env' and e': is-cupcake-exp e'
using cupcake-match-result-preserve[where envC = c env and v = v0 and
pes = pes and err-v = Bindv, unfolded mat1, simplified]by auto
have is-cupcake-all-env (update-v (λ-. nsAppend (alist-to-ns env') (sem-env.v env)) env)
apply (rule cupcake-v-update-preserve)
apply fact
apply (rule cupcake-nsAppend-preserve)
apply (rule cupcake-alist-to-ns-preserve)
apply fact
apply (rule is-cupcake-all-envD)
apply fact
done
then show ?case
using mat1 e' by blast
qed (auto intro: cupcake-nsLookup-preserve dest: is-cupcake-all-envD)

lemma cupcake-single-preserve:
cupcake-evaluate-single env e (Rval res) ⇒ is-cupcake-all-env env ⇒ is-cupcake-exp
e ⇒ is-cupcake-value res
by (fastforce dest: cupcake-single-preserve0)

lemma cupcake-list-preserve:
list-all2-shortcircuit (cupcake-evaluate-single env) es rs ⇒
is-cupcake-all-env env ⇒ list-all is-cupcake-exp es ⇒ sequence-result rs = Rval
vs ⇒ list-all is-cupcake-value vs
by (induction es rs arbitrary:vs rule:list-all2-shortcircuit-induct) (fastforce dest: cupcake-single-preserve) +

private lemma cupcake-list-correct-rval:
assumes list-all2-shortcircuit
(λe r.
cupcake-evaluate-single env e r ∧
(is-cupcake-all-env env → is-cupcake-exp e → ∀(s::a state). ∃s'. evaluate
env s e (s', r)))
es rs is-cupcake-all-env env list-all is-cupcake-exp es sequence-result rs = Rval
vs
shows ∃s'. evaluate-list (evaluate env) (s::a state) es (s', Rval vs)
using assms proof (induction es rs arbitrary: s vs rule:list-all2-shortcircuit-induct)
case (cons-val e es y ys)
have $e$: is-cupcake-exp $e$ list-all is-cupcake-exp $es$
using cons-val by fastforce+
then obtain $vs'$ where $ys$: sequence-result $ys = Rval vs'$
using cons-val by fastforce
hence $vs$: $Rval vs = Rval (y \# vs')$
using cons-val by fastforce

from $e$ obtain $s'$ where evaluate env $s$ $e$ ($s'$, $Rval y$)
using cons-val by fastforce
from $e$ $ys$ obtain $s''$ where evaluate-list (evaluate env) $s'$ $es$ ($s''$, $Rval vs$)
using cons-val by fastforce

show ?case
  unfolding vs
  by (rule; rule evaluate-list.val1) fact+
qed (auto intro:evaluate-list.intros)

private lemma cupcake-list-correct-rerr:
  assumes list-all2-shortcircuit
  ($\lambda e r$.
  cupcake-evaluate-single env $e$ $r$ \land
  (is-cupcake-all-env env \rightarrow is-cupcake-exp $e$ \rightarrow ($\forall (s::'a state). \exists s'. evaluate
  env s e (s', r))))
  $es$ $rs$ is-cupcake-all-env env list-all is-cupcake-exp $es$ sequence-result $rs = Rerr$
err
  shows $\exists s'. evaluate-list (evaluate env) (s::'a state) es (s', Rerr err)$
using assms proof (induction $es$ $rs$ arbitrary: $s$ $err$ rule:list-all2-shortcircuit-induct)
  case (cons-val $e$ $es$ $y$ $ys$)
then have is-cupcake-exp $e$ list-all is-cupcake-exp $es$
  by fastforce+
moreover have $err$: sequence-result $ys = Rerr err$
using cons-val
  by (cases sequence-result $ys$) (auto simp: error-result.map-id)
ultimately show ?case
using cons3 cons-val
  by fast
qed (auto intro:evaluate-list.intros)

private lemma cupcake-list-correct0:
  assumes list-all2-shortcircuit
  ($\lambda e r$.
  cupcake-evaluate-single env $e$ $r$ \land
  (is-cupcake-all-env env \rightarrow is-cupcake-exp $e$ \rightarrow ($\forall (s::'a state). \exists s'. evaluate
  env s e (s', r))))
  $es$ $rs$ is-cupcake-all-env env list-all is-cupcake-exp $es$
shows $\exists s'. evaluate-list (evaluate env) (s::'a state) es (s', sequence-result $rs$)$
using assms by (cases sequence-result $rs$) (fastforce intro: cupcake-list-correct-rval
cupcake-list-correct-rerr)+
lemma cupcake-single-correct:
  assumes cupcake-evaluate-single env e res is-cupcake-all-env env is-cupcake-exp e
  shows \exists s'. Big-Step-Unlocked-Single.evaluate env s e (s', res)
using assms proof (induction arbitrary:s rule:cupcake-evaluate-single.induct)
case (con1 env cn es rs vs v0)
  then have list-all is-cupcake-exp (rev es)
    by (cases rule: is-cupcake-exp.cases[where \( x = \text{Con} \ cn \ es \)]) auto
then show ?case
  using cupcake-list-correct-rval evaluate.con1 con1
  by blast
next
case (con3 env cn es rs err)
  then have list-all is-cupcake-exp (rev es)
    by (cases rule: is-cupcake-exp.cases[where \( x = \text{Con} \ cn \ es \)]) auto
then show ?case
  using cupcake-list-correct-rerr con3 evaluate.con3
  by blast
next
case (app1 env es rs vs env' e bv)
  hence es: list-all is-cupcake-exp (rev es)
    by fastforce
  hence list-all is-cupcake-value (rev vs)
    using app1 cupcake-list-preserve list-all2-shortcircuit-weaken
    by (metis (no-types, lifting) list-all-rev)
  hence is-cupcake-exp e and is-cupcake-all-env env'
    using app1 cupcake-opapp-preserve by blast
then show ?case
  using cupcake-list-correct-rval es app1 evaluate.app1
  by blast
next
case (app3 env es rs vs)
  hence list-all is-cupcake-exp (rev es)
    by simp
then show ?case
  using cupcake-list-correct-rval evaluate.app3 app3
  by blast
next
case (app6 env es rs err op0)
  hence list-all is-cupcake-exp (rev es)
    by simp
then show ?case
  using cupcake-list-correct-rerr app6 evaluate.app6
  by blast
next
case (mat1 env e v0 pes e' uu env' be)
  hence e: is-cupcake-exp e and cupcake-c-ns (c env) and pes: cupcake-clauses
pes and is-cupcake-value v0
   by (fastforce dest: is-cupcake-all-envD cupcake-single-preserve)+
   hence list-all (is-cupcake-value o snd) env' and e': is-cupcake-exp e'
   using cupcake-match-result-preserve[where envC = c env and v = v0 and
   pes = pes and err-v = Bindv, unfolded mat1, simplified]
   by blast+
   have env': is-cupcake-all-env (update-v (λ- nsAppend (alist-to-ns env') (sem-env.v env)) env)
      apply (rule cupcake-v-update-preserve)
      apply fact
      apply (rule cupcake-nsAppend-preserve)
      apply fact
      apply (rule is-cupcake-all-envD)
      apply fact
   done

from e obtain s' where evaluate env s e (s', Rval v0)
   using mat1 by blast
   have match-result env s' v0 pes Bindv = Rval (e', env')
      using mat1 cupcake-match-result-eq[OF pes, where env = env and v = v0 and
      err-v = Bindv and s = s']
      by fastforce
      from e' env' obtain s'' where evaluate (update-v (λ- nsAppend (alist-to-ns env') (sem-env.v env)) env) s' e' (s'', bv)
      using mat1 by blast

show ?case
   by rule+ fact+
next
case (mat1error env e v0 pes err)
   hence is-cupcake-exp e and pes: cupcake-clauses pes
   by (auto dest: is-cupcake-all-envD)
then obtain s' where Big-Step-Unclocked-Single.evaluate env s e (s', Rval v0)
   using mat1error by blast
   hence match-result env s' v0 pes Bindv = Rerr err
      using cupcake-match-result-eq[OF pes, where env = env and s = s' and v = v0 and err-v = Bindv] unfolding mat1error
      by (simp add: error-result.map-id)

show ?case
   by (rule; rule evaluate.mat1b) fact+
next
case (mat2 - e)
   hence is-cupcake-exp e
   by simp
then show ?case
   using mat2 evaluate.mat2 by blast
QED (blast intro:evaluate.intros)+

Lemma cupcake-list-correct:
Assumes list-all2-shortcircuit (cupcake-evaluate-single env) es rs is-cupcake-all-env env list-all is-cupcake-exp es
Shows \( \exists s'. \) evaluate-list (evaluate env) (s::'a state) es (s',sequence-result rs)
Using assms by (fastforce intro:cupcake-list-correct0 list-all2-shortcircuit-weaken cupcake-single-correct)+

Private lemma cupcake-list-complete0:
evaluate-list
\((\lambda s e r. \text{evaluate env } s e r \land (\text{is-cupcake-all-env env } \implies \text{is-cupcake-exp } e \implies \cupcake-evaluate-single env e (\text{snd } r)))\) s1 es res \implies
is-cupcake-all-env env \implies list-all is-cupcake-exp es \implies \exists rs. list-all2-shortcircuit (cupcake-evaluate-single env) es rs \land sequence-result rs = (\text{snd } res)
Proof (induction rule:evaluate-list.induct)
Case empty
Have list-all2-shortcircuit (cupcake-evaluate-single env) [] []
By fastforce
Then show ?case
By fastforce
Next
Case (cons1 s1 e s2 v es s3 vs)
Then obtain rs where list-all2-shortcircuit (cupcake-evaluate-single env) es rs
And sequence-result rs = Rval vs
And list-all2-shortcircuit (cupcake-evaluate-single env) (e # es) (Rval v # rs)
By fastforce+
Then show ?case
By fastforce
Next
case (cons2 s1 e s2 err es)
hence list-all2-shortcircuit (cupcake-evaluate-single env) (e # es) [Rerr err]
By simp
Then show ?case
By fastforce
Next
case (cons3 s1 e s2 v es s3 err)
Then obtain rs where list-all2-shortcircuit (cupcake-evaluate-single env) es rs
And err:sequence-result rs = Rerr err
And list-all2-shortcircuit (cupcake-evaluate-single env) (e # es) (Rval v # rs)
By fastforce
Moreover have sequence-result (Rval v # rs) = Rerr err
By (auto simp: error-result.map-id err)
Ultimately show ?case
By fastforce
QED

Private lemma cupcake-single-complete0:
evaluate env s e res \implies \text{is-cupcake-all-env env } \implies \text{is-cupcake-exp } e \implies \cupcake-evaluate-single
proof (induction rule: evaluate.induct)
case (con1 env cn es vs v s1 s2)
  hence list-all is-cupcake-exp (rev es)
    by (cases rule: is-cupcake-exp.cases [where x = Con cn es]) auto
  hence list-all2-shortcircuit (cupcake-evaluate-single env) (rev es) (map Rval vs)
    using cupcake-list-complete0 con1 by fastforce
show ?case
  by (simp|rule|fact)+
next
case (con3 env cn es s1 s2 err)
  hence list-all is-cupcake-exp (rev es)
    by (cases rule: is-cupcake-exp.cases [where x = Con cn es]) auto
then obtain rs where list-all2-shortcircuit (cupcake-evaluate-single env) (rev es)
    rs sequence-result rs = Rerr err
  using con3 by (fastforce dest: cupcake-list-complete0)
show ?case
  by (simp; rule cupcake-evaluate-single con3) fact+
next
case (app1 env s1 es s2 vs envʼ e bv)
then obtain rs where rs: list-all2-shortcircuit (cupcake-evaluate-single env) (rev es)
    rs sequence-result rs = Rval vs
  by (fastforce dest: cupcake-list-complete0)
  hence list-all is-cupcake-exp (rev es)
    using app1 by fastforce
  hence list-all is-cupcake-value vs list-all is-cupcake-value (rev vs)
    using cupcake-list-preserve app1 rs by fastforce+
  hence is-cupcake-exp e is-cupcake-all-env envʼ
    using app1 cupcake-opapp-preserve by fastforce+
  hence cupcake-evaluate-single envʼ e (snd bv)
    using app1 by fastforce
show ?case
  by rule fact+
next
case (app3 env s1 es s2 vs)
  hence list-all is-cupcake-exp (rev es)
    by simp
  obtain rs where list-all2-shortcircuit (cupcake-evaluate-single env) (rev es)
    rs sequence-result rs = Rval vs
    using app3 cupcake-list-complete0 by fastforce
show ?case
  by (simp|rule|fact)+
next
case (app6 env s1 es s2 err op0)
  obtain rs where list-all2-shortcircuit (cupcake-evaluate-single env) (rev es)
    rs sequence-result rs = Rerr err
    using cupcake-list-complete0 app6 by fastforce
show ?case
  by (simp|rule|fact)+
next
case (mat1 env s1 e s2 v1 pes e' env' bv)
hence is-cupcake-exp e and cupcake-c-ns (c env) and pes:cupcake-clauses pes
and is-cupcake-value v1
  by (fastforce dest: is-cupcake-all-envD cupcake-single-preserve+)

moreover obtain uu where cupcake-match-result (c env) v1 pes Bindu = Real (e', uu, env')
  using cupcake-match-result-eq[OF pes,where env = env and s = s2 and v = v1 and err-v = Bindu, unfolded mat1]
  by (cases cupcake-match-result (c env) v1 pes Bindu) auto

ultimately have list-all (is-cupcake-value o snd) env' is-cupcake-exp e'
  using cupcake-match-result-preserve[where env' = c env and v = v1 and pes = pes and err-v = Bindu]
  by fastforce+
moreover have is-cupcake-all-env (update-v (λ-. nsAppend (alist-to-ns env')) (sem-env.v env)) env
  apply (rule cupcake-v-update-preserve)
  apply fact
  apply (rule cupcake-nsAppend-preserve)
  apply fact
  apply (rule is-cupcake-all-envD)
  apply fact
  done

ultimately have cupcake-evaluate-single env e (Real v1)
and cupcake-evaluate-single (update-v (λ-. nsAppend (alist-to-ns env')) (sem-env.v env)) e' (snd bv)
  using mat1b by fastforce+

show ?case
  by (rule cupcake-evaluate-single.mat1) fact+
next
case (mat1b env s1 e s2 v1 pes err)
hence is-cupcake-exp e and pes: cupcake-clauses pes
  by (auto dest: is-cupcake-all-envD)

have cupcake-evaluate-single env e (Real v1)
  using mat1b by fastforce
have cupcake-match-result (c env) v1 pes Bindv = Rerr err
  using cupcake-match-result-eq[OF pes,where env = env and s = s2 and v = v1 and err-v = Bindv, unfolded mat1b]
  by (cases (cupcake-match-result (c env) v1 pes Bindv)) (auto simp:error-result.map-id)
show ?case
  by (simp; rule cupcake-evaluate-single.mat1error) fact+
qed (fastforce intro: cupcake-evaluate-single.intros)
lemma cupcake-single-complete:
  evaluate env s e (s', res) ⇒ is-cupcake-all-env env ⇒ is-cupcake-exp e ⇒
  cupcake-evaluate-single env e res
  by (fastforce dest:cupcake-single-complete0)

lemma cupcake-list-complete:
  evaluate-list (evaluate env) s1 es res ⇒ is-cupcake-all-env env ⇒
  list-all is-cupcake-exp es ⇒ ∃ rs. list-all2-shortcircuit
  (cupcake-evaluate-single env) es rs ∧ sequence-result rs = (snd res)
  by (fastforce intro:cupcake-list-complete0 cupcake-single-complete evaluate-list-mono-strong)

private lemma cupcake-list-state-preserve0:
  assumes evaluate-list (λs e res. Big-Step-Unclocked-Single.evaluate env s e res
  ∧ (is-cupcake-all-env env ⇒ is-cupcake-exp e ⇒ s = fst res)) s es res
  shows s = (fst res)
  using assms by (induction rule:evaluate-list.induct) auto

lemma cupcake-state-preserve:
  assumes Big-Step-Unclocked-Single.evaluate env s e res is-cupcake-all-env env
  is-cupcake-exp e
  shows s = (fst res)
  using assms proof (induction arbitrary: rule: evaluate.induct)
  case (con1 env cn es vs v s1 s2)
  hence list-all is-cupcake-exp es
    by (cases rule: is-cupcake-exp.cases[where x = Con cn es]) auto
  then show ?case
    using con1 by (fastforce dest:cupcake-list-state-preserve0)

next
  case (con3 env cn es s1 s2 err)
  hence list-all is-cupcake-exp es
    by (cases rule: is-cupcake-exp.cases[where x = Con cn es]) auto
  then show ?case
    using con3 by (fastforce dest:cupcake-list-state-preserve0)

next
  case (app1 env s1 es s2 vs env' e bv)
  have list-all is-cupcake-exp (rev es)
    using app1 by fastforce
  then obtain rs where: list-all2-shortcircuit (cupcake-evaluate-single env) (rev es) rs sequence-result rs = Rval vs
    using app1 by (fastforce dest:evaluate-list-mono-strong[THEN cupcake-list-complete])
  hence list-all is-cupcake-value vs list-all is-cupcake-value (rev vs)
    using cupcake-list-preserve app1 rs by fastforce+
  hence is-cupcake-exp e is-cupcake-all-env env'
    using app1 cupcake-opapp-preserve by fastforce+
  then show ?case
    using app1 by (fastforce dest:cupcake-list-state-preserve0)

next
  case (mat1 env s1 e s2 v1 pes e' env' bv)
hence \( \text{is-cupcake-exp} \ e \) and \( \text{cupcake-c-ns} \ (c \ \text{env}) \) and \( \text{pes} : \text{cupcake-clauses} \ \text{pes} \)
and \( \text{is-cupcake-value} \ v1 \)
by (fastforce dest: is-cupcake-all-envD cupcake-single-complete cupcake-single-preserve) +

moreover obtain \( uu \) where \( \text{cupcake-match-result} \ (c \ \text{env}) \ v1 \ \text{pes} \text{Bindv} = \text{Real} \ (e', \ uu, \ \text{env}') \)
using \( \text{cupcake-match-result-eq} \ ) [OF \ \text{pes}, \text{where} \ \text{env} = \text{env} \ and \ s = s2 \ and \ v = v1 \ and \ err-v = \text{Bindv}, \text{unfolded mat1}] \)
by (cases cupcake-match-result (c env) v1 pes Bindv) auto

ultimately have list-all \( (\text{is-cupcake-value} \circ \text{snd}) \ \text{env}' \) \( \text{is-cupcake-exp} \ e' \)
using \( \text{cupcake-match-result-preserve} \text{where} \ \text{env}C = c \ \text{env} \ and \ v = v1 \ and \ \text{pes} = \text{pes} \ and \ err-v = \text{Bindv} \)
by fastforce +
moreover have \( \text{is-cupcake-all-env} \ (\text{update-v} \ (\lambda-. \ \text{nsAppend} \ (\text{alist-to-ns env}') \)) \ (\text{sem-env} \text{v env})) \ \text{env} \)
apply (rule cupcake-v-update-preserve)
apply fact
apply (rule cupcake-nsAppend-preserve)
apply (rule cupcake-alist-to-ns-preserve)
apply fact
apply (rule is-cupcake-all-envD)
apply fact
done
ultimately show \( \text{?case} \)
using mat1 by fastforce
qed (fastforce dest: cupcake-list-state-preserve0) +

corollary cupcake-single-correct-strong:
assumes cupcake-evaluate-single env e res is-cupcake-all-env env is-cupcake-exp e
shows Big-Step-Unclocked-Single.evaluate env s e (s,res)
using assms cupcake-single-correct cupcake-state-preserve by fastforce

corollary cupcake-single-complete-weak:
evaluate env s e (s, res) \( \rightarrow \) is-cupcake-all-env env \( \rightarrow \) is-cupcake-exp e \( \rightarrow \)
cupcake-evaluate-single env e res
using cupcake-single-complete by fastforce

end end

hide-const (open) \( c \)

end
Chapter 3

Term rewriting

theory Doc-Rewriting
imports Main
begin

end

theory General-Rewriting
imports Terms-Extras
begin

locale rewriting =
  fixes R :: 'a::term ⇒ 'a ⇒ bool
  assumes R-fun: R t t' ⇒ R (app t u) (app t' u)
  assumes R-arg: R u u' ⇒ R (app t u) (app t u')
begin

lemma rt-fun:
  R** t t' ⇒ R** (app t u) (app t' u)
by (induct rule: rtranclp.induct) (auto intro: rtranclp.rtrancl-into-rtrancl R-fun)

lemma rt-arg:
  R** u u' ⇒ R** (app t u) (app t u')
by (induct rule: rtranclp.induct) (auto intro: rtranclp.rtrancl-into-rtrancl R-arg)

lemma rt-comb:
  R** t1 u1 ⇒ R** t2 u2 ⇒ R** (app t1 t2) (app u1 u2)
by (metis rt-fun rt-arg rtranclp-trans)

lemma rt-list-comb:
  assumes list-all2 R** ts us R** t u
  shows R** (list-comb t ts) (list-comb u us)
using assms
by (induction ts us arbitrary: t u rule: list.rel-induct) (auto intro: rt-comb)

end

77
3.1 Higher-order term rewriting using de-Bruijn indices

theory Rewriting-Term
imports
  ../Terms/General-Rewriting
  ../Terms/Strong-Term
begin

3.1.1 Matching and rewriting

type-synonym rule = term × term

inductive rewrite :: rule fset ⇒ term ⇒ term ⇒ bool (-/ ⊢ / - −→/ - [50,0,50] 50) for rs where
  step: r ∈ rs ⇒ r ⊢ t → u ⇒ rs ⊢ t −→ u
  beta: rs ⊢ t −→ t'⇒ rs ⊢ t $ u −→ t' $ u
  fun: rs ⊢ u −→ u'⇒ rs ⊢ t $ u −→ t $ u'

global-interpretation rewrite: rewriting rewrite rs for rs by standard (auto intro: rewrite.intro simp: app-term-def)+

abbreviation rewrite-rt :: rule fset ⇒ term ⇒ term ⇒ bool (-/ ⊢ / - −→∗/ - [50,0,50] 50) where
  rewrite-rt rs ≡ (rewrite rs)**

lemma rewrite-beta-alt: t [t \beta] = u ⇒ wellformed t' ⇒ rs ⊢ (Λ t $ t') −→ u by (metis rewrite.beta)

3.1.2 Wellformedness

primrec rule :: rule ⇒ bool where
  rule (lhs, rhs) ←→ basic-rule (lhs, rhs) ∧ Term.wellformed rhs

lemma ruleI[intro]:
  assumes basic-rule (lhs, rhs)
  assumes Term.wellformed rhs
  shows rule (lhs, rhs)
  using assms by simp

lemma split-rule-fst: fst (split-rule r) = head (fst r)
unfolding head-def by (smt prod.case-eq-if prod.collapse prod.inject split-rule.simps)

locale rules = constants C-info heads-of rs for C-info and rs :: rule fset +
  assumes all-rules: fBall rs rule
assumes \textit{arity}: \textit{arity-compatibles rs}
assumes \textit{fmap}: \textit{is-fmap rs}
assumes \textit{patterns}: \textit{pattern-compatibles rs}
assumes \textit{nonempty}: rs \neq \{\}
assumes \textit{not-shadows}: \textit{fBall rs} (\lambda(lhs, -). \neg \textit{shadows-consts} lhs)
assumes \textit{welldefined-rs}: \textit{fBall rs} (\lambda(-, rhs). \textit{welldefined} rhs)

begin

\textbf{lemma} \textit{rewrite-wellformed}:
assumes rs ⊢ t → t’ wellformed t
shows wellformed t’
using \textit{assms}

\textbf{proof} (\textit{induction rule}: rewrite.induct)
case (step r t u)
obtain lhs rhs where \( r = (lhs, rhs) \)
by force
hence wellformed rhs
using all-rules step by force
show ?case
apply (rule \textit{wellformed}.\textit{rewrite-step})
apply (rule step(2)[unfolded \( r = \)-])
apply \textit{fact}+
done

next
case (beta u t)
show ?case
unfolding \textit{wellformed-term-def}
apply (rule \textit{replace-bound-wellformed})
using \textit{beta} by \textit{auto}
qed \textit{auto}

\textbf{lemma} \textit{rewrite-rt-wellformed}:
rs ⊢ t →∗ t’ =⇒ wellformed t =⇒ wellformed t’
by (\textit{induction rule}: rtranclp.induct) (\textit{auto intro: rewrite-wellformed simp del: wellformed-term-def})

\textbf{lemma} \textit{rewrite-closed}:
rs ⊢ t → t’ =⇒ closed t =⇒ closed t’

\textbf{proof} (\textit{induction t t’ rule: rewrite.induct})
case (step r t u)
obtain lhs rhs where \( r = (lhs, rhs) \)
by force
with step have rule (lhs, rhs)
using all-rules by blast
hence frees rhs \( \subseteq \) frees lhs
by \textit{simp}
moreover have (lhs, rhs) ⊢ t → u
using step unfolding \( \langle r = \rangle \) by \textit{simp}

show ?case
apply (rule \textit{rewrite-step-closed})
apply \textit{fact}+
done

79
next
case (beta t t')
  have frees t [t'] β ⊆| frees t |∪| frees t' by (rule replace-bound-frees)
  with beta show ?case
    unfolding closed-except-def by auto
qed (auto simp: closed-except-def)

lemma rewrite-rt-closed: rs ⊢ t →* t' ⇒ closed t ⇒ closed t'
by (induction rule: rtranclp.induct) (auto intro: rewrite-closed)
end

end

3.2 Higher-order term rewriting using explicit bound variable names

theory Rewriting-Nterm
imports
  Rewriting-Term
  Higher-Order-Terms.Term-to-Nterm
  ../Terms/Strong-Term
begin

3.2.1 Definitions

type-synonym nrule = term × nterm

abbreviation nrule :: nrule ⇒ bool where
nrule ≡ basic-rule

fun (in constants) not-shadowing :: nrule ⇒ bool where
not-shadowing (lhs, rhs) ←→ ¬ shadows-consts lhs ∧ ¬ shadows-consts rhs

locale nrules = constants C-info heads-of rs for C-info and rs :: nrule fset +
  assumes all-rules: fBall rs nrule
  assumes arity: arity-compatibles rs
  assumes fmap: is-fmap rs
  assumes patterns: pattern-compatibles rs
  assumes nonempty: rs ≠ {||}
  assumes not-shadows: fBall rs not-shadowing
  assumes welldefined-rs: fBall rs (λ(_, rhs). welldefined rhs)

3.2.2 Matching and rewriting

inductive nrewrite :: nrule fset ⇒ nterm ⇒ nterm ⇒ bool (-/ ⊢/ - →*/ -
  [50,0,50] 50) for rs where
step: r ∈| rs ⇒ r ⊢ t → u ⇒ rs ⊢ t → u |
beta: \( \Gamma \vdash \lambda x.t \, \$n \, t' \rightarrow \text{subst} \ (\text{fmap-of-list} \ [(x, t')]) \ |
\)

fun: \( \Gamma \vdash t \rightarrow t' \rightarrow \Gamma \vdash t \, \$n \, u \rightarrow t' \, \$n \, u \ |
\)

arg: \( \Gamma \vdash \lambda n.t \rightarrow \lambda n.u \rightarrow \lambda n.t \, \$n \, u \rightarrow t \, \$n \, u' \ |
\)

global-interpretation \( \text{rewrite} \): rewriting \( \text{rewrite} \ \Gamma \) for \( \Gamma \)
by standard \( \text{(auto intro: \text{rewrite.intro simp: app-nterm-def})} \)

abbreviation \( \text{rewrite-rt} :: \text{nrule fset} \Rightarrow \text{nterm} \Rightarrow \text{nterm} \Rightarrow \text{bool} \ (-/- \vdash \_/- \rightarrow \_/-) - [50,0,50] 50 \) where
\( \text{rewrite-rt} \ \Gamma \equiv (\text{rewrite} \ \Gamma) \star\star \)

lemma (in \( \text{nrules} \)) \( \text{rewrite-closed} \): \( \vdash \text{assms} \)
shows \( \text{closed} t' \)
using \( \text{assms} \) proof induction
case \( \text{(step r t u)} \)
obtain \( \text{lhs rhs where } r = (\text{lhs}, \text{rhs}) \)
by force
with \( \text{step} \) have \( \text{nrule} \ (\text{lhs}, \text{rhs}) \)
using all-rules by blast
hence \( \text{frees rhs} \subseteq \text{frees lhs} \)
by simp
have \( (\text{lhs}, \text{rhs}) \vdash t \rightarrow u \)
using \( \text{step unfolding} \ (r = -) \) by simp

show \( ?\text{case} \)
apply \( \text{(rule rewrite-step-closed)} \)
apply fact+
done
next
case beta
show \( ?\text{case} \)
apply simp
apply \( \text{(subst closed-except-def)} \)
apply \( \text{(subst subst-frees)} \)
using \( \text{beta unfolding} \) closed-except-def by auto
qed \( \text{(auto simp: closed-except-def)} \)

corollary (in \( \text{nrules} \)) \( \text{rewrite-rt-closed} \):
assumes \( \vdash \text{assms} \)
shows \( \text{closed} t' \)
using \( \text{assms} \)
by induction \( \text{(auto intro: rewrite-closed)} \)

3.2.3 Translation from \( \text{Term-Class.term to nterm} \)
context begin

private lemma term-to-nterm-all-vars0:
assumes wellformed' (length Γ) t
shows ∃ T. all-frees (fst (run-state (term-to-nterm Γ t) x)) |⊆| fset-of-list Γ |∪| frees t |∪| T ∧ fBall T (λy. y ≠ x)
using assms proof (induction Γ t arbitrary: x rule: term-to-nterm-induct)
  case (bound Γ t)
    hence Γ ! i |∈| fset-of-list Γ
      by (simp add: fset-of-list-elem)
    with bound show ?case
      by (auto simp: State-Monad.return-def)
next
  case (abs Γ t)
    obtain T
      where all-frees (fst (run-state (term-to-nterm (next x # Γ) t) (next x))) |⊆|
        fset-of-list (next x ≠ Γ) |∪| frees t |∪| T
      and fBall T ((<) (next x))
      apply atomize-elim
      apply (rule abs (1))
      using abs by auto
    show ?case
      apply (simp add: split-beta create-alt-def)
      apply (rule exI[where x = finsert (next x) T])
      subgoal by auto
    subgoal using (all-frees (fst (run-state - (next x))) |⊆| -)
      by simp
    subgoal
      apply simp
      apply (rule conjI)
      apply (rule next-ge)
      using ⟨fBall T ((<) (next x))⟩
      by (metis fBallE fBallI fresh-class.next-ge.order.strict-trans)
    done
next
  case (app Γ t1 t2 x1)
    obtain t1' x2 where run-state (term-to-nterm Γ t1) x1 = (t1', x2)
      by fastforce
    moreover obtain T1
      where all-frees (fst (run-state (term-to-nterm Γ t1) x1)) |⊆| fset-of-list Γ |∪|
        frees t1 |∪| T1
      and fBall T1 ((<) x1)
      apply atomize-elim
      apply (rule app (1))
      using app by auto
    ultimately have all-frees t1' |⊆| fset-of-list Γ |∪| frees t1 |∪| T1
      by simp
    obtain T2
      where all-frees (fst (run-state (term-to-nterm Γ t2) x2)) |⊆| fset-of-list Γ |∪|
frees $t_2 \mid \cup \mid T_2$
and $f\text{Ball } T_2 ((<) \, x_2)$
apply atomize-clim
apply (rule app(2))
using app by auto
moreover obtain $t_2' \, x'$ where run-state (term-to-nterm $\Gamma \, t_2) \, x_2 = (t_2', \, x')$
by fastforce
ultimately have all-frees $t_2' \mid \subseteq \mid f\text{set-of-list} \, \Gamma \mid \cup \mid \text{frees } t_2 \mid \cup \mid T_2$
by simp
have $x_1 \leq x_2$
apply (rule state-io-relD[OF term-to-nterm-mono])
apply fact
done

show ?case
apply simp
unfolding :run-state (term-to-nterm $\Gamma \, t_1) \, x_1 = -$ by simp
apply simp
unfolding :run-state (term-to-nterm $\Gamma \, t_2) \, x_2 = -$ by simp
apply (rule exI[where $x = t_1 \mid \cup \mid T_2]$)
apply (intro conjI)
subgoal using (all-frees $t_1' \mid \subseteq \mid -$) by auto
subgoal using (all-frees $t_2' \mid \subseteq \mid -$) by auto
subgoal
apply auto
using (fBall $T_1 ((<) \, x_1)$: apply auto[])
using (fBall $T_2 ((<) \, x_2)$: ($x_1 \leq x_2$)
by (meson fBallE less-le-not-le order-trans)
done
qed auto

lemma term-to-nterm-all-vars:
assumes wellformed $t \, \text{fdisjnt} \, \text{frees}$ $t \, S$
shows $\text{fdisjnt} \, \text{(all-frees (fresh-frun (term-to-nterm [] $t) \, (T \mid \cup \mid S)))} \, S$
proof
let $?\Gamma = []$
let $?x = \text{fresh-fNext} \, (T \mid \cup \mid S)$
from assms have wellformed' (length $?\Gamma) \, t$
by simp
then obtain $F$ where all-frees (fst (run-state (term-to-nterm $?\Gamma \, t) \, $?x)) \mid \subseteq \mid f\text{set-of-list} \, ?\Gamma \mid \cup \mid \text{frees} \, t \mid \cup \mid F$
and fBall $F \, (\lambda y. \, y > $?x)$
by (metis term-to-nterm-all-vars0)

have fdisjnt $F \, (T \mid \cup \mid S)$ if $S \neq \{||\}$
apply (rule fdisjnt-ge-max)

83
apply (rule fBall-pred-weaken[OF - (fBall F (λy. y > ?x))])
apply (rule less-trans)
apply (rule fNext-ge-max)
using that by auto
show ?thesis
apply (rule fdisjnt-subset-left)
apply (subst fresh-frun-def)
apply (subst fresh-fNext-def[symmetric])
apply fact
apply simp
apply (rule fdisjnt-union-left)
apply fact
using (-⇒ fdisjnt F (T |∪| S)) by (auto simp: fdisjnt-alt-def)
qed

end

fun translate-rule :: name fset ⇒ rule ⇒ nrule where
translate-rule S (lhs, rhs) = (lhs, fresh-frun (term-to-nterm [] rhs) (frees lhs |∪| S))

lemma translate-rule-alt-def: translate-rule S = (λ(lhs, rhs). (lhs, fresh-frun (term-to-nterm [] rhs) (frees lhs |∪| S)))
by auto

definition compile’ where
compile’ C-info rs = translate-rule (pre-constants.all-consts C-info (heads-of rs)) |·| rs

context rules begin

definition compile :: nrule fset where
compile = translate-rule all-consts |·| rs

lemma compile’-compile-eq[simp]: compile’ C-info rs = compile
unfolding compile’-def compile-def ..

lemma compile-heads: heads-of compile = heads-of rs
unfolding compile-def translate-rule-alt-def head-def[abs-def]
by force

lemma compile-rules: nrules C-info compile
proof
have fBall compile (λ(lhs, rhs). nrule (lhs, rhs))
proof safe
fix lhs rhs
assume (lhs, rhs) |∈| compile
then obtain rhs’
where \((lhs, rhs') \in rs\)
and \(rhs\) = \text{fresh-frun} (\text{term-to-nterm} [] rhs') (\text{frees} lhs \cup \text{all-consts})
unfolding compile-def by force
then have rule: rule \((lhs, rhs')\)
using all-rules by blast

show nrule \((lhs, rhs)\)
proof
  from rule show linear \(lhs\) is-const \((\text{fst (strip-comb lhs)}) \sim\) is-const \(lhs\) by auto

  have frees rhs \(\subseteq\) frees rhs'
  unfolding rhs using rule
  by (metis rule.simps term-to-nterm-vars)
  also have frees rhs' \(\subseteq\) frees lhs
  using rule by auto
  finally show frees rhs \(\subseteq\) frees lhs .
  qed
qed

thus fBall compile nrule
  by fast

next
  show arity-compatibles compile
  proof safe
    fix \(lhs_1\) \(rhs_1\) \(lhs_2\) \(rhs_2\)
    assume \((lhs_1, rhs_1) \in compile (lhs_2, rhs_2) \in compile\)
    then obtain \(rhs_1'\) \(rhs_2'\) where \((lhs_1, rhs_1') \in rs (lhs_2, rhs_2') \in rs\)
    unfolding compile-def by force
    thus arity-compatible \(lhs_1\) \(lhs_2\)
    using arity by (blast dest: fpairwiseD)
    qed

next
  have is-fmap rs
  using fmap by simp
  thus is-fmap compile
  unfolding compile-def translate-rule-alt-def
  by (rule is-fmap-image)

next
  have pattern-compatibles rs
  using patterns by simp
  thus pattern-compatibles compile
  unfolding compile-def translate-rule-alt-def
  by (auto dest: fpairwiseD)

next
  show fdisjnt \((\text{heads-of compile})\) \(C\)
  using disjnt by (simp add: compile-heads)

next
  have fBall compile not-shadowing
proof safe
fix lhs rhs
assume (lhs, rhs) ∈ compile
then obtain rhs'
where rhs = fresh-frun (term-to-nterm [] rhs') (frees lhs |∪| all-consts)
and (lhs, rhs') ∈ rs
unfolding compile-def translate-rule-alt-def by auto
hence rule (lhs, rhs') ∼ shadows-consts lhs
using all-rules not-shadows by blast+
moreover hence wellformed rhs' frees rhs' |⊆| frees lhs fdisjnt all-consts
(frees lhs)
unfolding shadows-consts-def by simp+
moreover have ∼ shadows-consts rhs
apply (subst shadows-consts-def)
apply simp
unfolding (rhs = -)
apply (rule fdisjnt-swap)
apply (rule term-to-nterm-all-vars)
apply fact
apply (rule fdisjnt-subset-left)
apply fact
apply (rule fdisjnt-swap)
apply fact
done
ultimately show not-shadowing (lhs, rhs)
unfolding compile-heads by simp
qed
thus fBall compile (constants.not-shadowing C-info (heads-of compile))
unfolding compile-heads .

have fBall compile (λ(·, rhs). welldefined rhs)
unfolding compile-heads
unfolding compile-def ball-simps
apply (rule fBall-pred-weaken[OF welldefined-rs])
subgoal for x
apply (cases x)
apply simp
apply (subst fresh-frun-def)
apply (subst term-to-nterm-consts[THEN pred-state-run-state])
by auto
done
thus fBall compile (λ(·, rhs). consts rhs |⊆| pre-constants.all-consts C-info
(heads-of compile))
unfolding compile-heads .

next
show compile ≠ {}{}
using nonempty unfolding compile-def by auto
next
  show distinct all-constructors
  by (fact distinct-ctr)
qed

sublocale rules-as-nrules: nrules C-info compile
by (fact compile-rules)
end

3.2.4 Correctness of translation

theorem (in rules) compile-correct:
  assumes compile \( \vdash_n u \longrightarrow u' \) closed u
  shows rs \( \vdash nterm-to-term' \ u\longrightarrow nterm-to-term' \ u' \)
using assms proof (induction u u')
  case (step r u u')
  moreover obtain pat rhs' where r = (pat, rhs')
    by force
  ultimately obtain nenv where match pat u = Some nenv u' = subst rhs' nenv
    by auto
  then obtain env where nrelated.P-env [] env nenv match pat (nterm-to-term [] u) = Some env
    by (metis nrelated.match-rel option.exhaust option.rel-distinct(1) option.rel-inject(2))
  have closed-env nenv
    using step ⟨match pat u = Some nenv⟩ by (intro closed.match)
  from step obtain rhs
    where rhs' = fresh-frun (term-to-nterm [] rhs) (frees pat |∪| all-consts) (pat, rhs) \( |∈| \) rs
      unfolding ⟨r = \( \cdot \)⟩ compile-def by auto
  with assms have rule (pat, rhs)
    using all-rules by blast
  hence rhs = nterm-to-term [] rhs'
    unfolding ⟨rhs' = \( \cdot \)⟩
    by (simp add: term-to-nterm-nterm-to-term fresh-frun-def)
  have compile \( \vdash_n u \longrightarrow u' \)
    using step by (auto intro: nrewrite.step)
  hence closed u'
    by (rule rules-as-nrules.nrewrite-closed) fact
show ?case
proof (rule rewrite.step)
  show (pat, rhs) \( \vdash nterm-to-term [] u \longrightarrow nterm-to-term [] u' \)
    apply (subst nterm-to-term-eq-closed)
    apply fact
    apply simp
apply (rule exI[where \( x = \text{env} \)])
apply (rule conjI)
apply fact
unfolding (rhs = \( \))
apply (subst nrelated-subst)
apply fact
unfolding fdisjnt-alt-def apply simp
unfolding \( u' = \text{subst} \ rhs \ ' \ \text{nenv} \) by simp
qed fact
next
case beta
show ?case
apply simp
apply (subst subst-single-eq[symmetric, simplified])
apply (subst nterm-to-term-subst-replace-bound[where \( n = 0 \)])
subgoal using beta by (simp add: closed-except-def)
subgoal by simp
subgoal by simp
subgoal by simp (rule rewrite.beta)
done
qed (auto intro: rewrite.intros simp: closed-except-def)

3.2.5 Completeness of translation

context rules begin

context
notes [simp] = closed-except-def fdisjnt-alt-def
begin

private lemma compile-complete0:
assumes \( \Gamma \vdash t \rightarrow t' \) closed \( t \) wellformed \( t \)
obtains \( u' \) where compile \( \Gamma_n \) \( \text{fst} (\text{run-state (term-to-nterm } \Gamma \ [ ] \ t) \ s) \rightarrow u' \ u' \)
\( \approx_{\alpha} \text{fst (run-state (term-to-nterm } \Gamma \ [ ] t') \ s') \)
apply atomize-elim
using assms proof (induction \( t \ t' \) arbitrary: \( s \ s' \))
case (step \( r \ t \ t' \))
let \( ?\!\!t_n = \text{fst (run-state (term-to-nterm } \Gamma \ [ ] t) \ s) \)
let \( ?\!\!t_n' = \text{fst (run-state (term-to-nterm } \Gamma \ [ ] t') \ s') \)
from step have closed \( t \) closed \( ?\!\!t_n \)
using term-to-nterm-vars0[where \( \Gamma = [ ] \)]
by force+
from step have nterm-to-term' \( ?\!\!t_n = t \)
find-theorems nterm-to-term fdisjnt
by (auto intro!: term-to-nterm-nterm-to-term0)

obtain pat rhs' \( r = (\text{pat, rhs}') \)
by fastforce

88
with step obtain env' where match pat t = Some env' t' = subst rhs' env'
  by fastforce
with (= t) have rel-option (nrelated.P-env []) (match pat t) (match pat (?t_n))
  by (metis nrelated.match-rel)
with step (= Some env')
  obtain env where nrelated.P-env [] env' env match pat ?t_n = Some env
  by (metis (no-types, lifting) option-rel-Some1)
with (closed ?t_n) have closed-env env
  by (blast intro: closed.match)

from step obtain rhs
  where rhs = fresh-frun (term-to-nterm [] rhs') (frees pat |∪| allconsts) (pat, rhs) ∈ compile
  unfolding (= r = -) compile-def
  by force
with step have rule (pat, rhs')
  unfolding (= r = -)
  using all-rules by fast
hence nterm-to-term' rhs = rhs'
  unfolding (rhs = -)
  by (simp add: fresh-frun-def term-to-nterm-nterm-to-term)
obtain u' where subst rhs env = u'
  by simp
have t' = nterm-to-term' u'
  unfolding (= t' = -)
  unfolding (= r = rhs') [symmetric]
  apply (subst nrelated-subst)
  apply fact+      
  using (= u')
  by simp+

have compile ⊢_n ?t_n −→ u'
  apply (rule nrewrite.step)
  apply fact
  apply simp
  apply (intro conjI exI)
  apply fact+
  done
with (closed ?t_n) have closed u'
  by (blast intro:rules-as-nrules.nrewrite-closed)
with (t' = nterm-to-term' ⊢ u' ≈_α ?t_n')
  by (force intro: nterm-to-term-term-to-nterm [where Γ = [] and Γ' = [],simplified])

show ?case
  apply (intro conjI exI)
  apply (rule nrewrite.step)
  apply fact
  apply simp
  apply (intro conjI exI)
apply fact+
done
next
case (beta t t')
let ?name = next s
let ?t_n = fst (run-state (term-to-nterm ?name) t) (?name)
let ?t_n' = fst (run-state (term-to-nterm [] t') (snd (run-state (term-to-nterm [?name] t) (?name))))

from beta have closed t closed (t [t'β]) closed (?t_n $ t') closed ?t_n'
using replace-bound-frees
by fastforce+
moreover from beta have wellformed' (Suc 0) t wellformed t'
by simp+
ultimately have t = nterm-to-term [?name] ?t_n
and t' = nterm-to-term' ?t_n'
and *:frees ?t_n = {?name} ∨ frees ?t_n = fempty
and closed ?t_n'
using term-to-nterm-vars0[where Γ = [?name]]
using term-to-nterm-vars0[where Γ = []]
by (force simp: term-to-nterm-nterm-to-term0)+

hence **: t [t'β] = nterm-to-term' (subst-single ?t_n ?name ?t_n')
by (auto simp: nterm-to-term-subst-replace-bound[where n = 0])

from ⟨closed ?t_n'⟩ have closed-env (fmap-of-list [(?name, ?t_n')])
by auto

show ?case
apply (rule exI)
apply (auto simp: split-beta create-alt-def)
apply (rule rewrite.beta)
apply (subst subst-single-eq[symmetric])
apply (subst **) apply (rule nterm-to-term-term-to-nterm[where Γ = [] and Γ' = [], simplified])
apply (subst subst-single-eq) apply (subst subst-frees[OF ⟨closed-env -⟩])
using * by force

next
case (fun t t' u)
hence closed t closed u closed (t $ u)
and wellformed t wellformed u
by fastforce+
from fun obtain u'
where compile ∀n fst (run-state (term-to-nterm [] t) s) → u'
u' ≈_a fst (run-state (term-to-nterm [] t') s')
by force
show ?case
apply (rule exI)
apply (auto simp: split-beta create-alt-def)
apply (rule nrewrite.fun)
apply fact
apply rule
apply fact
apply (subst term-to-nterm-alpha-equiv[of [], simplified])
using (closed u); (wellformed u) by auto

next

  case (arg u u' t)

  hence closed t closed u closed (t $ u)
    and wellformed:wellformed t wellformed u
    by fastforce+

  from arg obtain t'

    where compile |-n fst (run-state (term-to-nterm [] u) (snd (run-state (term-to-nterm [] t) s))) t'

    t' ≈α fst (run-state (term-to-nterm [] u') (snd (run-state (term-to-nterm [] t) s')))

    by force

  show ?case
    apply (rule exI)
    apply (auto simp: split-beta create-alt-def)
    apply rule
    apply fact
    apply rule
    apply (subst term-to-nterm-alpha-equiv[of [], simplified])
    using (closed t); (wellformed t) apply force+
    by fact

lemmata compile-complete:
  assumes rs |- t ----> t' closed t wellformed t
  obtains u' where compile |-n term-to-nterm' t ----> u' u' ≈α term-to-nterm' t'

unfolding term-to-nterm'-'def
using assms by (metis compile-complete0)

end

end

3.2.6 Splitting into constants

type-synonym crules = (term list × nterm) fset

type-synonym crule-set = (name × crules) fset

abbreviation arity-compatibles :: (term list × 'a) fset ⇒ bool where
arity-compatibles ≡ fpairwise (λ(pats1, -) (pats2, -). length pats1 = length pats2)

lemmata arity-compatible-length:
  assumes arity-compatibles rs (pats, rhs) |∈| rs
shows \( \text{length} \ \text{pats} = \text{arity} \ \text{rs} \)

proof

have \( \text{fBall} \ \text{rs} \ (\lambda (\text{pats}_1, \cdot). \ \text{fBall} \ \text{rs} \ (\lambda (\text{pats}_2, \cdot). \ \text{length} \ \text{pats}_1 = \text{length} \ \text{pats}_2)) \)

using \( \text{assms} \) unfolding \( \text{fpairwise-alt-def} \) by blast

hence \( \text{fBall} \ \text{rs} \ (\lambda x. \ \text{fBall} \ \text{rs} \ (\lambda y. \ (\text{length} \circ \text{fst}) \ x = (\text{length} \circ \text{fst}) \ y)) \)

by force

hence \( (\text{length} \circ \text{fst}) \ (\text{pats}, \text{rhs}) = \text{arity} \ \text{rs} \)

using \( \text{assms} \) unfolding \( \text{arity-def} \ \text{fthe-elem} \cdot \text{eq} \) by (rule \( \text{singleton-fset-holds} \))

thus \( \text{thesis} \)

by simp

qed

locale \( \text{pre-crules} = \text{constants} \ \text{C-info} \ \text{fst} \ | \ \text{rs} \) for \( \text{C-info} \) and \( \text{rs} :: \text{crule-set} \)

locale \( \text{crules} = \text{pre-crules} + \)

assumes \( \text{fmap}: \text{is-fmap} \ \text{rs} \)

assumes \( \text{nonempty}: \ \text{rs} \not= \{\} \)

assumes \( \text{inner}: \)

\( \text{fBall} \ \text{rs} \ (\lambda (\cdot, \ \text{crs}). \ \text{arity-compatibles} \ \text{crs} \land \)

\( \text{is-fmap} \ \text{crs} \land \)

\( \text{patterns-compatibles} \ \text{crs} \land \)

\( \text{crs} \not= \{\} \land \)

\( \text{fBall} \ \text{crs} \ (\lambda (\text{pats}, \text{rhs}). \)

\( \text{linears} \ \text{pats} \land \)

\( \text{pats} \not= \[] \land \)

\( \text{fdisjnt} \ (\text{freess} \ \text{pats}) \ \text{all-consts} \land \)

\( \neg \ \text{shadows-consts} \ \text{rhs} \land \)

\( \text{frees} \ \text{rhs} \ \subseteq \text{freess} \ \text{pats} \land \)

\( \text{welldefined} \ \text{rhs}) \)

lemma (in \( \text{pre-crules} \)) \( \text{crulesI} : \)

assumes \( \Lambda \text{name} \ \text{crs}. \ (\text{name}, \ \text{crs}) \ |\ | \ \text{rs} \implies \text{arity-compatibles} \ \text{crs} \)

assumes \( \Lambda \text{name} \ \text{crs}. \ (\text{name}, \ \text{crs}) \ |\ | \ \text{rs} \implies \text{is-fmap} \ \text{crs} \)

assumes \( \Lambda \text{name} \ \text{crs}. \ (\text{name}, \ \text{crs}) \ |\ | \ \text{rs} \implies \text{patterns-compatibles} \ \text{crs} \)

assumes \( \Lambda \text{name} \ \text{crs}. \ (\text{name}, \ \text{crs}) \ |\ | \ \text{rs} \implies \text{crs} \not= \{\} \)

assumes \( \Lambda \text{name} \ \text{crs} \ \text{pats} \ \text{rhs}. \ (\text{name}, \ \text{crs}) \ |\ | \ \text{rs} \implies (\text{pats}, \ \text{rhs}) \ |\ | \ \text{crs} \implies \text{linears} \ \text{pats} \)

assumes \( \Lambda \text{name} \ \text{crs} \ \text{pats} \ \text{rhs}. \ (\text{name}, \ \text{crs}) \ |\ | \ \text{rs} \implies (\text{pats}, \ \text{rhs}) \ |\ | \ \text{crs} \implies \text{pats} \not= \[] \)

assumes \( \Lambda \text{name} \ \text{crs} \ \text{pats} \ \text{rhs}. \ (\text{name}, \ \text{crs}) \ |\ | \ \text{rs} \implies (\text{pats}, \ \text{rhs}) \ |\ | \ \text{crs} \implies \text{fdisjnt} \ (\text{freess} \ \text{pats}) \ \text{all-consts} \)

assumes \( \Lambda \text{name} \ \text{crs} \ \text{pats} \ \text{rhs}. \ (\text{name}, \ \text{crs}) \ |\ | \ \text{rs} \implies (\text{pats}, \ \text{rhs}) \ |\ | \ \text{crs} \implies \neg \ \text{shadows-consts} \ \text{rhs} \)

assumes \( \Lambda \text{name} \ \text{crs} \ \text{pats} \ \text{rhs}. \ (\text{name}, \ \text{crs}) \ |\ | \ \text{rs} \implies (\text{pats}, \ \text{rhs}) \ |\ | \ \text{crs} \implies \text{frees} \ \text{rhs} \ \subseteq \text{freess} \ \text{pats} \)

assumes \( \Lambda \text{name} \ \text{crs} \ \text{pats} \ \text{rhs}. \ (\text{name}, \ \text{crs}) \ |\ | \ \text{rs} \implies (\text{pats}, \ \text{rhs}) \ |\ | \ \text{crs} \implies \text{welldefined} \ \text{rhs} \)

assumes \( \text{is-fmap} \ \text{rs} \ \text{rs} \not= \{\} \)
shows crules C-info rs
using assms unfolding crules-axioms-def crules-def
by (auto simp: prod-fBallI intro: pre-crules-axioms)

lemmas crulesI[intro!] = pre-crules.crulesI[unfolded pre-crules-def]

definition consts-of :: nrule fset ⇒ crule-set where
consts-of = fgroup-by split-rule

lemma consts-of-heads: fst |
|consts-of rs = heads-of rs
unfolding consts-of-def
by (simp add: split-rule-fst comp-def)

lemma (in nrules) consts-rules: crules C-info (consts-of rs)
proof
have is-fmap rs
using fmap by simp
thus is-fmap (consts-of rs)
unfolding consts-of-def by auto

show consts-of rs ≠ {||}
using nonempty unfolding consts-of-def
by (meson fgroup-by-nonempty)

show constants C-info (fst |
|consts-of rs)
proof
  show fdisjnt (fst |
|consts-of rs) C
  using disjnt by (auto simp: consts-of-heads)
  next
  show distinct all-constructors
  by (fact distinct-ctr)
qed

fix name crs
assume crs: (name, crs) |∈| consts-of rs

thus crs ≠ {||}
unfolding consts-of-def
by (meson femptyE fgroup-by-nonempty-inner)

show arity-compatibles crs patterns-compatibles crs
proof safe
  fix pats1 rhs1 pats2 rhs2
  assume (pats1, rhs1) |∈| crs (pats2, rhs2) |∈| crs
with crs obtain lhs1 lhs2
  where rs: (lhs1, rhs1) |∈| rs (lhs2, rhs2) |∈| rs and
  split: split-rule (lhs1, rhs1) = (name, (pats1, rhs1))
  split-rule (lhs2, rhs2) = (name, (pats2, rhs2))
  unfolding consts-of-def by (force simp: split-beta)
hence arity: arity-compatible \( \text{lhs}_1 \text{ lhs}_2 \)
using arity by (force dest: \text{fpairwiseD})

from \( \text{rs} \) have const: is-const \((\text{fst} (\text{strip-comb} \text{ lhs}_1)) \) is-const \((\text{fst} (\text{strip-comb} \text{ lhs}_2))\)
using all-rules by force+

have name = const-name \((\text{fst} (\text{strip-comb} \text{ lhs}_1))\) name = const-name \((\text{fst} (\text{strip-comb} \text{ lhs}_2))\)
using split by (auto simp: split-beta)
with const have \(\text{fst} (\text{strip-comb} \text{ lhs}_1) = \text{Const name} (\text{fst} (\text{strip-comb} \text{ lhs}_2))\)
by simp

with arity have length \((\text{snd} (\text{strip-comb} \text{ lhs}_1))\) = length \((\text{snd} (\text{strip-comb} \text{ lhs}_2))\)
unfolding arity-compatible-def
by (simp add: split-beta)

with \text{split} show length \text{pats}_1 = length \text{pats}_2
by (auto simp: split-beta)

have pattern-compatible \text{lhs}_1 \text{ lhs}_2
using \text{rs} \text{patterns} by (auto dest: \text{fpairwiseD})
moreover have \text{lhs}_1 = name \text{ pats}_1
using split(1) \text{const}(1) by (auto simp: split-beta)
moreover have \text{lhs}_2 = name \text{ pats}_2
using split(2) \text{const}(2) by (auto simp: split-beta)
ultimately have pattern-compatible \((\text{name \text{ pats}_1} \text{ pats}_1) \) \((\text{name \text{ pats}_2} \text{ pats}_2)\)
by simp
thus pattern-compatible \text{pats}_1 \text{ pats}_2
using \text{length} \text{pats}_1 = \_ by (auto dest: pattern-compatible-combD)
qed

show is-fmap \text{crs}
proof
fix \text{pats} \text{ rhs}_1 \text{ rhs}_2
assume \((\text{pats}, \text{ rhs}_1) \mid \in \text{ crs} (\text{pats}, \text{ rhs}_2) \mid \in \text{ crs}\)
with \text{crs obtain} \text{lhs}_1 \text{ lhs}_2
where \text{rs}: \((\text{lhs}_1, \text{ rhs}_1) \mid \in \text{ rs} (\text{lhs}_2, \text{ rhs}_2) \mid \in \text{ rs and} \)
split: split-rule \((\text{lhs}_1, \text{ rhs}_1) = (\text{name}, (\text{pats}, \text{ rhs}_1))\)
split-rule \((\text{lhs}_2, \text{ rhs}_2) = (\text{name}, (\text{pats}, \text{ rhs}_2))\)
unfolding \text{consts-of-def} by (force simp: split-beta)
have \( \text{lhs}_1 = \text{lhs}_2 \)

proof (rule ccontr)
  assume \( \text{lhs}_1 \neq \text{lhs}_2 \)
  then consider (fst) \( \text{fst} (\text{strip-comb} \ \text{lhs}_1) \neq \text{fst} (\text{strip-comb} \ \text{lhs}_2) \)
  | \( \text{snd} \ \text{snd} (\text{strip-comb} \ \text{lhs}_1) \neq \text{snd} (\text{strip-comb} \ \text{lhs}_2) \)
  by (metis list-snd-snd)
thus False
proof cases
  case \( \text{fst} \)
  moreover have is-const (\( \text{fst} (\text{strip-comb} \ \text{lhs}_1) \)) is-const (\( \text{fst} (\text{strip-comb} \ \text{lhs}_2) \))
  using rs all-rules by force+
  ultimately show ?thesis
  using split const-name-simps by (fastforce simp: split-beta)
next
  case \( \text{snd} \)
  with split show ?thesis
  by (auto simp: split-beta)
qed

qed

with \( \text{rs} \) show \( \text{rhs}_1 = \text{rhs}_2 \)
  using (is-fmap rs) by (auto dest: is-fmapD)
qed

fix \( \text{pats} \ \text{rhs} \)
assume \( (\text{pats}, \ \text{rhs}) \ |\in| \text{crs} \)
then obtain \( \text{lhs} \) where \( (\text{lhs}, \ \text{rhs}) \ |\in| \text{rs} \ \text{pats} = \text{snd} (\text{strip-comb} \ \text{lhs}) \)
  using \( \text{crs} \) unfolding consts-of-def by (force simp: split-beta)
hence \( \text{nrule} \ (\text{lhs}, \ \text{rhs}) \)
  using all-rules by blast
hence \( \text{linear} \ \text{lhs} \ \text{frees} \ \text{rhs} \ |\subseteq| \ \text{frees} \ \text{lhs} \)
  by auto
thus \( \text{linears} \ \text{pats} \)
  unfolding (\( \text{pats} = \_ \)) by (intro linears-strip-comb)

have \( \neg \) is-const \( \text{lhs} \) is-const (\( \text{fst} (\text{strip-comb} \ \text{lhs}) \))
  using (\nrule \( \_ \)) by auto
thus \( \text{pats} \neq [\] \)
  unfolding (\( \text{pats} = \_ \)) using (linear lhs)
  apply (cases \( \text{lhs} \))
    apply (fold app-term-def)
  by (auto split: prod.splits)
from (\nrule (\text{lhs}, \ \text{rhs})\): have \( \text{frees} \ (\text{fst} (\text{strip-comb} \ \text{lhs})) = [\_] \)
  by (cases \( \text{fst} (\text{strip-comb} \ \text{lhs}) \)) (auto simp: is-const-def)
hence \( \text{frees} \ \text{lhs} = \text{freess} \ (\text{snd} (\text{strip-comb} \ \text{lhs})) \)
  by (subst frees-strip-comb) auto
thus \( \text{frees rhs} \subseteq \text{frees pats} \)

unfolding \( \text{pats} = \cdot \) using \( \text{frees rhs} \subseteq \text{frees lhs} \) by simp

have \( \neg \text{shadows-consts rhs} \)

using \( \langle \text{lhs, rhs} \rangle \in rs \) not-shadows

by force

thus \( \neg \text{pre-constants.shadows-consts C-info \langle \text{fst} \mid \cdot \mid \text{consts-of rs} \rangle \text{ rhs} \)

by (simp add: consts-of-heads)

have \( \text{fdisjnt all-consts \langle \text{frees lhs} \rangle} \)

using \( \langle \text{lhs, rhs} \rangle \in rs \) not-shadows

by (force simp: shadows-consts-def)

moreover have \( \text{frees pats} \subseteq \text{frees lhs} \)

unfolding \( \text{pats} = \cdot \) (frees lhs = \cdot)

by simp

ultimately have \( \text{fdisjnt \langle \text{frees pats} \rangle all-consts} \)

by (metis fdisjnt-subset-right fdisjnt-swap)

thus \( \text{fdisjnt \langle \text{frees pats} \rangle \langle \text{pre-constants.all-consts C-info \langle \text{fst} \mid \cdot \mid \text{consts-of rs} \rangle \rangle \)} \)

by (simp add: consts-of-heads)

show \( \text{pre-constants.welldefined C-info \langle \text{fst} \mid \cdot \mid \text{consts-of rs} \rangle \text{ rhs} \)

using welldefined-rs \( \langle \text{lhs, rhs} \rangle \in rs \)

by (force simp: consts-of-heads)

qed

sublocale nrules \( \subseteq \text{nrules-as-crules? \: crules C-info \text{consts-of rs}} \)

by (fact consts-rules)

3.2.7 Computability

export-code

\text{translate-rule consts-of arity nterm-to-term}

\text{checking Scala}

end

3.3 Higher-order term rewriting with explicit pattern matching

theory Rewriting-Pterm-Elim

imports

Rewriting-Nterm

../../Terms/Pterm

begin

3.3.1 Intermediate rule sets

type-synonym irules = \text{(term list \times pterm) fset}
type-synonym irule-set = (name × irules) fset

locale pre-irules = constants C-info fst |'| rs for C-info and rs :: irule-set

locale irules = pre-irules +
  assumes fmap: is-fmap rs
  assumes nonempty: rs ≠ {||}
  assumes inner:
    fBall rs (λ(-, irs).
      arity-compatibles irs ∧
      is-fmap irs ∧
      patterns-compatibles irs ∧
      irs ≠ {||} ∧
      fBall (λ(pats, rhs).
        linears pats ∧
        abs-ish pats rhs ∧
        closed-except rhs (freess pats) ∧
        fdisjnt (freess pats) all-consts ∧
        wellformed rhs ∧
        ~ shadows-consts rhs ∧
        welldefined rhs))

lemma (in pre-irules) irulesI:
  assumes ∧name irs. (name, irs) |∈| rs ⇒ arity-compatibles irs
  assumes ∧name irs. (name, irs) |∈| rs ⇒ is-fmap irs
  assumes ∧name irs. (name, irs) |∈| rs ⇒ patterns-compatibles irs
  assumes ∧name irs. (name, irs) |∈| rs ⇒ irs ≠ {||}
  assumes ∧name irs pats rhs. (name, irs) |∈| rs ⇒ (pats, rhs) |∈| irs ⇒
    linears pats
  assumes ∧name irs pats rhs. (name, irs) |∈| rs ⇒ (pats, rhs) |∈| irs ⇒
    abs-ish pats rhs
  assumes ∧name irs pats rhs. (name, irs) |∈| rs ⇒ (pats, rhs) |∈| irs ⇒
    fdisjnt (freess pats) all-consts
  assumes ∧name irs pats rhs. (name, irs) |∈| rs ⇒ (pats, rhs) |∈| irs ⇒
    closed-except rhs (freess pats)
  assumes ∧name irs pats rhs. (name, irs) |∈| rs ⇒ (pats, rhs) |∈| irs ⇒
    wellformed rhs
  assumes ∧name irs pats rhs. (name, irs) |∈| rs ⇒ (pats, rhs) |∈| irs ⇒
    ~ shadows-consts rhs
  assumes ∧name irs pats rhs. (name, irs) |∈| rs ⇒ (pats, rhs) |∈| irs ⇒
    welldefined rhs
  assumes is-fmap rs rs ≠ {||}
  shows irules C-info rs
using assms unfolding irules-axioms-def irules-def
by (auto simp: prod-fBallI intro: pre-irules-axioms)
lemmas irulesI[intro!] = pre-irules.irulesI[unfolded pre-irules-def]
Translation from \textit{nterm} to \textit{pterm}

\textbf{fun} \texttt{nterm-to-pterm :: nterm \to pterm} \textbf{where}
\begin{align*}
\texttt{nterm-to-pterm} (\texttt{Nvar} s) &= \texttt{Pvar} s \\
\texttt{nterm-to-pterm} (\texttt{Nconst} s) &= \texttt{Pconst} s \\
\texttt{nterm-to-pterm} (t_1 \texttt{\$} n t_2) &= \texttt{nterm-to-pterm} t_1 \texttt{\$} p \texttt{nterm-to-pterm} t_2 \\
\texttt{nterm-to-pterm} (\Lambda x. t) &= (\Lambda p x. \texttt{nterm-to-pterm} t)
\end{align*}

\textbf{lemma} \texttt{nterm-to-pterm-inj}: \texttt{nterm-to-pterm} \(x = \texttt{nterm-to-pterm} y \Rightarrow x = y\)
\textbf{by} (induction \(y\) arbitrary; \(x\)) (auto elim: \texttt{nterm-to-pterm.elims})

\textbf{lemma} \texttt{nterm-to-pterm}:
\begin{itemize}
\item \textbf{assumes} \texttt{no-abs} \(t\)
\item \textbf{shows} \texttt{nterm-to-pterm} \(t = \text{convert-term} t\)
\item \textbf{using} \texttt{assms}
\item \textbf{apply} induction
\item \textbf{apply} auto
\item \textbf{by} (auto simp: \texttt{free-nterm-def} \texttt{free-pterm-def} \texttt{const-nterm-def} \texttt{const-pterm-def} \texttt{app-nterm-def} \texttt{app-pterm-def})
\end{itemize}

\textbf{lemma} \texttt{nterm-to-pterm-frees[simp]}: \texttt{frees} (\texttt{nterm-to-pterm} \(t\)) = \texttt{frees} \(t\)
\textbf{by} (induct \(t\)) auto

\textbf{lemma} \texttt{closed-nterm-to-pterm[intro]}: \texttt{closed-except} (\texttt{nterm-to-pterm} \(t\)) (\texttt{frees} \(t\))
\textbf{unfolding} \texttt{closed-except-def} \textbf{by} simp

\textbf{lemma} \texttt{(in constants) shadows-nterm-to-pterm[simp]}: \texttt{shadows-consts} (\texttt{nterm-to-pterm} \(t\)) = \texttt{shadows-consts} \(t\)
\textbf{by} (induct \(t\)) (auto simp: \texttt{shadow-consts-def} \texttt{fdisj-alt-def})

\textbf{lemma} \texttt{wellformed-nterm-to-pterm[intro]}: \texttt{wellformed} (\texttt{nterm-to-pterm} \(t\))
\textbf{by} (induct \(t\)) auto

\textbf{lemma} \texttt{consts-nterm-to-pterm[simp]}: \texttt{consts} (\texttt{nterm-to-pterm} \(t\)) = \texttt{consts} \(t\)
\textbf{by} (induct \(t\)) auto

\textbf{Translation from} \textit{crule-set} to \textit{irule-set}

\textbf{definition} \texttt{translate-crules :: crules \to irules} \textbf{where}
\texttt{translate-crules} = \texttt{fimage} (\texttt{map-prod} \texttt{id} \texttt{nterm-to-pterm})

\textbf{definition} \texttt{compile :: crule-set \to irule-set} \textbf{where}
\texttt{compile} = \texttt{fimage} (\texttt{map-prod} \texttt{id} \texttt{translate-crules})

\textbf{lemma} \texttt{compile-heads}: \texttt{fst} [\cdot] \texttt{compile} \(rs\) = \texttt{fst} [\cdot] \(rs\)
\textbf{unfolding} \texttt{compile-def} \textbf{by} simp

\textbf{lemma} \texttt{(in crules) compile-rules}: \texttt{irules} \texttt{C-info} (\texttt{compile} \(rs\))
\textbf{proof}
\begin{itemize}
\item \textbf{have} \texttt{is-fmap} \(rs\)
\end{itemize}

98
using fmap by simp
thus is-fmap (compile rs)
  unfolding compile-def map-prod-def id-apply by (rule is-fmap-image)

show compile rs ≠ {||}
  using nonempty unfolding compile-def by auto

show constants C-info (fst |'| compile rs)
proof
  show fdisjnt (fst |'| compile rs) C
    using disjnt unfolding compile-def
    by force
next
  show distinct all-constructors
    by (fact distinct-ctr)
qed

fix name irs
assume irs: (name, irs) |∈| compile rs
then obtain irs′ where (name, irs′) |∈| rs irs = translate-crules irs′
  unfolding compile-def by force
hence arity-compatibles irs′
  using inner by (blast dest: fpairwiseD)
thus arity-compatibles irs
  unfolding (irs = translate-crules irs) translate-crules-def
  by (force dest: fpairwiseD)

have patterns-compatibles irs′
  using ⟨(name, irs′) |∈| rs⟩ inner
  by (blast dest: fpairwiseD)
thus patterns-compatibles irs
  unfolding (irs = -) translate-crules-def
  by (auto dest: fpairwiseD)

have is-fmap irs′
  using ⟨(name, irs′) |∈| rs⟩ inner by auto
thus is-fmap irs
  unfolding (irs = translate-crules irs) translate-crules-def map-prod-def id-apply
  by (rule is-fmap-image)

have irs′ ≠ {||}
  using ⟨(name, irs′) |∈| rs⟩ inner by auto
thus irs ≠ {||}
  unfolding (irs = translate-crules irs) translate-crules-def by simp

fix pats rhs
assume (pats, rhs) |∈| irs
then obtain rhs′ where (pats, rhs′) |∈| irs′ rhs = nterm-to-pterm rhs′
  unfolding (irs = translate-crules irs) translate-crules-def by force
hence $\text{linear}\ pats\ pats \neq \emptyset \implies \text{frees} \\vdash \text{shadows-consts} \\vdash \text{rhe}\'$
using $\text{fbspec}[OF \ inner \ (\text{name}, \ irs') |\in| rs]$ by blast+

show $\text{linear} \ pats$ by fact
show $\text{closed-except}\ rh\ (\text{freess} \ pats)$
  unfolding $rh = nterm-to-pterms \ rh'$
  using $\text{frees} \ rh' \subseteq \text{freess} \ pats$
  by $(\text{metis dual-order.trans closed-nterm-to-pterms closed-except-def})$

show $\text{wellformed} \ rh$
  unfolding $\langle rh = nterm-to-pterms rh' \rangle$ by auto

have $\text{fdisjnt} (\text{freess} \ pats) \\text{all-consts}$
  using $\langle \text{pats}, \ rh' \rangle |\in| \ irs' \langle \text{name}, \ irs' \rangle |\in| rs \rangle \ inner$
  by blast
thus $\text{fdisjnt} (\text{freess} \ pats) \ (\text{pre-constants.all-consts} C\text{-info} (\text{fst} |\cdot| \text{compile} \ rs))$
  unfolding $\text{compile-def}$ by simp

have $\neg \text{shadows-consts} \ rh$
  unfolding $\langle rh = \cdot \rangle$ using $\langle \neg \text{shadows-consts} \rangle \text{-} \text{by simp}$
thus $\neg \text{pre-constants.shadows-consts} C\text{-info} (\text{fst} |\cdot| \text{compile} \ rs) \ rh$
  unfolding $\text{compile-heads}$

show $\text{abs-ish} \ pats \ rh$
  using $\langle \text{pats} \neq \emptyset \rangle$ unfolding $\text{abs-def}$ by simp

have $\text{welldefined} \ rh'$
  using $\text{fbspec}[OF \ inner \ (\text{name}, \ irs') |\in| rs), \text{simplified}]$
  using $\langle \text{pats}, \ rh' \rangle |\in| irs'$
  by blast

thus $\text{welldefined} (\text{pre-constants.welldefined} C\text{-info} (\text{fst} |\cdot| \text{compile} \ rs) \ rh$
  unfolding $\text{compile-def}$ $\langle rh = \cdot \rangle$ by simp
qed

sublocale $\text{crules} \subseteq \text{crules-as-irules}: \text{irules} C\text{-info} \ \text{compile} \ rs$
  by $(\text{fact compile-rules})$

Transformation of irule-set

definition transform-irules :: irules $\Rightarrow$ irules where
transform-irules $rs = (
  if \text{arity} \ rs = 0 \ then \ rs$
  else $\text{map-prod} \ \text{id} \ Pabs \ |\cdot| \ fgroup-by (\lambda (\text{pats}, \ rh). (\text{butlast} \ \text{pats}, (\text{last} \ \text{pats}, \ \text{rhs}))) \ \text{rs}$)

lemma $\text{arity-compatibles-transform-irules}$:
assumes \textit{arity-compatibles rs}
shows \textit{arity-compatibles (transform-irules rs)}

proof (cases \textit{arity rs} = 0)
  case \textit{True}
  thus \textit{?thesis}
    unfolding \textit{transform-irules-def} using \textit{assms} by simp
  
next
  case \textit{False}
  let \textit{?rs'} = \textit{transform-irules rs}
  let \textit{?f} = \lambda (\textit{pats, rhs}). (\textit{butlast pats, (last pats, rhs)})
  let \textit{?grp} = \textit{fgroup-by} \textit{?f} \textit{rs}
  have \textit{rs':} \textit{?rs'} = \textit{map-prod id Pabs} |\textit{?grp} |
    using \textit{False} unfolding \textit{transform-irules-def} by simp
  show \textit{?thesis}
    proof safe
      fix \textit{pats} \textit{1} \textit{rhs} \textit{1} \textit{pats} \textit{2} \textit{rhs} \textit{2}
      assume \textit{(pats}_1\textit{, rhs}_1\textit{, pats}_2\textit{, rhs}_2\textit{)} |\textit{?rs'} |\textit{?grp} |
      then obtain \textit{rhs}_1\textit{'} \textit{?rs}_2\textit{'} where \textit{(pats}_1\textit{, rhs}_1\textit{, pats}_2\textit{, rhs}_2\textit{'}) |\textit{?grp} |
        unfolding \textit{rs'} by auto
      then obtain \textit{pats}_1\textit{'} \textit{pats}_2\textit{'} \textit{x y} — dummies
        where \textit{fst (\textit{?f (pats}_1\textit{, x)}) = pats}_1\textit{ (pats}_1\textit{', x) |\textit{rs} |\textit{?grp} |\textit{?grp} |
          and \textit{fst (\textit{?f (pats}_2\textit{, y)}) = pats}_2\textit{ (pats}_2\textit{', y) |\textit{rs} |\textit{?grp} |
          by (fastforce simp: split-beta elim: \textit{fgroup-byE2})
        hence \textit{pats}_1\textit{ = butlast pats}_1\textit{'} \textit{pats}_2\textit{ = butlast pats}_2\textit{'} \textit{length pats}_1\textit{'} = \textit{length pats}_2\textit{'}
        using \textit{assms} by (force dest: \textit{fpairwiseD})+
      thus \textit{length pats}_1\textit{ = length pats}_2\textit{'}
        by auto
      qed

    qed

lemma \textit{arity-transform-irules}:
  assumes \textit{arity-compatibles rs rs} \neq \{\}
  shows \textit{arity (transform-irules rs) = (if arity rs = 0 then 0 else arity rs - 1)}
  proof (cases \textit{arity rs} = 0)
    case \textit{True}
    thus \textit{?thesis}
      unfolding \textit{transform-irules-def} by simp
  
next
  case \textit{False}
  let \textit{?f} = \lambda (\textit{pats, rhs}). (\textit{butlast pats, (last pats, rhs)})
  let \textit{?grp} = \textit{fgroup-by} \textit{?f} \textit{rs}
  let \textit{?rs'} = \textit{map-prod id Pabs} |\textit{?grp} |\textit{?rs} |
  have \textit{arity ?rs'} = \textit{arity rs} - 1
    proof (rule \textit{arityI})
      show \textit{fBall ?rs'} (\lambda (\textit{pats, -}). \textit{length pats = arity rs - 1})
        proof (rule \textit{prod-fBallI})
          fix \textit{pats} \textit{rhs}

101
assume \((pats, rhs) \in \mathbb{?}rs\)
then obtain \(cs\) where \((pats, cs) \in \mathbb{?}grp\) \(rhs = Pabs\) \(cs\)
by force
then obtain \(pats'\) \(x\) — dummy
where \(pats = butlast\) \(pats'\) \((pats', x) \in \mathbb{?}rs\)
by \((fastforce\ simp\ split-beta\ elim: fgroup-byE2)\)
hence \(\text{length}\ pats' = \text{arity}\ rs\)
using \(\text{assms}\) by \((\text{metis}\ \text{arity-compatible-length})\)
thus \(\text{length}\ pats = \text{arity}\ rs - 1\)
unfolding \(\langle pats = butlast\ pats' \rangle\) using \(\text{False}\) by simp
qed
next
show \(\mathbb{?}rs' \neq \{\}\)
using \(\text{assms}\) by \((\text{simp}\ add: fgroup-by-nonempty)\)
qed
with \(\text{False}\) show \(\mathbb{?}\text{thesis}\)
unfolding transform-irules-def by simp
qed

**definition**

\(\text{transform-irule-set} :: \text{irule-set} \Rightarrow \text{irule-set}\) where
\(\text{transform-irule-set} = \text{fimage}\) \((\text{map-prod}\ \text{id}\ \text{transform-irules})\)

**lemma**

\(\text{transform-irule-set-heads}: \text{fst}\ [\cdot]\ \text{transform-irule-set}\ \text{rs} = \text{fst}\ [\cdot]\ \text{rs}\)

unfolding \(\text{transform-irule-set-def}\) by simp

**lemma** \((\text{in}\ \text{irules})\) \(\text{rules-transform}: \text{irules}\ \mathbb{C}\)-\text{info} \((\text{transform-irule-set}\ \text{rs})\)

proof

have \(\text{is-fmap}\ \text{rs}\)
using \(\text{fmap}\) by simp
thus \(\text{is-fmap}\) \((\text{transform-irule-set}\ \text{rs})\)
unfolding \(\text{transform-irule-set-def}\) \((\text{map-prod-def}\ \text{id}-\text{apply})\) by \((\text{rule}\ \text{is-fmap-image})\)

show \(\text{transform-irule-set}\ \text{rs} \neq \{\}\)
using \(\text{nonempty}\) unfolding \(\text{transform-irule-set-def}\) by auto

show \(\text{constants}\ \mathbb{C}\)-\text{info} \((\text{fst}\ [\cdot]\ \text{transform-irule-set}\ \text{rs})\)

proof

show \(\text{disjnt}\) \((\text{fst}\ [\cdot]\ \text{transform-irule-set}\ \text{rs})\) \(\mathbb{C}\)
using \(\text{disjnt}\) unfolding \(\text{transform-irule-set-def}\)
by force
next
show \(\text{distinct}\ \text{all-constructors}\)
by \((\text{fact}\ \text{distinct-ctr})\)
qed

fix \(\text{name}\ \text{irs}\)
assume \(\text{irs}: (\text{name}, \text{irs}) \in \mathbb{?}\text{transform-irule-set}\ \text{rs}\)
then obtain \(\text{irs'}\) where \((\text{name}, \text{irs'}) \in \mathbb{?}\) \(\text{rs}\ \text{irs} = \text{transform-irules}\ \text{irs'}\)

102
unfolding transform-irule-set-def by force

hence arity-compatibles irs'

using inner by (blast dest: fpairwiseD)

thus arity-compatibles irs

unfolding irs = transform-irules irs' by (rule arity-compatibles-transform-irules)

have irs' \neq \{\} using ⟨name, irs'⟩ |∈| rs inner by blast

thus irs \neq \{\}

unfolding ⟨irs = transform-irules irs'⟩ transform-irules-def

by (simp add: fgroup-by-nonempty)

let \( ?f = \lambda (pats, rhs). (\text{butlast pats}, (\text{last pats}, rhs)) \)

let \( ?\text{grp} = \text{fgroup-by} \ ?f \)

have patterns-compatibles irs'

using ⟨name, irs'⟩ |∈| rs inner by (blast dest: fpairwiseD)

show patterns-compatibles irs

proof (cases arity irs' = 0)

case True

thus \?thesis

unfolding ⟨irs = transform-irules irs'⟩ transform-irules-def

using ⟨patterns-compatibles irs'⟩ by simp

next

case False

hence irs': irs = map-prod id Pabs |\} \ ?\text{grp}

unfolding ⟨irs = transform-irules irs'⟩ transform-irules-def by simp

show \?thesis

proof safe

  fix pats_1 rhs_1 pats_2 rhs_2

  assume ⟨pats_1, rhs_1⟩ |∈| irs ⟨pats_2, rhs_2⟩ |∈| irs

  with irs' obtain cs_1 cs_2 where ⟨pats_1, cs_1⟩ |∈| \?\text{grp} ⟨pats_2, cs_2⟩ |∈| \?\text{grp}

  by force

  then obtain pats_1', pats_2' and \ x \ y \ — \ dummies

  where ⟨pats_1', x⟩ |∈| irs' (pats_2', y) |∈| irs'

  and pats_1 = \text{butlast} pats_1' pats_2 = \text{butlast} pats_2'

  unfolding irs'

  by (fastforce elim: fgroup-byE2)

  hence patterns-compatible pats_1' pats_2'

  using ⟨patterns-compatibles irs'⟩ by (auto dest: fpairwiseD)

  thus patterns-compatible pats_1 pats_2

  unfolding ⟨pats_1 = \_⟩ ⟨pats_2 = \_⟩

  by auto

  qed

qed

have is-fmap irs'
using \langle \text{name, irs'} \mid \in \mid \text{rs} \rangle \text{inner by blast}

show is-fmap irs

proof (\text{cases arity irs'} = 0)

case True

thus \text{thesis}

unfolding \langle \text{irs = transform-irules irs'} \rangle \text{transform-irules-def}

using \langle is-fmap irs' \rangle \text{by simp}

case False

hence irs': \text{irs = map-prod \text{id} Pabs} \mid \text{\?grp}

unfolding \langle \text{irs = transform-irules irs'} \rangle \text{transform-irules-def} \text{by simp}

show \text{thesis}

proof

fix \text{pats rhs1 rhs2}

assume \langle \text{pats, rhs1} \mid \in \mid \text{irs} \rangle \langle \text{pats, rhs2} \mid \in \mid \text{irs} \rangle

with \text{irs'}

obtain \text{cs1 cs2}

where \langle \text{pats, cs1} \mid \in \mid \text{?grp rhs1 = Pabs cs1} \rangle

and \langle \text{pats, cs2} \mid \in \mid \text{?grp rhs2 = Pabs cs2} \rangle

by \text{force}

moreover have is-fmap \text{?grp}

by \text{auto}

ultimately show \text{rhs1 = rhs2}

by (\text{auto dest: is-fmapD})

qed

qed

fix \text{pats rhs}

assume \langle \text{pats, rhs} \mid \in \mid \text{irs} \rangle

show linears \text{pats}

proof (\text{cases arity irs'} = 0)

case True

thus \text{thesis}

using \langle \text{pats, rhs} \mid \in \mid \text{irs} \rangle \langle \text{name, irs'} \mid \in \mid \text{rs} \rangle \text{inner}

unfolding \langle \text{irs = transform-irules irs'} \rangle \text{transform-irules-def}

by \text{\text{smt fBallE split-conv}}

next

case False

hence irs': \text{irs = map-prod \text{id} Pabs} \mid \text{\?grp}

unfolding \langle \text{irs = transform-irules irs'} \rangle \text{transform-irules-def} \text{by simp}

then obtain \text{cs where} \langle \text{pats, cs} \mid \in \mid \text{?grp} \rangle

using \langle \text{pats, rhs} \mid \in \mid \text{irs} \rangle \text{by force}

then obtain \text{pats' \text{x \text{-- dummy}}}

where \text{fst} \langle \text{\text{?f (pats', x)} = pats (pats', x) \mid \in \mid \text{irs'} \rangle}

by (\text{fastforce simp: split-beta elim: fgroup-byE2})

hence \text{pats = butlast pats'}

by \text{simp}

moreover have linears \text{pats'}
using \(\langle \text{pats}', x \rangle \mid \epsilon \| \text{irs} \rangle \langle \text{name}, \text{irs}' \rangle \mid \epsilon \| \text{rs} \rangle \text{inner}\)
by blast
ultimately show ?thesis
by auto
qed

have \(\text{fdisjnt (freess pats) all-consts}\)
proof (cases arity \(\text{irs}' = 0\))
case True then ?thesis
using \(\langle \text{pats}, \text{rhs} \rangle \mid \epsilon \| \text{irs} \rangle \langle \text{name}, \text{irs}' \rangle \mid \epsilon \| \text{rs} \rangle \text{inner}\)
unfolding \(\text{irs} = \text{transform-irules} \text{irs}' \text{transform-irules-def}\)
by (smt \text{fBallE} split-conv)
next
case False hence \(\text{irs}': \text{irs} = \text{map-prod} \text{id} \text{Pabs} \mid \epsilon \) ?grp
unfolding \(\text{irs} = \text{transform-irules} \text{irs}' \text{transform-irules-def}\) by simp
then obtain \(\text{cs where} \ (\text{pats}, \text{cs}) \mid \epsilon \| \text{irs} \rangle \) ?grp
using \(\langle \text{pats}, \text{rhs} \rangle \mid \epsilon \| \text{irs} \rangle \text{by force}\)
then obtain \(\text{pats}' x — \text{dummy}\)
where \(\text{fst} (\?f (\text{pats}', x)) = \text{pats} (\text{pats}', x) \mid \epsilon \| \text{irs}'\)
by (fastforce simp; split-beta elim: \text{fgroup-byE2})
hence \(\text{pats} = \text{butlast} \text{pats}'\)
by simp
moreover have \(\text{fdisjnt (freess pats') all-consts}\)
using \(\langle \text{pats}', x \rangle \mid \epsilon \| \text{irs}' \rangle \langle \text{name}, \text{irs}' \rangle \mid \epsilon \| \text{rs} \rangle \text{inner}\) by blast
ultimately show ?thesis
by (metis subsetI in-set-butlastD freess-subset fdisjnt-subset-left)
qed

thus \(\text{fdisjnt (freess pats) (pre-constants.all-consts C-info} (\text{fst} \mid \epsilon \text{transform-irule-set} \text{rs}))\)
unfolding \text{transform-irule-set-def} by simp

show closed-except \text{rhs (freess pats)}
proof (cases arity \(\text{irs}' = 0\))
case True then ?thesis
using \(\langle \text{pats}, \text{rhs} \rangle \mid \epsilon \| \text{irs} \rangle \langle \text{name}, \text{irs}' \rangle \mid \epsilon \| \text{rs} \rangle \text{inner}\)
unfolding \(\text{irs} = \text{transform-irules} \text{irs}' \text{transform-irules-def}\)
by (smt \text{fBallE} split-conv)
next
case False hence \(\text{irs}': \text{irs} = \text{map-prod} \text{id} \text{Pabs} \mid \epsilon \) ?grp
unfolding \(\text{irs} = \text{transform-irules} \text{irs}' \text{transform-irules-def}\) by simp
then obtain \(\text{cs where} \ (\text{pats}, \text{cs}) \mid \epsilon \mid ?\text{grp} = \text{Pabs cs}\)
using \(\langle \text{pats}, \text{rhs} \rangle \mid \epsilon \| \text{irs} \rangle \text{by force}\)

show ?thesis

105
unfolding ⟨rhs = Pabs cs⟩ closed-except-simps
proof safe
fix pat t
assume (pat, t) ∈ cs
then obtain pats' where (pats', t) ∈ irs' ?f (pats', t) = (pats, (pat, t))
  using ⟨(pats, cs) ∈ ?grp⟩ by auto
hence closed-except t (freess pats')
  using ⟨(name, irs') ∈ rs⟩ inner by blast
have pats' ≠ []
  using ⟨arity-compatibles irs' (pats', t) ∈ irs'⟩ False
by ⟨metis list.size(3) arity-compatible-length⟩
hence pats' = pats @ [pat]
  using ⟨?f (pats', t) = (pats, (pat, t))⟩
by ⟨fastforce simp; split-beta snoc-eq-iff-butlast⟩
hence freess pats |∪| freess pat = freess pats'
unfolding freess-def by auto
thus closed-except t (freess pats |∪| freess pat)
  using ⟨closed-except t (freess pats')⟩ by simp
qed
qed

show wellformed rhs
proof (cases arity irs' = 0)
case True
  thus ?thesis
  using ⟨(pats, rhs) ∈ irs ⟩ ⟨(name, irs') ∈ rs⟩ inner
  unfolding ⟨irs = transform-irules irs'⟩ ⟨transform-irules-def⟩
  by ⟨smt fBallE split-conv⟩
next
case False
  hence irs': irs = map-prod id Pabs | | ?grp
unfolding ⟨irs = transform-irules irs'⟩ ⟨transform-irules-def⟩ by simp
then obtain cs where (pats, cs) ∈ ?grp rhs = Pabs cs
  using ⟨(pats, rhs) ∈ irs⟩ by force
show ?thesis
unfolding ⟨rhs = Pabs cs⟩
proof (rule wellformed-PabsI)
show cs ≠ [||]
  using ⟨(pats, cs) ∈ ?grp ⟩ ⟨irs' ≠ [||]⟩
  by ⟨meson femptyE fgroup-by-nonempty-inner⟩
next
show is-fmap cs
proof
fix pat t₁ t₂
assume (pat, t₁) ∈ cs (pat, t₂) ∈ cs
then obtain pats₁' pats₂'
where \((\text{pats}_1', \text{t}_2) \in \text{irs}' \not\mathrel{\hat{=} } (\text{pats}_2', \text{t}_2) = (\text{pats}, (\text{pat}, \text{t}_1))\)
and \((\text{pats}_2', \text{t}_2) \in \text{irs}' \not\mathrel{\hat{=} } (\text{pats}_2', \text{t}_2) = (\text{pats}, (\text{pat}, \text{t}_2))\)

using \((\text{pats}, \text{cs}) \in \text{?grp}\) by force

moreover hence \(\text{pats}_1' \not\mathrel{\hat{=} } [] \) \(\text{pats}_2' \not\mathrel{\hat{=} } []\)

using \(\text{arity-compatibles irs}' False\)

unfolding prod.case
by (metis list.size(3) arity-compatible-length)+

ultimately have \(\text{pats}_1' = \text{pats} @ [\text{pat}] \) \(\text{pats}_2' = \text{pats} @ [\text{pat}]\)

unfolding split-beta fst-cone snd-conv
by (metis prod.inject snoc-eq-iff-butlast)+

with \(\text{is-fmap irs}'\) show \(\text{t}_1 = \text{t}_2\)

using \(\langle \text{pats}_1', \text{t}_1 \rangle \in \text{irs}' (\text{pats}_2', \text{t}_2) \in \text{irs}'\)

by (blast dest: is-fmapD)

qed

next

show \(\text{pattern-compatibles cs}\)

proof safe

fix \(\text{pat}_1 \) \(\text{rhs}_1 \) \(\text{pat}_2 \) \(\text{rhs}_2 \)

assume \((\text{pats}_1, \text{rhs}_1) \in \text{cs} (\text{pat}_2, \text{rhs}_2) \in \text{cs}\)

then obtain \(\text{pats}_1' \) \(\text{pats}_2'\)

where \((\text{pats}_1', \text{rhs}_1) \in \text{irs}' \not\mathrel{\hat{=} } (\text{pats}_1', \text{rhs}_1) = (\text{pats}, (\text{pat}_1, \text{rhs}_1))\)

and \((\text{pats}_2', \text{rhs}_2) \in \text{irs}' \not\mathrel{\hat{=} } (\text{pats}_2', \text{rhs}_2) = (\text{pats}, (\text{pat}_2, \text{rhs}_2))\)

using \((\text{pats}, \text{cs}) \in \text{?grp}\)

by force

moreover hence \(\text{pats}_1' \not\mathrel{\hat{=} } [] \) \(\text{pats}_2' \not\mathrel{\hat{=} } []\)

using \(\text{arity-compatibles irs}' False\)

unfolding prod.case
by (metis list.size(3) arity-compatible-length)+

ultimately have \(\text{pats}_1' = \text{pats} @ [\text{pat}_1] \) \(\text{pats}_2' = \text{pats} @ [\text{pat}_2]\)

unfolding split-beta fst-cone snd-conv
by (metis prod.inject snoc-eq-iff-butlast)+

moreover have \(\text{patterns-compatible pat}_1 \) \(\text{pat}_2\)

using \(\langle \text{pats}_1', \text{rhs}_1 \rangle \in \text{irs}' (\text{pats}_2', \text{rhs}_2) \in \text{irs}' (\text{patterns-compatibles irs}')\)

by (auto dest: fpairwiseD)

ultimately show \(\text{pattern-compatible pat}_1 \) \(\text{pat}_2\)

by (auto elim: rev-accum-rel-snoc-eqE)

qed

next

fix \(\text{pat} \)

assume \((\text{pat}, \text{t}) \in \text{cs}\)

then obtain \(\text{pats}'\) where \((\text{pats}', \text{t}) \in \text{irs}' \text{pat} = \text{last pats}'\)

using \((\text{pats}, \text{cs}) \in \text{?grp}\) by auto

moreover hence \(\text{pats}' \not\mathrel{\hat{=} } []\)

using \(\text{arity-compatibles irs}' False\)

by (metis list.size(3) arity-compatible-length)

ultimately have \(\text{pat} \in \text{set pats}'\)

by auto
moreover have linear pats'
  using ⟨(pats', t) |∈| irs'⟩ (name, irs') |∈| rs inner by blast
ultimately show linear pat
  by (metis linear-linear)

show wellformed t
  using ⟨(pats', t) |∈| irs'⟩ (name, irs') |∈| rs inner by blast
qed

have ¬ shadows-consts rhs
proof (cases arity irs' = 0)
  case True
  thus ?thesis
  using ⟨(pats, rhs) |∈| irs⟩ (name, irs') |∈| rs⟩ inner
  unfolding ⟨irs = transform-irules irs'⟩ transform-irules-def
  by (smt fBallE split-conv)
next
  case False
  hence irs': irs = map-prod id Pabs |?| ?grp
  unfolding ⟨irs = transform-irules irs'⟩ transform-irules-def by simp
then obtain cs where ⟨(pats, cs) |∈| ?grp⟩ rhs = Pabs cs
  using ⟨(pats, rhs) |∈| irs⟩ by force

show ?thesis
  unfolding ⟨rhs = -⟩
  proof
    assume shadows-consts (Pabs cs)
  then obtain pat t where ⟨(pat, t) |∈| cs⟩ shadows-consts t ∨ shadows-consts pat
    by force
  then obtain pats' where ⟨(pats', t) |∈| irs'⟩ pat = last pats'
    using ⟨(pats, cs) |∈| ?grp⟩ by auto
  moreover hence pats' ≠ []
    using ⟨arity-compatibles irs'⟩ False
    by (metis list.size(3) arity-compatible-length)
ultimately have pat ∈ set pats'
    by auto

show False
  using ⟨shadows-consts t ∨ shadows-consts pat⟩
  proof
    assume shadows-consts t
    thus False
      using ⟨(name, irs') |∈| rs⟩ ⟨(pats', t) |∈| irs'⟩ inner by blast
next
  assume shadows-consts pat

  have fdisjnt (freess pats') all-consts
using ⟨(name, irs′) ∈ rs⟩ ⟨(pats′, t) ∈ irs′⟩ inner by blast
have fdisjnt (frees pat) allconsts
apply (rule fdisjnt-subset-left)
apply (subst freess-single[symmetric])
apply (rule freess-subset)
apply simp
apply fact+
done
thus False
using (shadows-consts pat)
unfolding shadows-consts-def fdisjnt-alt-def by auto
qed
qed
qed
thus ¬ pre-constants.shadows-consts C-info (fst |'| transform-irule-set rs) rhs
by (simp add: transform-irule-set-heads)

show abs-ish pats rhs
proof (cases arity irs′ = 0)
case True
thus ?thesis
using ⟨(pats, rhs) ∈ irs⟩ ⟨(name, irs′) ∈ rs⟩ inner
unfolding ⟨irs = transform-irules irs′⟩ transform-irules-def by (smt fBallE split-conv)
next
case False
hence irs′: irs = map-prod id Pabs |'| ?grp
unfolding ⟨irs = transform-irules irs′⟩ transform-irules-def by simp
then obtain cs where ⟨pats, cs⟩ ∈ ?grp rhs = Pabs cs
using ⟨(pats, rhs) ∈ irs⟩ by force
thus ?thesis
unfolding abs-ish-def by (simp add: is-abs-def term-cases-def)
qed

have welldefined rhs
proof (cases arity irs′ = 0)
case True
hence ⟨(pats, rhs) ∈ irs⟩
using ⟨(pats, rhs) ∈ irs⟩ ⟨(name, irs′) ∈ rs⟩ inner
unfolding ⟨irs = transform-irules irs′⟩ transform-irules-def by (smt fBallE split-conv)
thus ?thesis
unfolding transform-irule-set-def
using fbspec[OF inner ⟨(name, irs′) ∈ rs⟩, simplified]
by force
next
case False
hence irs′: irs = map-prod id Pabs |'| ?grp
unfolding \( \{ \text{irs} = \text{transform-rules} \text{irs} \} \) \text{transform-rules-def} \text{ by simp}

then obtain \( cs \) where \((\text{pats}, cs) \in \) ?grp \( \text{rhs} = \text{Pabs} \text{cs} \)

using \((\text{pats}, \text{rhs}) \in \) \text{irs} \text{ by force}

show \( \? \text{thesis} \)

unfolding \( \{ \text{rhs} = - \} \)

apply simp

apply (rule \text{ffUnion-least})

unfolding \text{ball-simps}

apply (rule \text{rename-tac x, case-tac x, hypsubst-thin})

apply simp

subgoal premises \( \text{prems for pat} \ t \)

proof −

from \( \text{prems} \) obtain \( \text{pats'} \) where \((\text{pats'}, t) \in \) \text{irs'}

using \((\text{pats}, cs) \in \) ?grp \text{ by auto}

hence \text{welldefined} \( t \)

using \text{fbspec}[\text{OF inner} \ (\text{name}, \text{irs'}) \in \text{rs}, \text{simplified}]

by blast

thus \( \? \text{thesis} \)

unfolding \text{transform-rule-set-def}

by simp

qed

done

thus \text{pre-constants} . \text{welldefined} \ C\text{-info} \ (\text{fst} \in \text{transform-rule-set} \text{rs}) \text{ rhs}

unfolding \text{transform-rule-set-heads} .

qed

Matching and rewriting

definition \text{rewrite-step} :: \text{name} \Rightarrow \text{term list} \Rightarrow \text{pterm} \Rightarrow \text{pterm} \Rightarrow \text{pterm option}

where

abbreviation \text{rewrite-step' ::} \text{name} \Rightarrow \text{term list} \Rightarrow \text{pterm} \Rightarrow \text{pterm} \Rightarrow \text{pterm} \Rightarrow \text{bool} \ (-/ -/ -/ -/ -/ [50,0,50] 50) \text{ where}

name, \text{pats, rhs} \vdash i \ t \to u \equiv \text{rewrite-step name pats rhs t} = \text{Some} \ u

lemma \text{rewrite-step1}:

assumes \text{match (name $$ pats \) t} = \text{Some env subst rhs env} = u

shows name, \text{pats, rhs} \vdash i \ t \to u

using \text{assms unfolding} \text{rewrite-step-def by simp}

inductive \text{rewrite} :: \text{irule-set} \Rightarrow \text{pterm} \Rightarrow \text{pterm} \Rightarrow \text{bool} \ (-/ -/ -/ [50,0,50] 50) \text{ for} \ \text{irs}

step: [ \langle \text{name}, \text{rs} \rangle \in \text{irs}; (\text{pats, rhs}) \in \text{rs}; \text{name, pats, rhs} \vdash i \ t \to t' \] \Rightarrow \text{irs}

\vdash i \ t \to t'

beta: [ \langle c \in \text{cs}; c \vdash i \to t' \rangle \Rightarrow \text{irs} \vdash i \text{Pabs cs} \ \text{\$p} \ t \to t' \]

fun: \text{irs} \vdash i \ t \to t' \Rightarrow \text{irs} \vdash i \ t \text{\$p} \ u \to t' \text{\$p} \ u \ |
arg: $irs \vdash i \rightarrow u' \Rightarrow irs \vdash it \rightarrow p \vdash t \rightarrow pu' = \Rightarrow irs \vdash it$

global-interpretation irewrite: rewriting irewrite rs for rs by standard (auto intro: irewrite.intros simp: app-fterm-def)+

abbreviation irewrite-rt :: irule-set $\Rightarrow$ pterm $\Rightarrow$ pterm $\Rightarrow$ bool (-/ $\vdash_i$/ - $\rightarrow^*$ - [50,0,50] 50) where
irewrite-rt rs $\equiv$ (irewrite rs)**

lemma (in irules) irewrite-closed:
assumes rs $\vdash_i$ t $\rightarrow$ u closed t
shows closed u
using assms proof induction
case (step name rs pats rhs t t')
then obtain env where match (name $\$$ pats) t = Some env t' = subst rhs env
unfolding irewrite-step-def by auto
hence closed-env env
using step by (auto intro: closed.match)

show ?case
unfolding (t' = -)
apply (subst closed-except-def)
apply (subst subst-frees)
apply fact
apply (subst match-dom)
apply fact
apply (subst frees-list-comb)
apply simp
apply (subst closed-except-def[symmetric])
using inner step by blast

next
case (beta c cs t t')
then obtain pat rhs where c = (pat, rhs)
by (cases c) auto
with beta obtain env where match pat t = Some env t' = subst rhs env
by auto
moreover have closed t
using beta unfolding closed-except-def by simp
ultimately have closed-env env
using beta by (auto intro: closed.match)

show ?case
unfolding (t' = subst rhs env)
apply (subst closed-except-def)
apply (subst subst-frees)
apply fact
apply (subst match-dom)
apply fact
apply simp

111
apply (subst closed-except-def[symmetric])
using inner beta ∘c = ∘ by (auto simp: closed-except-simps)
qed (auto simp: closed-except-def)

corollary (in irules) irewrite-rt-closed:
  assumes rs ⊢[i] t →∗ u closed t
  shows closed u
using assms by induction (auto intro: irewrite-closed)

Correctness of translation

abbreviation irelated :: nterm ⇒ pterm ⇒ bool (≈i :: [0,50] 50) where
n ≈i p ≡ nterm-to-pterms m = p

global-interpretation irelated: term-struct-rel-strong irelated
by standard
  (auto simp: app-pterms app-nterm-def const-pterms const-nterm-def elim:
  nterm-to-pterms.elims)

lemma irelated-vars: t ≈i u ⇒ frees t = frees u
by auto

lemma irelated-no-abs:
  assumes t ≈i u
  shows no-abs t ⇏ no-abs u
using assms
apply (induction arbitrary: t)
  apply (auto elim!: nterm-to-pterms.elims)
  apply (fold const-nterm-def const-pterms-def free-nterm-def free-pterms-def app-nterm-def
app-nterm-def)
by auto

lemma irelated-subst:
  assumes t ≈i u irelated.P-env nenv penv
  shows subst t nenv ≈i subst u penv
using assms proof (induction arbitrary: nenv penv u rule: nterm-to-pterms.induct)
  case (1 s)
    then show ?case
    by (auto elim!: fmrel-space[where x = s])
next
  case 4
  from 4[2][symmetric] show ?case
    apply simp
    apply (rule 4)
    apply simp
    using 4(3)
    by (simp add: fmrel-space)
qed auto

112
lemma related-irewrite-step:
  assumes name, pats, nterm-to-pterm rhs ⊢_i u → u' t ≈_i u
  obtains t' where unsplit-rule (name, pats, rhs) ⊢ t → t' t' ≈_i u'

proof
  let ?rhs' = nterm-to-pterm rhs
  let ?x = name $$
pats \from\ assms
\obtain env where match ?x u = Some env u' = subst ?rhs' env
  unfolding irewrite-step-def by blast
then obtain nenv where match ?x t = Some nenv irelated.P-env nenv env
  using assms
  by (metis Option.is-none-def not-None-eq option.rel-distinct(1) option.sel rel-option-unfold
      irelated.match-rel)

show thesis
proof
  show unsplit-rule (name, pats, rhs) ⊢ t → subst rhs nenv
    using ⟨match ?x t = -⟩ by auto
  next
    show subst rhs nenv ≈_i u'
      unfolding ⟨u' = -⟩ using ⟨irelated.P-env nenv env⟩
      by (auto intro: irelated-subst)
  qed
qed

theorem (in nrules) compile-correct:
  assumes compile (consts-of rs) ⊢_i u → u' t ≈_i u closed t
  obtains t' where rs ⊢_t t → t' t' ≈_i u'
  using assms(1–3) proof (induction arbitrary: t thesis rule: irewrite.induct)
  case (step name irs pats rhs u u')
then obtain crs where irs = translate-crules crs (name, crs) |∈| consts-of rs
  unfolding compile-def by force
moreover with step obtain rhs' where rhs = nterm-to-pterm rhs' (pats, rhs')
  |∈| crs
  unfolding translate-crules-def by force
ultimately obtain rule where split-rule rule = (name, (pats, rhs')) rule |∈| rs
  unfolding consts-of-def by blast
hence nrule rule
  using all-rules by blast

obtain t' where unsplit-rule (name, pats, rhs') ⊢ t → t' t' ≈_i u'
  using ⟨name, pats, rhs ⊢_i u → u' t ≈_i u⟩ unfolding ⟨rhs = nterm-to-pterm rhs'⟩
  by (elim related-irewrite-step)

hence rule ⊢ t → t'
  using ⟨nrule rule⟩ (split-rule rule = (name, (pats, rhs'))
  by (metis unsplit-split)
show ?case
proof (rule step.prems)
  show \( \Gamma_n t \rightarrow t' \)
    apply (rule nrewrite.step)
    apply fact
    apply fact
    done
next
  show \( t' \approx_i u' \)
    by fact
qed
next
case (\( \beta c cs u u' \))
  then obtain \( \text{pat rhs where } c = (\text{pat}, \text{rhs}) (\text{pat}, \text{rhs}) \mid\in\mid cs \)
    by (cases c) auto
  obtain \( v w \) where \( t = v \$_n w v \approx_i Pabs cs w \approx_i u \)
    using \( \Gamma \approx_i Pabs cs \) by (auto elim: nterm-to-pterm.elims)
  obtain \( x \text{ nrhs irhs where } v = (\Lambda_n x. \text{nrhs}) cs = \{ (Free x, \text{irhs}) \} \text{ nrhs} \approx_i \text{irhs} \)
    using \( \Gamma \approx_i Pabs cs \) by (auto elim: nterm-to-pterm.elims)
  hence \( t = (\Lambda_n x. \text{nrhs}) \$_n w \approx_i \Lambda_p x. \text{irhs} \)
    unfolding \( t = v \$_n w \) using \( \Gamma \approx_i Pabs cs \) by auto
  have \( \text{pat} = \text{Free x rhs} = \text{irhs} \)
    using \( cs = \{ (\text{Free x, irhs}) \} \Gamma (\text{pat}, \text{rhs}) \mid\in\mid cs \) by auto
  hence \( \text{Free x, irhs} \Gamma u \rightarrow u' \)
    using \( \beta c = \rightarrow \) by simp
  hence \( u' = \text{subst irhs} (fmap-of-list [(x, u)]) \)
    by simp
next
  case (\( \text{fun v v' u} \))
  obtain \( w x \) where \( t = w \$_n x w \approx_i v x \approx_i u \)
    using \( \Gamma \approx_i v \$_p w \) by (auto elim: nterm-to-pterm.elims)
with fun obtain $w'$ where $rs \vdash_n w \rightarrow w' \approx_i v'$

unfolding closed Except-def by auto

define

show ?case
proof (rule fun.prems)
  show $rs \vdash_n t \rightarrow w' \approx_i x$
    unfolding (t = $w \approx_i x$)
    by (rule nrewrite.fun) fact
next
  show $w' \approx_i v' \approx_i u$
    by auto fact
qed

define

next

proof (rule arg.prems)
  show $rs \vdash_n n t \rightarrow t' \approx_i u$
    unfolding (t = $n t \approx_i u$)
    by (rule nrewrite.arg) fact
next
  show $w' \approx_i v' \approx_i p$
    by auto fact
qed

corollary (in nrules) compile-correct-rt:
assumes compile (consts-of $rs \vdash_n t \rightarrow u' \approx_i u$ closed $t$
obtains $t' \approx_i t'$
using assms proof (induction arbitrary: thesis $t$)
case (step $u' \approx_i u'$)

obtain $t' \approx_i t'$
  using step by blast

obtain $t'' \approx_i u'$
proof (rule compile-correct)
  show compile (consts-of $rs \vdash_n t \rightarrow u'' \approx_i u'$
    by fact
next
  show closed $t'$
    using (rs $\vdash_n t \rightarrow u'$ (closed $t$)
    by (rule nrewrite-rt-closed)
qed blast

115
show \(?\)case
  proof (rule step.prems)
    show \(\text{rs} \vdash \text{nt} \rightarrow^{*} \text{t''}\)
      using \((\text{rs} \vdash \text{nt} \rightarrow^{*} \text{t'}) (\text{rs} \vdash \text{nt}' \rightarrow^{*} \text{t''})\) by auto
  qed fact
  qed blast

Completeness of translation

lemma (in nrules) compile-complete:
  assumes \(\text{rs} \vdash \text{nt} \rightarrow \text{t}'\) closed t
  shows compile \((\text{consts-of \text{rs}}) \vdash \text{nterm-to-pterm t \rightarrow nterm-to-pterm t}'\)
  using assms proof induction
  case \((\text{step r t t'})\)
  then obtain \(\text{pat} \text{ rhs}'\) where \(\text{r} = (\text{pat}, \text{rhs}')\)
    by force
  then have \((\text{pat}, \text{rhs}') \in\} \text{rs (pat, rhs')} \vdash \text{t} \rightarrow \text{t}'\)
    using \(\text{step}\) by blast+
  then have nrule \((\text{pat}, \text{rhs}')\)
    using all-rules by blast
  then obtain \(\text{name} \text{ pats where} (\text{name}, (\text{pats}, \text{rhs}')) = \text{split-rule r pat} = \text{name}\)
    unfolding \(\text{split-rule-def}\ (r = -)\)
    apply atomize-elim
    by (auto simp: split-beta)
  obtain \(\text{crs where} (\text{name}, \text{crs}) \in\} \text{consts-of \text{rs (pats, rhs')} \in\} \text{crs}\)
    using \((\text{step r t t'})\)
    by (metis consts-of-def fgroup-by-complete fst-conv snd-conv)
  then obtain \(\text{irs where} \text{irs} = \text{translate-crules crs}\)
    by blast
  then have \((\text{name}, \text{irs}) \in\} \text{compile (consts-of \text{rs})}\)
    unfolding \(\text{compile-def}\)
    using \((\text{name}, -) \in\} -\)
    by (metis \text{fimageI id-def map-prod-simp})
  obtain \(\text{rhs where} \text{rhs} = \text{nterm-to-pterm rhs'} (\text{pats}, \text{rhs}) \in\} \text{irs}\)
    using \((\text{irs} = -) (- \in\} \text{crs})\)
    unfolding \(\text{translate-crules-def}\)
    by (metis \text{fimageI id-def map-prod-simp})

from \text{step obtain env' where match pat t = Some env' t' = subst rhs' env'}
  unfolding \(r = -) using rewrite-step.simps
  by force
  then obtain \text{env where match pat (nterm-to-pterm t) = Some env irelated.P-env env env'}
    by (metis \text{irelated.match-rel option-rel-SomeI})
  then have subst \(\text{rhs env = nterm-to-pterm t'}\)
    unfolding \(t' = -)\)
    apply -
apply (rule sym)
apply (rule irelated-subst)
unfolding (rhs = ~)
  by auto

have name, pats, rhs ⊢ nterm-to-pterms t → nterm-to-pterm t'
  apply (rule irewrite-step1)
  using (match - - = Some env) unfolding (pat = ~)
  apply assumption
  by fact

show ?case
  by rule fact+

next
case (beta x t t')
obtain c where c = (Free x, nterm-to-pterm t)
  by blast
from beta have closed (nterm-to-pterm t')
  using closed-nterm-to-pterm[where t = t']
  unfolding closed-except-def
  by auto
show ?case
  apply simp
  apply rule
  using (c = ~)
  by (fastforce intro: irelated-subst[THEN sym])+

next
case (fun t t' u)
show ?case
  apply simp
  apply rule
  apply (rule fun)
  using fun
  unfolding closed-except-def
  apply simp
  done

next
case (arg u u' t)
show ?case
  apply simp
  apply rule
  apply (rule arg)
  using arg
  unfolding closed-except-def
  by simp

qed
Correctness of transformation

abbreviation irules-deferred-matches :: pterm list ⇒ irules ⇒ (term × pterm) fset where
irules-deferred-matches args ≡ fselect
  (λ(pats, rhs). map-option (λenv. (last pats, subst rhs env)) (matches (butlast pats) args))

category irules begin

inductive prelated :: pterm ⇒ pterm ⇒ bool (- ≈p - [0,50] 50) where
const: Pconst x ≈p Pconst x |
var: Pvar x ≈p Pvar x |
app: t1 ≈p u1 ⇒ t2 ≈p u2 ⇒ t1 $p t2 ≈p $p u1 $p u2 |
pat: rel-fset (rel-prod (=) prelated) cs1 cs2 ⇒ Pabs cs1 ≈p Pabs cs2 |
deref: (name, rsi) |∈| rs ⇒ 0 < arity rsi ⇒
  rel-fset (rel-prod (=) prelated) (irules-deferred-matches args rsi) cs ⇒
  list-all closed args ⇒
  name $S$ args ≈p Pabs cs

inductive-cases prelated-absE[consumes 1, case-names pat defer]: t ≈p Pabs cs

lemma prelated-refl[intro!]: t ≈p t
proof (induction t)
case Pabs
  thus ?case
    by (auto simp: snds.simps fnmember.rep-eq intro!: prelated.pat rel-fset-refl-strong rel-prodintros)
qed (auto intro: prelated.intros)

sublocale prelated: term-struct-rel prelated
by standard (auto simp: const-pterm-def app-ptermdf intro: prelated.intros elim: prelated.cases)

lemma prelated-pvars:
  assumes t ≈p u
  shows frees t = frees u
using assms proof (induction rule: prelated.induct)
case (pat cs1 cs2)
  show ?case
    apply simp
    apply (rule arg-cong[where f = ffUnion])
    apply (rule rel-fset-image-eq)
    apply fact
    apply auto
    done
next
case (defer name rsi args cs)
\{
  fix \texttt{pat t}
  \begin{align*}
  & \text{assume} (\texttt{pat, t}) \in \texttt{cs} \\
  & \text{with defer obtain} \ t' \\
  & \text{where} (\texttt{pat, t'}) \in \texttt{irules-deferred-matches args rsi frees t = frees t'} \\
  & \text{by (auto elim: rel-fsetE2)}
  \end{align*}
\}

\begin{align*}
  \text{then obtain} \ & \texttt{pats rhs env} \\
  & \text{where} \ (\texttt{pat, t')} \in \texttt{irules-deferred-matches args rsi}
  \begin{align*}
  & \text{frees t} = \text{frees t'} \\
  & \text{by (auto elim: rel-fsetE2)}
  \end{align*}
\end{align*}

\begin{align*}
  \text{then obtain} \ & \texttt{pats rhs env} \\
  & \text{where} \ (\texttt{pat, t')} \in \texttt{irules-deferred-matches args rsi}
  \begin{align*}
  & \text{frees t} = \text{frees t'} \\
  & \text{by (auto elim: rel-fsetE2)}
  \end{align*}
\end{align*}

\begin{align*}
  \begin{align*}
  & \text{have} \ \texttt{closed-except rhs (fress pats) linears pats} \\
  & \ \text{using} \ ((\texttt{pats, rhs}) \in \texttt{rsi}) \ (\texttt{name, rsi}) \in \texttt{rsi} \ \texttt{inner by blast+}
  \end{align*}
\end{align*}

\begin{align*}
  \text{have} \ & \texttt{arity-compatibles rsi} \\
  & \text{using defer inner by (blast dest: \texttt{fpairwiseD})}
\end{align*}

\begin{align*}
  \text{have} \ & \texttt{length pats > 0} \\
  & \text{by (subst arity-compatible-length) fact+}
\end{align*}

\begin{align*}
  \text{hence} \ & \texttt{pats = butlast pats @ [last pats]} \\
  & \text{by simp}
\end{align*}

\begin{align*}
  \text{note} \ & \texttt{fress t = frees t'} \\
  \text{also have} \ & \texttt{fress t'} = \texttt{fress rhs - fmdom env} \\
  & \text{unfolding} \ (t' = \cdot) \\
  & \text{apply (rule subst-frees)} \\
  & \text{apply (rule closed.matches)} \\
  & \text{apply fact+}
\end{align*}

\begin{align*}
  \text{done}
\end{align*}

\begin{align*}
  \text{also have} \ & \texttt{... = frees rhs - freess (butlast pats)} \\
  & \text{using (matches - - = \cdot) by (metis matches-dom)}
\end{align*}

\begin{align*}
  \text{also have} \ & \texttt{... \subseteq freess pats - freess (butlast pats)} \\
  & \text{using (closed-except - -)} \\
  & \text{by (auto simp: closed-except-def)}
\end{align*}

\begin{align*}
  \text{also have} \ & \texttt{... = frees (last pats) \subseteq freess (butlast pats)} \\
  & \text{by (subst (pats = \cdot)) (simp add: funion-fminus)}
\end{align*}

\begin{align*}
  \text{also have} \ & \texttt{... = frees (last pats)} \\
  & \text{proof (rule fminus-triv)} \\
  & \text{have fdisjnt (freess (butlast pats)) (freess [last pats])} \\
  & \text{using (linears pats) (pats = \cdot)} \\
  & \text{by (metis linears-appendD)}
\end{align*}

\begin{align*}
  \text{thus} \ & \texttt{frees (last pats) \cap freess (butlast pats) = \{\cdot\}} \\
  & \text{by (fastforce simp: fdisjnt-alt-def)}
\end{align*}

\text{qed}

\text{also have} \ & \texttt{... = frees pat unfolding (pat = \cdot) ..}

\begin{align*}
  \text{finally have} \ & \texttt{fres t} \subseteq \texttt{fress pat}.
\end{align*}

\text{hence} \ & \texttt{closed (Pabs cs)}

\text{unfolding} \ & \texttt{closed-except-simps}

\text{by (auto simp: closed-except-def)}
moreover have closed (name $\$ args)
unfolding closed-list-comb by fact
ultimately show $\langle ?case
unfolding closed-except-def by simp
qed auto

corollary prelated-closed: t $\approx_p u \implies$ closed $t \iff$ closed $u$
unfolding closed-except-def by (auto simp: prelated-pvars)

lemma prelated-no-abs-right:
  assumes t $\approx_p u$ no-abs $u$
  shows t = u
using assms
apply (induction rule: prelated.induct)
  apply auto
apply (fold app-pterm-def)
apply auto
done

corollary env-prelated-refl[intro!]: prelated.P-env env env
by (auto intro: fmap.rel-refl)

The following, more general statement does not hold: $t \approx_p u \implies$ rel-option prelated.P-env (match x t) (match x u) If t and u are related because of the prelated.defer rule, they have completely different shapes. Establishing is-abs t = is-abs u as a precondition would rule out this case, but at the same time be too restrictive.

Instead, we use $\llbracket \text{match } ?x \ ?u = \text{Some } ?env; ?t \approx_p ?u; \land env'. \llbracket \text{match } ?x \ ?t = \text{Some } env'; \text{prelated.P-env'} env' \rrbracket \rrbracket \implies \llbracket \text{thesis} \rrbracket \rrbracket \implies \llbracket \text{thesis} \rrbracket$.

lemma prelated-subst:
  assumes t_1 $\approx_p$ t_2 prelated.P-env env_1 env_2
  shows subst t_1 env_1 $\approx_p$ subst t_2 env_2
using assms proof (induction arbitrary: env_1 env_2 rule: prelated.induct)
  case (var x)
  thus $\langle ?case
  proof (cases rule: fmrel-cases[where x = x])
    case none
    thus $\langle ?thesis
      by (auto intro: prelated.var)
  next
    case (some t u)
    thus $\langle ?thesis
      by simp
  qed
next
  case (pat cs_1 cs_2)
let $\langle ?drop = \lambda env. \lambda (pat::term). \text{fmdrop-fset } \text{frees } pat \rangle env

120
\begin{verbatim}
from pat have prelated \text{P-env} (?drop env\_1 pat) (?drop env\_2 pat) for pat
   by blast
with pat show ?case
   by (auto intro!: prelated.pat rel-fset-image)
next
case (defer name rsi args cs)
have name $$\app\x args \equiv_p \text{Pabs} \app\_\text{cs}$$
   apply (rule prelated.defer)
   apply fact+
   apply (rule fset.rel-mono-strong)
   apply fact
   apply force
   apply fact
done
moreover have closed (name $$\app\x args)\$
   unfolding closed-list-comb by fact
ultimately have closed (Pabs cs)
   by (metis prelated-closed)

let ?drop = \lambda env. \lambda pat. fmdrop-fset (frees pat) env
let ?f = \lambda env. (\lambda (pat, rhs). (pat, subst rhs (?drop env pat)))

have name $$\app\x args \approx_p \text{Pabs} (\app\x ?f env\_2 |' cs)
   proof (rule prelated.defer)
   show (name, rsi) |\in| rs \theta < \text{arity rsi list-all closed args}
      using defer by auto
next
    {
      fix pat\_1 rhs\_1
      fix pat\_2 rhs\_2
      assume (pat\_2, rhs\_2) |\in| cs
      assume pat\_1 = pat\_2 rhs\_1 \equiv_p rhs\_2
      have rhs\_1 \equiv_p subst rhs\_2 (fmdrop-fset (frees pat\_2) env\_2)
         by (subst subst-closed-pabs) fact+
    }
    hence rel-fset (rel-prod (=) prelated) (id |' |) irules-deferred-matches args rsi
       (?f env\_2 |' | cs)
       by (force intro!: rel-fset-image[OF rel-fset - - -])
   thus rel-fset (rel-prod (=) prelated) (irules-deferred-matches args rsi) (?f env\_2 |' | cs)
      by simp
   qed
moreover have map (\lambda t. subst t env\_1) args = args
   apply (rule map-idI)
   apply (rule subst-closed-id)
   using defer by (simp add: list-all-iff)
\end{verbatim}
ultimately show ?case
  by (simp add: subst-list-comb)
qed (auto intro: prelated.intros)

lemma prelated-step:
  assumes name, pats, rhs |- i u → u' t ≈ₚ u
  obtains t' where name, pats, rhs |- t → t' t' ≈ₚ u'
proof –
  let ?lhs = name $ pats
  from assms obtain env where match ?lhs u = Some env u' = subst rhs env
    unfolding irewrite-step-def by blast
  then obtain env' where match ?lhs t = Some env' prelated.P-env env' env
    using assms by (auto elim: prelated.related-match)
  hence subst rhs env' ≈ₚ subst rhs env
    using assms by (auto intro: prelated-subst)
  show thesis
  proof
    show name, pats, rhs |- t → subst rhs env'
      unfolding irewrite-step-def using (match ?lhs t = Some env')
      by simp
    next
    show subst rhs env' ≈ₚ u'
      unfolding ⟨u' = subst rhs env⟩
      by fact
  qed
qed

lemma prelated-beta: — same problem as prelated.related-match
  assumes (pat, rhs) 2 |- t₂ → u₂ rhs₁ 2 ≈ₚ t₁ 2
  obtains u₁ where (pat, rhs₁) 1 |- t₁ → u₁ u₁ ≈ₚ u₂
proof –
  from assms obtain env₂ where match pat t₂ = Some env₂ u₂ = subst rhs₂ env₂
    by auto
  with assms obtain env₁ where match pat t₁ = Some env₁ prelated.P-env env₁ env₂
    by (auto elim: prelated.related-match)
  with assms have subst rhs₁ env₁ ≈ₚ subst rhs₂ env₂
    by (auto intro: prelated-subst)
  show thesis
  proof
    show (pat, rhs₁) |- t₁ → subst rhs₁ env₁
      using (match pat t₁ = -) by simp
    next
    show subst rhs₁ env₁ ≈ₚ u₂
      unfolding ⟨u₂ = -⟩ by fact
theorem transform-correct:
assumes transform-irule-set rs ⊢ i u −→ u′ t ≈ p u closed t
obtains t′ where rs ⊢ i t −→∗ t′ — zero or one step and t′ ≈ p u'
using assms(1-3) proof (induction arbitrary: t thesis rule: irewrite.induct)
case (beta c cs2 u2 x2')
obtain v u1 where t = v §p u1 v ≈ p u closed t
using (t ≈ p Pabs cs2 u1 ≈ p u2) by cases
with beta have closed u1
by (simp add: closed-except-def)

obtain pat rhs2 where c = (pat, rhs2) by (cases c) auto

from (v ≈ p Pabs cs2) show ?case
proof (cases rule: prelated-absE)
case (pat cs1)
with beta (c = -) obtain rhs1 where (pat, rhs1) |∈| cs1 rhs1 ≈ p rhs2
by (auto elim: rel-fsetE2)
with beta obtain x1' where (pat, rhs1) ⊢ u1 −→ x1' x1' ≈ p x2'
using (u1 ≈ p u2) assms (c = -)
by (auto elim: prelated-beta simp del: rewrite-step.simps)

show ?thesis
proof (rule beta.prems)
show rs ⊢ t −→∗ x1'
unfolding (t = -) (v = -)
by (intro r-into-rtranclp irewrite.beta) fact+
next
show x1' ≈ p x2'
by fact
qed

next
case (defer name rsi args)
with beta (c = -) obtain rhs1' where (pat, rhs1') |∈| irules-deferred-matches
args rsi rhs1' ≈ p rhs2
by (auto elim: rel-fsetE2)
then obtain enva rhs1 pats
where matches (butlast pats) args = Some enva pat = last pats rhs1' = subst
rhs1 enva
and (pats, rhs1) |∈| rsi
by auto
hence linears pats
using (name, rsi) |∈| rsi inner unfolding irules-def by blast

have arity-compatibles rsi
using defer inner by (blast dest: fpairwiseD)
have length pats > 0
by (subst arity-compatible-length) fact+

hence pats = butlast pats @ [pat]
unfolding \( \text{pat} = \& \) by simp

from beta \( c = \& \) obtain \( \text{env}_b \) where match pat \( u_2 = \text{Some env}_b \) \( x_2' = \text{subst} \)
\( \text{rhs}_2 \) \( \text{env}_b \)

by fastforce

with \( u_1 \approx_p u_2 \) obtain \( \text{env}_b' \) where match pat \( u_1 = \text{Some env}_b' \) \text{prelated.P-env} \( \text{env}_b' \) \( \text{env}_b \)

by (metis \text{prelated.related-match})

have \( \text{closed-env} \) \( \text{env}_a \)
by (rule \text{closed.matches}) fact+

have \( \text{closed-env} \) \( \text{env}_b' \)

apply (rule \text{closed.matches}[where pats = [pat] and ts = [u_1]])

apply simp
apply fact
apply simp
apply fact
apply fact

done

have \( \text{fmdom} \) \( \text{env}_a \) = freess (butlast pats)
by (rule \text{matches-dom}) fact

moreover have \( \text{fmdom} \) \( \text{env}_b' \) = freess pat
by (rule \text{matchs-dom}) fact

moreover have \( \text{fdisjnt} \) (freess (butlast pats)) (freess pat)
using \( \langle \text{pats} = \& \rangle \) \text{linears pats}
by (metis \text{freess-single linears-appendD(3)})

ultimately have \( \text{fdisjnt} \) (\( \text{fmdom} \) \( \text{env}_a \)) (\( \text{fmdom} \) \( \text{env}_b' \))
by simp

show \( \text{thesis} \)

proof (rule beta.prems)

have \( r s \vdash_i \text{name} \) \text{$$ args \$ p u_1 \rightarrow \text{subst} \text{rhs}_1' \text{env}_b' \)

proof (rule \text{irrewrite.step})

show \( \langle \text{name}, \text{rsi} \rangle \in r s (\text{pats}, \text{rhs}_1) \in r s \)
by fact+

next

show \( \text{name}, \text{pats}, \text{rhs}_1 \vdash_i \text{name} \) \text{$$ args \$ p u_1 \rightarrow \text{subst} \text{rhs}_1' \text{env}_b' \)

apply (rule \text{irrewrite-stepI})

apply (fold \text{app-termd-def})
apply (\text{subst list-comb-snoc})
apply (\text{subst \text{matches-match-list-comb}})
apply (\text{subst \&pats = \&})
apply (rule \text{matchs-appI})

apply fact
apply simp
apply fact
unfolding \( \langle \text{rhs}_1' = \& \rangle \)
apply (rule subst-indep')
apply fact+
done
qed
thus rs ⪰ t →∗ subst rhs₁ envb'
unfolding ⟨t = -⟩ ⟨v = -⟩
by (rule r-into-rtranclp)
next
show subst rhs₁ envb' ≈ x₂'
unfolding ⟨x₂' = -⟩
by (rule prelated-subst) fact+
qed
next
case (step name rs₂ pats rhs u u')
then obtain rs₁ where rs₂ = transform-irules rs₁ (name, rs₁) |∈| rs
unfolding transform-irule-set-def by force
hence arity-compatibles rs₁
using inner by (blast dest: fpairwiseD)

show ?case
proof (cases arity rs₁ = 0)
  case True
  hence rs₂ = rs₁
  unfolding ⟨rs₂ = -⟩ transform-irules-def by simp
  with step have (pats, rhs) |∈| rs₁
  by simp
  from step obtain t' where name, pats, rhs ⪰ t → t' t' ≈_p u'
  using assms
  by (auto elim: prelated-step)

show ?thesis
proof (rule step.prems)
  show rs ⪰ t →∗ t'
  by (intro conjI exI r-into-rtranclp irewrite.step) fact+
qed fact
next
let ?f = λ(pats, rhs). (butlast pats, last pats, rhs)
let ?grp = fgroup-by ?f rs₁

case False
hence rs₂ = map-prod id Pabs |'[| ?grp
unfolding ⟨rs₂ = -⟩ transform-irules-def by simp
with step obtain cs where rhs = Pabs cs (pats, cs) |∈| ?grp
by force

from step obtain env₂ where match (name $$ pats) u = Some env₂ u' = subst rhs env₂
unfolding irewrite-step-def by auto
then obtain \( \text{args}_2 \) where \( u = \text{name} \) \( \$\) \( \text{args}_2 \) matches \( \text{pats} \) \( \text{args}_2 = \text{Some} \) \( \text{env}_2 \)
by (\text{auto elim: match-list-combE})
with \( \text{step} \) obtain \( \text{args}_1 \) where \( t = \text{name} \) \( \$\) \( \text{args}_1 \) list-all2 \( \text{prelated}\) \( \text{args}_1 \)
\( \text{args}_2 \)
by (\text{auto elim: prelated.list-combE})

then obtain \( \text{env}_1 \) where \( \text{matches}\) \( \text{pats} \) \( \text{args}_1 = \text{Some} \) \( \text{env}_1 \) \( \text{prelated}.P\text{-env} \)
\( \text{env}_1 \) \( \text{env}_2 \)
using \( \langle\text{matches}\) \( \text{pats}\) \( \text{args}_2 = \text{-}\rangle \) by (\text{metis prelated.related-matches})
hence \( \text{fndom}\) \( \text{env}_1 = \text{free}\) \( \text{pats} \)
by (\text{auto simp: matches-dom})

obtain \( \text{cs}' \) where \( u' = \text{Pabs}\) \( \text{cs}' \)
unfolding \( \langle u' = \text{-}\rangle \langle\text{rhs}\) = \text{-}\rangle \) by auto
hence \( \text{cs}' = (\lambda\langle\text{pat}, \text{rhs}\. \langle\text{pat}, \text{subst}\) \( \text{rhs}\) \( \text{free}\) \( \text{pats}\) \text{env}_2 \rangle)\) \text{[\text{\mid}]}
\( \text{cs} \)
using \( \langle u' = \text{subst}\) \( \text{rhs}\) \( \text{env}_2 \rangle \) unfolding \( \langle\text{rhs}\) = \text{-}\rangle \)
by \text{simp}

show \( \text{thesis} \)
proof (\text{rule step.prems})
show \( rs \vdash t \rightarrow^* t \)
by (\text{rule rtranclp.rtrancl-refl})
next
show \( t \sim_p u' \)
unfolding \( \langle u' = \text{Pabs}\) \( \text{cs}'\rangle \langle t = \text{-}\rangle \)
proof (\text{intro prelated.defer rel-fsetI; safe?})
show \( \langle\text{name}, rs_1 \rangle\) \text{[-]} \( rs \)
by \text{fact}
next
show \( 0 < \text{arity}\) \( rs_1 \)
using \text{False} by simp
next
show \( \text{list-all closed}\) \( \text{args}_1 \)
using \( \langle\text{closed}\) \( t\rangle \)\text{ unfolding} \( \langle t = \text{-}\rangle \)\text{ closed-list-comb} \text{.}
next
fix \( \text{pat}\) \( \text{rhs}' \)
assume \( \langle\text{pat}, \text{rhs}'\rangle\) \text{[-]} \( \text{irules-deferred-matches}\) \( \text{args}_1 rs_1 \)
then obtain \( \text{pats}'\) \( \text{rhs}\) \( \text{env} \)
where \( \langle\text{pats}', \text{rhs}\rangle\) \text{[-]} \( rs_1 \)
and \( \text{matches}\) \( \langle\text{butlast}\) \( \text{pats}'\rangle\) \( \text{args}_1 = \text{Some}\) \( \text{env}\) \( \text{pat} = \text{last}\) \( \text{pats}'\) \( \text{rhs}' \)
= \( \text{subst}\) \( \text{rhs}\) \( \text{env} \)
by auto
with \text{False} have \( \text{pats}' \neq [\text{\n}] \)
using \( \langle\text{arity-compatibles}\) \( rs_1\rangle \)
by (\text{metis list.size(3) arity-compatible-length})
hence \( \text{butlast}\) \( \text{pats}' \otimes [\text{last}\) \( \text{pats}'] = \text{pats}' \)

126
by simp

from \langle \text{pats, cs} \mid \in \text{?grp} \rangle \text{ obtain pats}_e \text{ rhs}_e
where \langle \text{pats}_e, \text{ rhs}_e \rangle \mid \in \text{rs}_1 \text{ \(=\)} \text{butlast pats}_e
by (auto elim: fgroup-byE2)

have \text{patterns-compatible} \text{ (butlast pats') pats}
unfolding \langle \text{pats} = \cdot \rangle
apply (rule rev-accum-rel-butlast)
using \langle \text{pats}', \text{ rhs} \rangle \mid \in \text{rs}_1 \langle \text{pats}_e, \text{ rhs}_e \rangle \mid \in \text{rs}_1 \langle \text{name, rs}_1 \rangle \mid \in \text{rs}

inner
by (blast dest: fpairwiseD)

interpret \text{irules'}: \text{irules C-info transform-irule-set rs} by (rule rules-transform)

have butlast pats' = pats \text{ env} = \text{env}_1
apply (rule \text{matchs-compatible-\text{eq}})

subgoal by fact

subgoal
apply (rule \text{linears-butlastI})
using \langle \text{pats}', \text{ rhs} \rangle \mid \in \text{rs}_1 \langle \text{name, rs}_1 \rangle \mid \in \text{rs} \text{ inner} by blast

subgoal
using \langle \text{pats}, \cdot \rangle \mid \in \text{rs}_2 \langle \text{name, rs}_2 \rangle \mid \in \text{transform-irule-set rs} \text{ using \text{irules'}.inner by blast}
apply fact+

subgoal
apply (rule \text{matchs-compatible-\text{eq}})
apply fact
apply (rule \text{linears-butlastI})
using \langle \text{pats}', \text{ rhs} \rangle \mid \in \text{rs}_1 \langle \text{name, rs}_1 \rangle \mid \in \text{rs} \text{ inner}
apply blast
using \langle \text{pats}, \cdot \rangle \mid \in \text{rs}_2 \langle \text{name, rs}_2 \rangle \mid \in \text{transform-irule-set rs} \text{ using \text{irules'}.inner apply blast}
by fact+

done

let ?rhs-subst = \lambda \text{env}. \text{subst rhs} \text{ (fmdrop-fset \{frees pat\} \text{ env})}

have \text{fdom env}_2 = \text{freess pats}
using \langle \text{match} (- \$\$ \cdot) \cdot = \text{Some env}_2 \rangle
by (simp add: \text{match-dom})

show \text{fBex cs'} (\text{rel-prod} (=) \text{prelated} \langle \text{pat, rhs'} \rangle)
unfolding \langle \text{rhs'} = \cdot \rangle
proof (rule \text{fBexI, rule rel-prod.intros})
have \text{fdisjnt} \text{ (frees (butlast pats'))} \text{ (frees (last pats'))}
apply (\text{subst frees-single[symmetric]})
apply (rule \text{linears-appendD})
apply \((\text{subst } \langle \text{butlast } \text{pats'} @ \text{last } \text{pats'} \rangle = \text{pats'})\)
using \((\text{pats'}, \text{rhs}) |\in| \text{rs}_1\) \(\langle \text{name, } \text{rs}_1 \rangle |\in| \text{rs} \rangle\) inner by blast

show \(\text{subst } \text{rhs } \text{env} \simeq_p \text{?rhs-subst } \text{env}_2\)
apply \((\text{rule prelated-subst})\)
apply \((\text{rule prelated-refl})\)
unfolding \text{fmfilter-alt-defs}
apply \((\text{subst } \text{fmfilter-true})\)

subgoal premises \text{prems} for \(x \ y\)
using \(\text{fmdomI}[\text{OF prems}]\)
unfolding \(\langle \text{pat} = \_ \rangle \text{fmdom } \text{env}_2 = \_\)
apply \((\text{subt } \text{asm} )\langle \text{butlast } \text{pats'} = \text{pats} |\text{symmetric} \rangle\)
using \(\text{fdisjnt } (\text{frees } (\text{butlast } \text{pats'}) ) (\text{frees } (\text{last } \text{pats'}) )\)
by \((\text{auto simp: fdisjnt-alt-def})\)

subgoal
unfolding \(\langle \text{env} = \_ \rangle\)
by fact
done

next
have \((\text{pat, } \text{rhs}) |\in| \text{cs} \)
unfolding \(\langle \text{pat} = \_ \rangle\)
apply \((\text{rule fgroup-byD}[\text{where } a = (x, y) \text{ for } x \ y])\)
apply fact
apply simp
apply \((\text{intro conjI})\)
apply fact
apply \((\text{rule refl}+)\)
apply fact
done
thus \((\text{pat, } \text{?rhs-subst } \text{env}_2) |\in| \text{cs'} \)
unfolding \(\langle \text{cs'} = \_ \rangle\) by force
qed simp

next
fix \(\text{pat } \text{rhs'}\)
assume \((\text{pat, } \text{rhs'}) |\in| \text{cs'} \)
then obtain \(\text{rhs} \)
where \((\text{pat, } \text{rhs}) |\in| \text{cs} \)
and \(\text{rhs'} = \text{subst } \text{rhs } (\text{fndrop-fset } (\text{frees pat}) \text{ env}_2 )\)
unfolding \(\langle \text{cs'} = \_ \rangle\) by auto
with \((\text{pats, } \text{cs}) |\in| \text{?grp} \) obtain \(\text{pats}'\)
where \((\text{pats', } \text{rhs}) |\in| \text{rs}_1 \text{ pats} = \text{butlast pats'} \text{ pat} = \text{last pats'}\)
by auto
with \(\text{False}\) have \(\text{length pats'} \neq 0\)
using \(\text{arity-compatibles } \_\) by \((\text{metis arity-compatible-length})\)
hence \(\text{pats'} = \text{pats } @ [\text{pat}]\)
unfolding \(\langle \text{pats} = \_ \rangle \langle \text{pat} = \_ \rangle\) by auto
moreover have \(\text{linears pats'}\)
using \((\text{pats', } \text{rhs}) |\in| \text{rs}_1 \) \(\langle \text{name, } \text{rs}_1 \rangle |\in| \_ \rangle\) inner by blast
ultimately have \( \text{fdisjnt} (\text{fmdom} \ env_1) \) (frees \( \text{pat} \))

unfolding \( \text{fmdom} \ env_1 = \) by (auto dest: linears-appendD)

let \(?rhs\text{-subst} = \lambda env. \text{subst} rhs (\text{fmdrop-fset} (\text{frees} \text{pat}) env)\)

show \( \text{fBex} (\text{irules-deferred-matches} \text{args}_1 \text{rs}_1) \) (\( \lambda e \). rel-prod (=) \( \text{prelated} e \))

unfolding \( \langle \text{rhs'} = - \rangle \)

proof (rule \( \text{fBexI} \), rule rel-prod.intros)

show \(?rhs\text{-subst} env_1 \approx ?rhs\text{-subst} env_2\)

using (\( \text{prelated}.P\text{-env} env_1 env_2 \)) inner

by (auto intro: prelated-subst)

next

have matches (butlast \text{pats'}) \text{args}_1 = \text{Some} env_1

using (matches \text{pats} \text{args}_1 = \) \( \langle \text{pats} = - \rangle \) by simp

moreover have \( \text{subst} rhs \text{env}_1 = ?rhs\text{-subst} env_1 \)

apply (rule \( \text{arg}-\text{cong}[\text{where} f = \text{subst} rhs] \))

unfolding \( \text{fmfilter-alt-defs} \)

apply (rule \( \text{fmfilter-true}[\text{symmetric}] \))

using \( \text{fdisjnt} (\text{fmdom} env_1) \)

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

ultimately show \( (\text{pat}, ?rhs\text{-subst} env_1) \mid\in\mid \text{irules-deferred-matches} \text{args}_1 \text{rs}_1 \) (\( \text{pat} = \text{last} \text{pats'} \))

show \text{rs}_1 \triangleright x \approx \text{p}_v \approx \text{p}_u \) closed \text{w}

using (\( \langle \text{pats'}, \text{rhs} \rangle \mid\in\mid \text{rs}_1 \) \( \langle \text{pat} = \text{last} \text{pats'} \rangle \))

by auto

qed simp

qed

qed

next
case (\( \text{fun} \ v \ v' \ u \))

obtain \text{w} \ x \ where \( t = \text{w} \approx_p \text{v} \approx_p \text{u} \) closed \text{w}

using (\( t \approx_p \text{v} \approx_p \text{w} \)) by cases (auto simp: closed-except-def)

with \( \text{fun} \) obtain \text{w'} \ where \( \text{rs} \vdash_i \text{w} \rightarrow^* \text{w'} \approx_p \text{v}' \)

by blast

show ?case

proof (rule \( \text{fun}.\text{prems} \))

show \( \text{rs} \vdash_i t \rightarrow^* \text{w'} \approx_p \text{x} \)

unfolding (\( t \rightarrow \))

by (intro irewrite.rt-comb[unfolded \( \text{app-pterms-def} \)] rtranclp.rtrancl-refl) fact

next

show w' \approx_p \text{v'} \approx_p \text{w} \approx_p \text{u}

by (rule prelated.app) fact

qed

next
case (\( \text{arg} \ u \ u' \v \))

obtain \text{w} \ x \ where \( t = \text{w} \approx_p \text{v} \approx_p \text{u} \) closed \text{x}

Next
using \( t \approx_p v, t \approx_p u \) \( (\text{closed} t) \) by cases (auto simp: closed-except-def)
with \( \text{arg} \) obtain \( x' \) where \( rs \vdash iT \rightarrow\ast x' x' \approx_p u' \)
by blast

show \( ?\text{case} \)
proof (rule \( \text{arg.prems} \))
show \( rs \vdash iT \rightarrow\ast w \$ p x' \)
unfolding \( (t = w \$ p x) \)
by (intro \text{irewrite}.rt-comb[unfolded \text{app-pterm-def}] \text{rtranclp}.rtrancl-refl) \text{fact}
next
show \( w \$ p x' \approx_p v \$ p u' \)
by (rule \text{prelated.app}) \text{fact+}
qed
qed
end

Completeness of transformation

lemma (in \text{irules}) \text{transform-completeness}:
assumes \( rs \vdash iT \rightarrow t' \) \( \text{closed} t \)
shows \( \text{transform-irule-set} \; rs \vdash iT \rightarrow\ast t' \)
using \( \text{assms} \) proof induction
\begin{itemize}
\item case \( \text{arity} \; irs' = 0 \)
\item hence \( irs = irs' \)
\item unfolding \( \text{transform-irules-def} \)
\item by (metis \text{fimageI} \text{id-apply} \text{map-prod-simp})
\end{itemize}
show \( ?\text{case} \)
proof (cases \text{arity} \; irs' = 0)
\begin{itemize}
\item case \( \text{True} \)
\item hence \( irs = irs' \)
\item unfolding \( (irs = \cdot) \)
\item unfolding \( \text{transform-irules-def} \)
\item by \text{simp}
\end{itemize}
with \( \text{step} \) have \( \{\text{pats}', \text{rhs}'\} \in\} \; \text{irs name, pats', rhs'} \vdash iT \rightarrow t' \)
by \text{blast+}
have \( \text{transform-irule-set} \; rs \vdash iT \rightarrow\ast t' \)
apply (rule \text{r-into-rtranclp})
apply rule
by \text{fact+}
show \( ?\text{thesis} \) by \text{fact}
next
let \( ?f = \lambda(\text{pats}, \text{rhs}).(\text{butlast} \; \text{pats}, \text{last} \; \text{pats}, \text{rhs}) \)
let \( ?\text{grp} = \text{fgroup-by} \; ?f \; \text{irs}' \)
note \text{closed-except-def} \| \text{simp add}
\begin{itemize}
\item case \( \text{False} \)
\item then have \( irs = \text{map-prod} \; \text{id} \; \text{Pats} \; |' \) \( ?\text{grp} \)
\end{itemize}
unfolding \( \text{irs} = \cdot \) 
by simp

with \( \text{False} \) have \( \text{irs} = \text{transform-irules} \ \text{irs}' \)
unfolding \( \text{transform-irules-def} \)
by simp

obtain \( \text{pat} \ \text{pats} \) where \( \text{pat} = \text{last} \ \text{pats} \ \text{pats} = \text{butlast} \ \text{pats}' \)
by blast

from \( \text{step} \ \text{False} \) have \( \text{length} \ \text{pats}' \neq 0 \)
using \( \text{arity-compatible-length} \ \text{inner} \)
by \((\text{smt} \ \text{fBallE} \ \text{prod}.\text{simps}(2))\)
then have \( \text{pats}' = \text{pats} @ [\text{pat}] \)
unfolding \( \text{pat} = \cdot \ (\text{pats} = \cdot) \)
by simp

from \( \text{step} \ \text{have} \ \text{linears} \ \text{pats}' \)
using \( \text{inner} \ \text{fBallE} \)
by \((\text{metis} \ (\text{mono-tags}, \ \text{lifting}) \ \text{old}.\text{prod}.\text{case})\)
then have \( \text{fdisjnt} \ (\text{frees} \ \text{pats}) \ (\text{frees} \ \text{pat}) \)
unfolding \( \text{pats}' = \cdot \)
using \( \text{linears-appendD} \ (3) \ \text{frees-single} \)
by force

from \( \text{step} \ \text{obtain} \ \text{cs} \ \text{where} \ (\text{pats}, \ \text{cs}) |\in| \ ?\text{grp} \)
unfolding \( \text{pats} = \cdot \)
by \((\text{metis} \ (\text{no-types}, \ \text{lifting}) \ \text{fgroup-by-complete} \ \text{fst-conv} \ \text{prod}.\text{simps}(2))\)
with \( \text{step} \ \text{have} \ \ (\text{pat}, \ \text{rhs}') |\in| \ \text{cs} \)
unfolding \( \text{pat} = \cdot \ (\text{pats} = \cdot) \)
by \((\text{meson} \ \text{fgroup-by-complete} \ \text{fst-conv} \ \text{prod}.\text{case})\)
have \( (\text{pats}, \ \text{Pabs} \ \text{cs}) |\in| \ \text{irs} \)
using \( (\text{irs} = \text{map-prod} \ \text{id} \ \text{Pabs} |' | \ ?\text{grp} ; (\text{pats}, \ \text{cs}) |\in| \ ?) \)
by \((\text{metis} \ (\text{no-types}, \ \text{lifting}) \ \text{eq-snd-iff} \ \text{fst-map-prod} \ \text{id-def} \ \text{rev-fimage-eqI} \ \text{snd-map-prod})\)
from \( \text{step} \ \text{obtain} \ \text{env}' \ \text{where} \ \text{match} \ (\text{name} \ $$ \text{pats}') \ t = \text{Some} \ \text{env}' \ \text{subst} \ \text{rhs}' \ \text{env}' = t' \)
using \( \text{irewrite-step-def} \ \text{by auto} \)
have \( \text{name} \ $$ \text{pats}' = (\text{name} \ $$ \text{pats}) \ $ \text{pat} \)
unfolding \( \text{pats}' = \cdot \)
by \((\text{simp add: app-term-def})\)
then obtain \( t_0 \ t_1 \ \text{env}_0 \ \text{env}_1 \ \text{where} \ t = t_0 \ \text{p} \ t_1 \ \text{match} \ (\text{name} \ $$ \text{pats}) \ t_0 = \text{Some} \ \text{env}_0 \ \text{match} \ t_1 = \text{Some} \ \text{env}_1 \ \text{env}' = \text{env}_0 \ + + f \ \text{env}_1 \)
using \( \text{match-appE-split} \ [\text{OF} \ \text{match} \ (\text{name} \ $$ \text{pats}) \ - = \ ?[\text{unfolded} \ (\text{name} \ $$ \text{pats}' = \cdot)], \ \text{unfolded} \ \text{app-pterm-def}] \)
by blast
with \( \text{step} \ \text{have} \ \text{closed} \ t_0 \ \text{closed} \ t_1 \)
by auto
then have \( \text{closed-env} \ \text{env}_0 \ \text{closed-env} \ \text{env}_1 \)
using \( \text{match-vars} \ [\text{OF} \ \text{match} - t_0 = \cdot] \ \text{match-vars} \ [\text{OF} \ \text{match} - t_1 = \cdot] \)
unfolding \( \text{closed-except-def} \)
by auto

131
obtain $t_0'$ where $\text{subst} (\text{Pabs} \ cs) \ env_0 = t_0'$ 
by blast
then obtain $cs'$ where $t_0' = \text{Pabs} \ cs' \ cs' = ((\lambda (\text{pat}, \text{rhs}). (\text{pat}, \text{subst} \ \text{rhs}$$ (\text{fmdrop-fset} (\text{frees} \ \text{pat}) \ env_0))) \ |^t| \ cs)$
using subst-pterms.simps(3) by blast
obtain $rhs$ where $\text{subst} \ \text{rhs}' \ (\text{fmdrop-fset} (\text{frees} \ \text{pat}) \ env_0) = rhs$
by blast
then have $(\text{pat}, \text{rhs}) |\in| cs'$ 
unfolding $(cs' = \cdot)$
using $(\cdot |\in| \ cs)$
by (metis (mono-tags, lifting) old.prod.case rev-fimage-eqI)
have $env_0 ++f \ env_1 = (\text{fmdrop-fset} (\text{frees} \ \text{pat}) \ env_0) ++f \ env_1$
apply $(\text{subst} \ \text{fnadd-drop-left-dom}[\text{symmetric}])$
using $(\text{match} \ \text{pat} - = \cdot \ \text{match-dom})$
by metis
have $\text{fdisjnt} (\text{fmdom} \ env_0) \ (\text{fmdom} \ env_1)$
using $\text{match-dom}$
using $(\text{match} \ \text{pat} - = \cdot \ : \text{match} (\text{name} \ $$ \text{pats}) - = \cdot)$
using $(\text{fdisjnot} - \cdot)$
unfolding $\text{fdisjnt-alt-def}$
by (metis $\text{matches-dom \ match-list-combE}$)
have $\text{subst} \ \text{rhs} \ env_1 = t'$
unfolding $(\cdot = \text{rhs})[\text{symmetric}]$
unfolding $(\cdot = t')[\text{symmetric}]$
unfolding $(env' = \cdot)$
unfolding $(env_0 ++f \cdot = \cdot)$
apply $(\text{subst} \ \text{subst-indep})$
using $(\text{closed-env} \ env_0)$
apply blast
using $(\text{fdisjnt} (\text{fmdom} \ - \cdot) \ - \cdot)$
unfolding $\text{fdisjnt-alt-def}$
by auto

show $\text{thesis}$
unfolding $(t = \cdot)$
apply rule
apply $(\text{rule} \ r\text{-into-rtranclp})$
apply $(\text{rule} \ \text{irewrite.intro}(3))$
apply rule
apply fact$	ext{+}$
apply $(\text{rule} \ \text{irewrite-stepI})$
apply fact$	ext{+}$
unfolding $(t_0' = \cdot)$
apply rule
apply fact
using $(\text{match} \ \text{pat} \ t_1 = \cdot \ : \text{subst} \ \text{rhs} - = \cdot)$
by force
qed

qed (auto intro: \text{irewrite.rt-comb[unfolded app-pterms-def]} intro!: \text{irewrite.intros}
simp: closed-except-def

Computability

export-code
  compile transform-irules
  checking Scala SML

end

3.3.2 Pure pattern matching rule sets

theory Rewriting-Pterm
imports Rewriting-Pterm-Elim
begin

type-synonym prule = name × pterm

primrec prule :: prule ⇒ bool where
prule (-, rhs) ←→ wellformed rhs ∧ closed rhs ∧ is-abs rhs

lemma pruleI[intro!]: wellformed rhs =⇒ closed rhs =⇒ is-abs rhs =⇒ prule
(name, rhs)
by simp

locale prules = constants C-info fst ⊢ | rs for C-info and rs :: prule fset +
  assumes all-rules: fBall rs prule
  assumes fmap: is-fmap rs
  assumes not-shadows: fBall rs (λ(-, rhs). ¬ shadows-consts rhs)
  assumes welldefined-rs: fBall rs (λ(-, rhs). welldefined rhs)

Rewriting

inductive prewrite :: prule fset ⇒ pterm ⇒ pterm ⇒ bool (-/ ⊢_p/ - →*/ -
[50,0,50] 50) for rs where
step: (name, rhs) ∈ rs ⊢_p Pconst name → rhs |
beta: c ∈| cs =⇒ c ⊢ t → t' =⇒ rs ⊢_p Pabs cs $p t → t' |
fun: rs ⊢_p t → t' =⇒ rs ⊢_p t $p u → t'$ $p u |
arg: rs ⊢_p u → u' =⇒ rs ⊢_p t $p u → t $p u'

global-interpretation prewrite: rewriting prewrite rs for rs
by standard (auto intro: prewrite.intros simp: app-pterm-def)+

abbreviation prewrite-rt :: prule fset ⇒ pterm ⇒ pterm ⇒ bool (-/ ⊢_p/ - →*/ -
[50,0,50] 50) where
prewrite-rt rs ≡ (prewrite rs)**

Translation from irule-set to prule fset

definition finished :: irule-set ⇒ bool where
\[ \text{finished } rs = fBall rs \ (\lambda (-, \ irs). \ \text{arity } irs = 0) \]

**definition** translate-rhs :: irules \Rightarrow pterm where 
\[ \text{translate-rhs} = \text{snd} \circ fthe-elem \]

**definition** compile :: irule-set \Rightarrow prule fset where 
\[ \text{compile} = \text{fimage} \ (\text{map-prod } \text{id} \ \text{translate-rhs}) \]

**lemma** compile-heads: fst \ | \ compose \ rs = \text{fst} \ | \ compose \ rs

**unfolding** compile-def by simp

**Correctness of translation**

**lemma** arity-zero-shape:
\[ \text{assumes arity-compatibles } rs \ \text{arity } rs = 0 \ \text{is-fmap } rs \ \text{rs} \neq \{||\}\]
\[ \text{obtains } t \text{ where } rs = \{|(\|, \ t)\|} \]
\[ \text{proof} – \]
\[ \text{from } \text{assms} \ \text{obtain } ppats \ \text{prhs where} \ (ppats, \ \text{prhs}) \ \in \ rs \]
\[ \text{by fast} \]

moreover \{ \]
\[ \text{fix } \text{pats} \ \text{rhs} \]
\[ \text{assume} \ (\text{pats, } \text{rhs}) \ \in \ rs \]
\[ \text{with } \text{assms} \ \text{have} \ \text{length pats} = 0 \]
\[ \text{by (metis arity-compatible-length)} \]
\[ \text{hence } \text{pats} = [] \]
\[ \text{by simp} \]
\}

**note** all = this

ultimately have proto: ([], \text{prhs}) \ \in \ rs \ \text{by auto}

have fBall rs (\lambda (\text{pats, rhs}). \ \text{pats} = [] \land \text{rhs} = \text{prhs})
\[ \text{proof safe} \]
\[ \text{fix } \text{pats} \ \text{rhs} \]
\[ \text{assume} \ \text{cur:} \ (\text{pats, rhs}) \ \in \ rs \]
\[ \text{with } \text{all} \ \text{show} \ \text{pats} = [] . \]
\[ \text{with } \text{cur} \ \text{have} \ ([], \ \text{rhs}) \ \in \ rs \ \text{by auto} \]

\[ \text{with proto show} \ \text{rhs} = \text{prhs} \]
\[ \text{using } \text{assms by (auto dest: is-fmapD)} \]
\[ \text{qed} \]

hence fBall rs (\lambda r. \ r = ([], \ \text{prhs}))
\[ \text{by blast} \]
\[ \text{with } \text{assms} \ \text{have} \ \text{rs} = \{|(\|, \ \text{prhs})\|} \]
\[ \text{by (simp add: singleton-fset-is)} \]

**thus** thesis 
\[ \text{by (rule that)} \]
\[ \text{qed} \]
lemma (in irules) compile-rules:
  assumes finished rs
  shows prules C-info (compile rs)

proof
  show is-fmap (compile rs)
    using fmap
    unfolding compile-def map-prod-def id-apply
    by (rule is-fmap-image)

next
  show fdisjnt (fst |'| compile rs) C
    unfolding compile-def
    using disjnt by simp

next
  have
    fBall (compile rs) prule
    fBall (compile rs) (λ(-, rhs). ¬ shadowsconsts rhs)
    fBall (compile rs) (λ(-, rhs). welldefined rhs)
  proof (safe del: fsubsetI)
    fix name rhs
    assume (name, rhs) |∈| compile rs
    then obtain irs where (name, irs) |∈| rs rhs = translate-rhs irs
      unfolding compile-def by force
    hence is-fmap irs {||} arity irs = 0
      using assms inner unfolding finished-def by blast+
    moreover have arity-compatibles irs
      using ⟨(name, irs) |∈| rs⟩ inner by (blast dest: fpairwiseD)
    ultimately obtain u where irs = {||, u} |∈| translate-rhs-def by simp+
    hence is-abs [] u
      using inner ⟨(name, irs) |∈| rs⟩ by blast
    hence closed rhs
      unfolding abs-ish-def (rhs = u) by simp
  show wellformed rhs
    using u ⟨(name, irs) |∈| rs⟩ unfolding (rhs = u)
    by blast

  have closed-except u {||}
    using u inner ⟨(name, irs) |∈| rs⟩
    by (metis (mono-tags, lifting) case-prod-conv fbspec freess-empty)
  thus closed rhs
  unfolding (rhs = u).

{assume shadowsconsts rhs
  hence shadowsconsts u
unfolding compile-def \( \langle \text{rhs} = u \rangle \) by simp
moreover have \( \neg \text{shadows-consts } u \)
  using inner \( \langle [], u \rangle \in |\text{irs} \rangle, (\text{name}, \text{irs}) \in |\text{rs} \rangle \) by blast
ultimately show False by blast

have welldefined u
  using fbspec[OF inner \( \langle \text{name}, \text{irs} \rangle \in |\text{rs} \rangle \), simplified]
  \( \langle [], u \rangle \in |\text{irs} \rangle \) by blast
thus welldefined rhs
unfolding \( \langle \text{rhs} = u \rangle \) compile-def
by simp
qed

thus
fBall \( (\text{compile } \text{rs}) \) prule
fBall \( (\text{compile } \text{rs}) \) \( (\lambda (-, \text{rhs}). \neg \text{pre-constants.shadows-consts } \text{C-info } (\text{fst } |^\sim| \text{compile } \text{rs}) \text{ rhs} ) \)
  \( (\text{compile } \text{rs}) \) \( (\lambda (-, \text{rhs}). \text{pre-constants.welldefined } \text{C-info } (\text{fst } |^\sim| \text{compile } \text{rs}) \text{ rhs} ) \)

unfolding compile-heads by auto
next
show distinct all-constructors
  by (fact distinct-ctr)
qed

theorem (in \text{irules}) compile-correct:
assumes \( \text{compile } \text{rs } \vdash_p t \rightarrow t' \) finished \text{rs}
shows \( \text{rs } \vdash_i t \rightarrow t' \)
using assms(1) proof induction
  case (\text{step name } \text{rhs})
  then obtain \text{irs} where \text{rhs} = \text{translate-rhs } \text{irs} \( \langle \text{name}, \text{irs} \rangle \in |\text{rs} \rangle \)
unfolding compile-def by force
hence arity-compatibles \text{irs}
  using inner by (blast dest: fpairwiseD)

have is-fmap \text{irs} \text{irs} \neq \{[]\} \( \text{arity } \text{irs} = 0 \)
  using assms inner \( \langle \text{name}, \text{irs} \rangle \in |\text{rs} \rangle \) unfolding finished-def by blast+
then obtain \text{u} where \text{irs} = \{[[], \text{u}]\}
  using \( \langle \text{arity-compatibles } \text{irs} \rangle \) by (metis arity-zero-shape)

show \( ?\text{case} \)
  unfolding \( \langle \text{rhs} = \rightarrow \rangle \)
  apply (rule irewrite.step)
  apply fact
  unfolding \( \langle \text{irs} = \rightarrow \rangle \) translate-rhs-def irewrite-step-def
  by (auto simp: const-term-def)
qed (auto intro: irewrite.intros)
theorem (in irules) compile-complete:
  assumes rs ⊢ i t t' finished rs
  shows compile rs ⊢ p t t'
using assms(1) proof induction
  case (step name irs params rhs t t')
  hence arity-compatibles irs
    using inner by (blast dest: fpairwiseD)
  have is-fmap irs irs ≠ {{}} arity irs = 0
    using assms inner step unfolding finished-def by blast+
  then obtain u where irs = {[[], u]}
    using (arity-compatibles irs)
    by (metis arity-zero-shape)
  with step have name, [], u ⊢ i t t'
    by simp
  hence t = Pconst name
    unfolding irewrite-step-def
    by (cases t) (auto split: if-splits simp: const-term-def)
  hence t' = u
    unfolding irewrite-step-def
    by (cases t) (auto split: if-splits simp: const-term-def)
  have (name, t') ∈| compile rs
    unfolding compile-def
  proof
    show (name, t') = map-prod id translate-rhs (name, irs)
      using (irs = []) t' = u
      by (simp add: split-beta translate-rhs-def)
  qed fact
  thus ?case
    unfolding (t = -)
    by (rule prewrite.step)
  qed (auto intro: prewrite.intros)

export-code
  compile finished
  checking Scala

end

3.4 Sequential pattern matching

theory Rewriting-Sterm
imports Rewriting-Pterm
begin

  type-synonym srule = name × sterm
abbreviation closed-srules :: srule list ⇒ bool where
closed-srules ≡ list-all (closed o snd)

primrec srule :: srule ⇒ bool where
srule (name, rhs) ≡ wellformed rhs ∧ closed rhs ∧ is-abs rhs

lemma sruleI[intro!]: wellformed rhs ⇒ closed rhs ⇒ is-abs rhs ⇒ srule (name, rhs)
by simp

locale srules = constants C-info fst | fset-of-list rs for C-info and rs :: srule list +
  assumes all-rules: list-all srule rs
  assumes distinct: distinct (map fst rs)
  assumes not-shadows: list-all (λ(-, rhs). ¬ shadows-consts rhs) rs
  assumes swelldefined-rs: list-all (λ(-, rhs). swelldefined rhs) rs
begin

lemma map: is-map (set rs)
using distinct by (rule distinct-is-map)

lemma clausesE:
  assumes (name, rhs) ∈ set rs
  obtains cs where rhs = Sabs cs
proof –
  from assms have is-abs rhs
  using all-rules unfolding list-all-iff by auto
  then obtain cs where rhs = Sabs cs
  by (cases rhs) (auto simp: is-abs-def term-cases-def)
  with that show thesis .
qed

end

Rewriting

inductive srrewrite-step where
cons-match: srrewrite-step ((name, rhs) ≠ rest) name rhs |
cons-nomatch: name ≠ name' ⇒ srrewrite-step rs name rhs ⇒ srrewrite-step ((name', rhs') ≠ rs) name rhs

lemma srrewrite-stepI0:
  assumes (name, rhs) ∈ set rs is-map (set rs)
  shows srrewrite-step rs name rhs
using assms proof (induction rs)
  case (Cons r rs)
  then obtain name' rhs' where r = (name', rhs') by force
  show ?case
  proof (cases name = name')
case False
show ?thesis
  unfolding (r = -)
  apply (rule srewrite-step.cons-nomatch)
  subgoal by fact
  apply (rule Cons)
  using False Cons(2) (r = -) apply force
  using Cons(3) unfolding is-map-def by auto
next
case True
have rhs = rhs'
  apply (rule is-mapD)
  apply fact
  unfolding (r = -)
  using Cons(2) (r = -) apply simp
  using True apply simp
  done
show ?thesis
  unfolding (r = -) (name = -) (rhs = -)
  by (rule srewrite-step.cons-match)
qed
qed auto

lemma (in srules) srewrite-stepI: (name, rhs) ∈ set rs ⇒ srewrite-step rs name rhs
using map
by (metis srewrite-stepI0)
hide-fact srewrite-stepI0

inductive srewrite :: srule list ⇒ sterm ⇒ sterm ⇒ bool (-/ ⊢ / - −→ / - [50,0,50] 50) for rs where
  step: srewrite-step rs name rhs ⇒ rs ⊢ Sconst name −→ rhs |
  beta: rs ⊢ t t' −→ ⇒ rs ⊢ Sabs cs $a t −→ t' |
  fun: rs ⊢ t $a u −→ −→ ⇒ rs ⊢ t $a u −→ t $a u' |
  arg: rs ⊢ u u' −→ −→ ⇒ rs ⊢ t $a u −→ t $a u' |

code-pred srewrite .

abbreviation srewrite-rt :: srule list ⇒ sterm ⇒ sterm ⇒ bool (-/ ⊢ / - −→*/ - [50,0,50] 50) where
  srewrite-rt rs ≡ (srewrite rs)**

global-interpretation srewrite: rewriting srewrite rs for rs
by standard (auto intro: srewrite.intros simp: app-sterm-def)+

code-pred (modes: i ⇒ i ⇒ o ⇒ bool) srewrite-step .
code-pred (modes: i ⇒ i ⇒ o ⇒ bool) srewrite .
Translation from \textit{pterm} to \textit{stern}

In principle, any function of type \((\forall a \times b) \Rightarrow (a \times b)\) \texttt{list} that orders by keys would do here. However, for simplicity’s sake, we choose a fixed one \((\texttt{ordered-fmap})\) here.

\begin{verbatim}
primrec pterm-to-stern :: pterm \Rightarrow stern where
  pterm-to-stern \ (Pconst name) = Sconst name |
  pterm-to-stern \ (Pvar name) = Svar name |
  pterm-to-stern \ (t \$\_\_ u) = pterm-to-stern \ t \$\_\_ pterm-to-stern \ u |
  pterm-to-stern \ (Pabs cs) = Sabs \ (ordered-fmap \ (map-prod \ id \ pterm-to-stern \ |\ cs))

lemma pterm-to-stern:
  assumes \ (no-abs \ t)
  shows \ (pterm-to-stern \ t = convert-term \ t)
using \ (assms \ proof \ induction)
  case \ (free \ name)
  show \ (?case
      apply simp
      apply \ (simp \ add: \ free-stern-def \ free-pterm-def)
  done
next
  case \ (const \ name)
  show \ (?case
      apply simp
      apply \ (simp \ add: \ const-stern-def \ const-ptermedef)
  done
next
  case \ (app \ t\_\_1 \ t\_\_2)
  then show \ (?case
      apply simp
      apply \ (simp \ add: \ app-stern-def \ app-ptermedef)
  done
qed

stern-to-ptermed has to be defined, for technical reasons, in \textit{CakeML-Codegen.Pterm}.

\begin{verbatim}
lemma pterm-to-stern-wellformed:
  assumes \ (wellformed \ t)
  shows \ (wellformed \ (pterm-to-stern \ t))
using \ (assms \ proof \ \ (induction \ t \ rule: \ pterm-induct)
  case \ (Pabs \ cs)
  show \ (?case
      apply simp
      unfolding \ map-prod-def \ id-apply
      apply \ (intro conjI)
      subgoal
        apply \ (subst \ list-all-iff-fset)
        apply \ (subst \ ordered-fmap-set-eq)
        apply \ (rule \ is-fmap-image)

140
\end{verbatim}
using \textit{Pabs} apply simp
apply (rule \textit{fBallI})
apply (erule \textit{fimageE})
apply auto[
using \textit{Pabs}(2) apply auto[]
apply (rule \textit{Pabs})
using \textit{Pabs}(2) by auto
subgoal
apply (rule ordered-fmap-distinct)
apply (rule \textit{is-fmap-image})
using \textit{Pabs}(2) by simp
subgoal
apply (\textit{subgoal-tac cs} \neq \{\mid\})
including \textit{fset.lifting} apply transfer
unfolding ordered-map-def
using \textit{Pabs}(2) by auto
done
qed auto

\textbf{lemma} \texttt{pterm-to-sterm-sterm-to-pterms}:
assumes \texttt{wellformed \(t\)}
shows \texttt{sterm-to-pterms \(\texttt{pterm-to-sterm} \(t\)\) = \(t\)}
using \texttt{assms} proof \(\textit{induction} \(t\)\)
case \((\textit{Pabs} cs)\)
note \texttt{fset-of-list-map[simp del]}
show \texttt{\_case}
\textit{apply} simp
\textit{unfolding} \texttt{map-prod-def} id-apply
\textit{apply} (subst ordered-fmap-image)
\textbf{subgoal}
\textit{apply} (rule \textit{is-fmap-image})
\textit{using} \textit{Pabs} \textit{by} simp
\textit{apply} (\textit{subgoal ordered-fmap-set-eq})
\textbf{subgoal}
\textit{apply} (rule \textit{is-fmap-image})
\textit{apply} (rule \textit{is-fmap-image})
\textit{using} \textit{Pabs} \textit{by} simp
\textbf{subgoal}
\textit{apply} (subst \textit{fset.map-comp})
\textit{apply} (\textit{subt map-prod-def [symmetric]}+\textit{)}
\textit{unfolding} \textit{o-def}
\textit{apply} (\textit{subt prod.map-comp})
\textit{apply} (\textit{subt id-def [symmetric]}+\textit{)}
\textit{apply} simp
\textit{apply} (\textit{subt map-prod-def})
\textit{unfolding} \textit{id-def}
\textit{apply} (\textit{rule fset-map-snd-id})
\textit{apply} simp
\textit{apply} (\textit{rule Pabs})
using Pabs(2) by (auto simp: fmember.rep-eq snds.simps)
done
qed auto

corollary pterm-to-sterm-frees: wellformed t ⇒ frees (pterm-to-sterm t) = frees t
by (metis pterm-to-sterm-sterm-to-pterm sterm-to-pterm-frees)

corollary pterm-to-sterm-closed:
closed-except t S ⇒ wellformed t ⇒ closed-except (pterm-to-sterm t) S
unfolding closed-except-def
by (simp add: pterm-to-sterm-frees)

corollary pterm-to-sterm-consts: wellformed t ⇒ consts (pterm-to-sterm t) = consts t
by (metis pterm-to-sterm-sterm-to-pterm sterm-to-pterm-consts)

corollary (in constants) pterm-to-sterm-shadows:
wellformed t ⇒ shadows-consts t ←→ shadows-consts (pterm-to-sterm t)
unfolding shadows-consts-def
by (metis pterm-to-sterm-sterm-to-pterm sterm-to-pterm-all-frees)

definition compile :: prule fset ⇒ srule list where
compile rs = ordered-fmap (map-prod id pterm-to-sterm |⁺| rs)

Correctness of translation
context prules begin
lemma compile-heads: fst |⁺| fset-of-list (compile rs) = fst |⁺| rs
unfolding compile-def
apply (subt ordered-fmap-set-eq)
apply (subt map-prod-def, subt id-apply)
apply (rule is-fmap-image)
apply (rule fmap)
apply simp
done

lemma compile-rules: srules C-info (compile rs)
proof
show list-all srule (compile rs)
  using fmap all-rules
  unfolding compile-def list-all-iff
including fset.lifting
apply transfer
apply (subt ordered-map-set-eq)
subgoal by simp
subgoal
  unfolding map-prod-def id-def
by (erule is-map-image)

subgoal
  apply (rule ballI)
  apply safe
  subgoal
    apply (rule pterm-to-sterm-wellformed)
    apply fastforce
    done
  subgoal
    apply (rule pterm-to-sterm-closed)
    apply fastforce
    apply fastforce
    done
  subgoal for - - a b
    apply (erule ballE[where x = (a, b)])
    apply (cases b; auto)
      apply (auto simp: is-abs-def term-cases-def)
    done
    done
    done
next
show distinct (map fst (compile rs))
  unfolding compile-def
  apply (rule ordered-fmap-distinct)
  unfolding map-prod-def id-def
  apply (rule is-fmap-image)
  apply (rule fmap)
  done
next
have list-all (λ(·, rhs). welldefined rhs) (compile rs)
  unfolding compile-def
  apply (subst ordered-fmap-list-all)
  subgoal
    apply (subst map-prod-def)
    apply (subst id-apply)
    apply (rule is-fmap-image)
    by (fact fmap)
    apply simp
    apply (rule fBallI)
  subgoal for x
    apply (cases x, simp)
    apply (subst pterm-to-sterm consts)
    using all-rules apply force
    using welldefined-rs by force
    done
  thus list-all (λ(·, rhs). consts rhs |⊆| pre-constants, all-consts C-info (fst |' fset-of-list (compile rs))) (compile rs)
    by (simp add: compile-heads)
next

143
interpret \( c \cdot \text{constants - fset-of-list (map fst (compile rs))} \)
  by (simp add: constants-axioms compile-heads)
have all-consts: \( c \cdot \text{all-consts = all-consts} \)
  by (simp add: compile-heads)

note fset-of-list-map[simp del]
have list-all \((\lambda (-, rhs). \neg \text{shadows-consts rhs}) (\text{compile rs})\)
  unfolding compile-def
  apply (subst list-all-iff-fset)
  apply (subst ordered-fmap-set-eq)
  apply (subst map-prod-def)
  unfolding id-apply
  apply (rule is-fmap-image)
  apply (fact fmap)
  apply simp
  apply (rule \text{fBall-pred-weaken[where } P = \lambda (-, rhs). \neg \text{shadows-consts rhs}]\))
subgoal for \( x \)
  apply (cases \( x \), simp)
  apply (subst (asm)\term-to-sterm-shadows)
  using all-rules apply force
  by simp
subgoal
  using not-shadows by force
done
thus list-all \((\lambda (-, rhs). \neg \text{pre-constants. shadows-consts C-info (fst | fset-of-list (compile rs)) rhs}) (\text{compile rs})\)
  unfolding compile-heads all-consts.
next
show fdisjnt \((\text{fst | fset-of-list (compile rs)}) C\)
  unfolding compile-def
  apply (subst fset-of-list-map[symmetric])
  apply (subst ordered-fmap-keys)
  apply (subst map-prod-def)
  apply (subst id-apply)
  apply (rule is-fmap-image)
  using fmap disjnt by auto
next
show distinct all-constructors
  by (fact distinct-ctr)
qed

sublocale prules-as-srules: srules C-info compile rs
by (fact compile-rules)
end

global-interpretation srelated: term-struct-rel-strong \((\lambda p. p = \text{term-to-pterms})\)
proof (standard, goal-cases)
\textbf{lemma} srelated-subst:
\begin{itemize}
\item \textbf{assumes} srelated \ P-env penv senv
\item \textbf{shows} subst (stern-to-pterm t) penv = stern-to-pterm (subst t senv)
\end{itemize}
\textbf{using} assms
\textbf{proof} (induction t arbitrary: penv senv)
\begin{itemize}
\item \textbf{case} (Svar name)
\item \textbf{thus} ?case by (cases rule: fnrel-cases\[where x = name\]) auto
\end{itemize}
next
\begin{itemize}
\item \textbf{case} (Sabs cs)
\item \textbf{show} ?case
\item \textbf{apply} simp including fset.lifting
\item \textbf{apply} (transfer' fixing: cs penv senv)
\item \textbf{unfolding} set-map image-comp
\item \textbf{apply} (rule image-cong[OF refl])
\item \textbf{unfolding} comp-apply
\item \textbf{apply} (case-tac x)
\item \textbf{apply} hypsubst-thin
\item \textbf{apply} simp
\item \textbf{apply} (rule Sabs)
\item \textbf{apply assumption}
\item \textbf{apply} (simp add: snds.simps)
\item \textbf{apply} rule
\item \textbf{apply} (rule Sabs)
\item \textbf{done}
\end{itemize}
\textbf{qed} auto

\textbf{context} begin

\textbf{private lemma} srewrite-step-non-empty: srewrite-step rs' name rhs \implies rs' \neq []
\textbf{by} (induct rule: srewrite-step.induct) auto

\textbf{private lemma} compile-consE:
\begin{itemize}
\item \textbf{assumes} (name, rhs') \# rest = compile rs is-fmap rs
\item \textbf{obtains} rhs where rhs' = pterm-to-sterm rhs (name, rhs) |\in| rs rest = compile (rs - {\{(name, rhs)\} |})
\end{itemize}
\textbf{proof} –
\item \textbf{from} assms \textbf{have} ordered-fmap (map-prod id pterm-to-sterm |\cdot| rs) = (name, rhs') \# rest
unfolding compile-def
by simp
hence (name, rhs') ∈ set (ordered-fmap (map-prod id pterm-to-sterm |\'| rs))
by simp

have (name, rhs') |∈| map-prod id pterm-to-sterm |\'| rs
apply (rule ordered-fmap-sound)
subgoal
unfolding map-prod-def id-apply
apply (rule is-fmap-image)
done
subgoal by fact
done
then obtain rhs where rhs' = pterm-to-sterm rhs (name, rhs) |∈| rs
by auto

have rest = compile (rs − { | (name, rhs) | })
unfolding compile-def
apply (subt inj-on-fimage-set-diff[where C = rs])
subgoal
apply (rule inj-onI)
apply safe
apply auto
apply (subt (asm) fmember.rep-eq[symmetric])+
using (is-fmap rs) by (blast dest: is-fmapD)
subgoal by simp
subgoal using (| (name, rhs) |∈| rs) by simp
subgoal
apply simp
apply (subt ordered-fmap-remove)
apply (subt map-prod-def)
unfolding id-apply
apply (rule is-fmap-image)
done

show thesis
by (rule that) fact+
qed

private lemma compile-correct-step:
assumes srewrite-step (compile rs) name rhs is-fmap rs fBall rs prule
shows (name, stern-to-pterms rhs) |∈| rs
using assms proof (induction compile rs name rhs arbitrary: rs)
case (cons-match name rhs rest)
then obtain rhs where rhs = pterm-to-sterm rhs (name, rhs) ∈ rs
    by (auto elim: compile-consE)

show ?case
  unfolding (rhs' = -)
  apply (subst pterm-to-sterm-to-pterms)
  using fbspec[OF fBall rs prule; (name, rhs) ∈ rs] apply force
by fact

next
  case (cons-nomatch name name1 rest rhs rhs1)
  then obtain rhs1 where rhs1 = pterm-to-sterm rhs1 (name1, rhs1) ∈ rs rest
  = compile (rs - {| (name1, rhs1) |})
  by (auto elim: compile-consE)

  let ?rs' = rs - {| (name1, rhs1) |}
  have (name, sterm-to-pterm rhs) ∈ ?rs'
  proof (intro cons-nomatch)
    show rest = compile ?rs'
      by fact
  
    show is-fmap (rs |- {|(name1, rhs1)|})
    using (is-fmap rs)
    by (rule is-fmap-subset auto)

    show fBall ?rs' prule
    using cons-nomatch by blast
  qed

  thus ?case
  by simp

qed

lemma compile-correct0:
  assumes compile rs ⊢ u → u' prules C rs
  shows rs ⊢ p sterm-to-pterms u → sterm-to-pterms u'
using assms proof induction
  case (beta cs t t')
  then obtain pat rhs env where (pat, rhs) ∈ set cs match pat t = Some env t'
  = subst rhs env
  by (auto elim: rewrite-firstE)

  then obtain env' where match pat (sterm-to-pterms t) = Some env' srelated.P-env env' env
  by (metis option.distinct(1) option.inject option.rel-cases srelated.match-rel)

  hence subst (sterm-to-pterms rhs) env' = sterm-to-pterms (subst rhs env)
  by (simp add: srelated-subst)

let ?rhs' = sterm-to-pterms rhs
have \((\text{pat}, \text{?rhs}') |\in| \text{fset-of-list (map (map-prod id \text{sterm-to-pterm}) \text{cs})}\)
using \((\text{pat}, \text{rhs}) \in \text{set cs})
including \text{fset.lifting}
by \text{transfer'} force

note \text{fset-of-list-map[simp del]}
show ?case
  apply simp
  apply (rule prewrite.intros)
  apply fact
unfolding \text{rewrite-step.simps}
apply (subst map-option-eq-Some)
apply (intro \text{exI conjI})
apply fact
unfolding \((t' = \cdot)\)
by fact

next
case \((\text{step name rhs})\)
hence \((\text{name, \text{sterm-to-pterm rhs}) |\in| \text{rs})\)
unfolding \text{prules-def prules-axioms-def}
by \text{metis compile-correct-step}?
thus ?case
  by (auto intro: prewrite.intros)
qed (auto intro: prewrite.intros)

end

lemma \((\text{in prules}) \text{compile-correct}:\)
assumes \(\text{compile rs} \vdash_s u \longrightarrow u'\)
shows \(\text{rs} \vdash_p \text{sterm-to-pterm u} \longrightarrow \text{sterm-to-pterm u}'\)
by (rule compile-correct0) \text{(fact | standard)+}

hide-fact \text{compile-correct0}

Completeness of translation

global-interpretation \text{related'}: \text{term-struct-rel-strong (}\lambda p. \text{pterm-to-sterm p = s)}
proof (standard, goal-cases)
case \((1 t \text{name})\)
  then show ?case by (cases t) (auto simp: \text{const-sterm-def const-pterm-def split: option.splits})
next
case \((3 t u_1 u_2)\)
  then show ?case by (cases t) (auto simp: \text{app-sterm-def app-pterm-def split: option.splits})
qed (auto simp: \text{const-sterm-def const-pterm-def app-sterm-def app-pterm-def})

corollary \text{related-env-unique}:
\( \text{related}' \cdot P \text{-env } penv \text{ senv } \Rightarrow \text{related}' \cdot P \text{-env } penv' \text{ senv'} \Rightarrow \text{senv} = \text{senv}' \) 

apply (subst (asm) fmrel-iff)+
apply (subst (asm) option.rel-sel)+
apply (rule fmap-ext)
by (metis option.exhaust-sel)

lemma \text{related-subst}':
assumes \text{related}' \cdot P \text{-env } penv \text{ senv wellformed } t
shows \text{pterm-to-sterm} \ (\text{subst } t \text{ penv}) = \text{subst} \ (\text{pterm-to-sterm } t) \text{ senv}
using \text{assms} proof (induction arbitrary: penv senv)
case (P\text{var } \text{name})
thus ?case
by (cases rule: fmrel-cases[where \text{x} = \text{name}]) auto
next
case (P\text{abs } cs)
hence \text{is-fmap } cs
by force

show ?case
apply simp
unfolding \text{map-prod-def} id-apply
apply (subst ordered-fmap-image[symmetric])
apply fact
apply (subst \text{fset.map-comp}[symmetric])
apply (subst ordered-fmap-image[symmetric])
subgoal by (rule \text{is-fmap-image}) fact
apply (subst ordered-fmap-image[symmetric])
apply fact
apply auto
apply (drule ordered-fmap-sound[OF \langle \text{is-fmap } cs \rangle])
subgoal for \text{pat } \text{rhs}
apply (rule P\text{abs})
apply (subst (asm) fmember.rep-eq)
apply assumption
apply auto
using P\text{abs} by force+
done
qed auto

lemma \text{related-find-match}:
assumes \text{find-match } cs \ t = \text{Some} (penv, pat, rhs) \text{ related}' \cdot P \text{-env } penv \text{ senv}
shows \text{find-match} \ (\text{map } (\text{map-prod id pterm-to-sterm}) \text{ cs}) \ (\text{pterm-to-sterm } t) = \text{Some } (\text{senv}, \text{pat}, \text{pterm-to-sterm } \text{rhs})
proof --
let \( \text{cs'} \) = \text{map } (\text{map-prod id pterm-to-sterm}) \text{ cs}
let \( \text{t'} \) = \text{pterm-to-sterm } t
have \*: \text{\text{list-all2} } \text{(rel-prod } (=) \ (\lambda p s. \text{pterm-to-sterm } p = s)) \text{ cs } ?\text{cs}' \)
unfolding \text{list.rel-map}
by (auto intro: \text{list.rel-refl})

149
obtain senv0
  where find-match ?cs' ?t' = Some (senv0, pat, pterm-to-sterm rhs) srelated'.P-env penv senv0
  using srelated'.find-match-rel[OF * refl, where t = t, unfolded assms]
  unfolding option-rel-Some1 rel-prod-conv
  by auto
with assms have senv = senv0
  by (metis srelated-env-unique)
show ?thesis
  unfolding ⟨senv = -⟩ by fact
qed

lemma (in prules) compile-complete:
  assumes rs |− p t −→ t' wellformed t
  shows compile rs |− s pterm-to-sterm t −→ pterm-to-sterm t'
using assms proof induction
case (step name rhs)
then show ?case
  apply simp
  apply rule
  apply (rule prules-as-srules.srewrite-stepI)
  unfolding compile-def
  apply (subst fset-of-list-elem[symmetric])
  apply (subst ordered-fmap-set-eq)
  apply (insert fmap)
  apply (rule is-fmapI)
  apply (force dest: is-fmapD)
  by (simp add: rev-fimage-eqI)
next
case (beta c cs t t')
from beta obtain pat rhs penv where c = (pat, rhs) match pat t = Some penv
subst rhs penv = t'
  by (metis (no-types, lifting) map-option-eq-Some rewrite-step.simps surj-pair)
then obtain senv where match pat (pterm-to-sterm t) = Some senv srelated'.P-env penv senv
  by (metis option-rel-Some1 srelated'.match-rel)
have wellformed rhs
  using beta (c = -) prules.all-rules prule.simps
  by force
then have subst (pterm-to-sterm rhs) senv = pterm-to-sterm t'
  using srelated-subst' (- = t') (srelated'.P-env - -)
  by metis
have (pat, pterm-to-sterm rhs) |∈| map-prod id pterm-to-sterm | · | cs
  using beta (c = -)
  by (metis fimage-eqI id-def map-prod-simp)
have is-fmap cs
  using beta
  by auto
have find-match (ordered-fmap cs) t = Some (penv, pat, rhs)
apply (rule compatible-find-match)
subgoal
apply (subst ordered-fmap-set-eq[OF ⟨is-fmap cs⟩])+
using beta by simp
subgoal
unfolding list-all-iff
apply rule
apply (rename-tac x, case-tac x)
apply simp
apply (drule ordered-fmap-sound[OF ⟨is-fmap cs⟩])
using beta by auto
subgoal
apply (subst ordered-fmap-set-eq)
by fact
subgoal
by fact
subgoal
using beta(1) (c = - ⟨is-fmap cs⟩)
using fset-of-list-elem ordered-fmap-set-eq by fast
done

show ?case
apply simp
apply rule
apply (subst :- = pterm-to-stern t'[symmetric])
apply (rule find-match-rewrite-first)
unfolding map-prod-def id-apply
apply (subst ordered-fmap-image[symmetric])
apply fact
apply (subst map-prod-def[symmetric])
apply (subst id-def[symmetric])
apply (rule srelated-find-match)
by fact+
qed (auto intro: srewrite.intros)

Computability
export-code compile
  checking Scala

end

3.5 Big-step semantics

theory Big-Step-Sterm
imports
  Rewriting-Sterm
  ../Terms/Term-as-Value
3.5.1 Big-step semantics evaluating to irreducible sterms

inductive (in constructors) seval :: srule list ⇒ (name, sterm) fmap ⇒ sterm ⇒ sterm ⇒ bool (\l / \Gamma s / - / - [50,0,50] 50) for rs where
const: (name, rhs) ∈ set rs ⇒ rs, Γ ⊢ Sconst name ↓ rhs |
var: fmlookup Γ name = Some val ⇒ rs, Γ ⊢ Svar name ↓ val |
abs: rs, Γ ⊢ Sabs cs ↓ Sabs (map (λ(pat, t). (pat, subst t (fmdrop-fset (frees pat) Γ))) cs) |
comb: rs, Γ ⊢ Sabs cs t ↓ Sabs cs u u′ =⇒ rs, Γ ⊢ Sabs cs t $s u ↓ val |
constr: name ∈| C =⇒ list-all2 (seval rs Γ) ts us =⇒ 
rs, Γ ⊢ Sconst name $s ts ↓ name $s us

lemma (in constructors) seval-closed:
assumes rs, Γ ⊢ Sconst name u closed-srules rs closed-env Γ closed-except t (fmdom Γ)
shows closed u
using assms proof induction

next case (comb Γ t cs u u′ env pat rhs val)
hence closed (Sabs cs) closed u′
  by (auto simp: closed-except-def)

moreover have (pat, rhs) ∈ set cs match pat u′ = Some env
  using comb by (auto simp: find-match-elem)
ultimately have closed-except rhs (frees pat)
  by (auto dest: closed-except-sabs)

show ?case

proof (rule comb)

have closed-env env
  by (rule closed.match) fact+

thus closed-env (Γ ++f env)
  using ⟨closed-env Γ⟩ by auto

next

have closed-except rhs (fmdom Γ |∪| frees pat)
  using ⟨closed-except rhs ⟩

unfolding closed-except-def by auto
hence closed-except rhs (fmdom Γ |∪| fmdom env)
  using ⟨match pat u′ = Some env⟩ by (metis match-dom)

thus closed-except rhs (fmdom (Γ ++f env))

152
using comb by simp
qed fact
next
case (abs Γ cs)
show ?case
  apply (subst subst-term.simps[symmetric])
  apply (subst closed-except-def)
  apply (subst subst-frees)
  apply fact
  apply (subst fminus-fsubset-conv)
  apply (subst closed-except-def[symmetric])
  apply (subst funion-fempty-right)
  apply fact
  done
next
case (constr name Γ ts us)
have list-all closed us
  using ⟨list-all2 - - - ⟩ ⟨closed-except (list-comb - - -) - - ⟩
proof (induction ts us rule: list.rel-induct)
  case (Cons v vs u us)
  thus ?case
    using constr unfolding closed.list-comb
    by auto
  qed simp
  thus ?case
    unfolding closed.list-comb
    by (simp add: closed-except-def)
  qed auto

lemma (in srules) seval-wellformed:
  assumes rs, Γ Γ', t ⊢ u wellformed t wellformed-env Γ
  shows wellformed u
using assms proof induction
  case (const name rhs Γ)
  thus ?case
    using all-rules
    by (auto simp: list-all-iff)
next
case (comb Γ t cs u u' env pat rhs val)
hence (pat, rhs) ∈ set cs match pat u' = Some env
  by (auto simp: find-match-elem)
show ?case
  proof (rule comb)
    show wellformed rhs
      using ⟨(pat, rhs) ∈ set cs; comb ⟩
      by (auto simp: list-all-iff)
next
  have wellformed-env env
apply (rule wellformed.match)
apply fact
apply (rule comb)
using comb apply simp
apply fact+
done
thus wellformed-env (Γ ++ f env)
using comb by auto
qed

next
case (abs Γ cs)
thus ?case
by (metis subst-sterm.simps subst-wellformed)

next
case (constr name Γ ts us)
have list-all wellformed us
using ⟨list-all2 - - -⟩ ⟨wellformed (list-comb - -)⟩
proof (induction ts us rule: list.rel-induct)
case (Cons v vs u us)
thus ?case
using constr by (auto simp: wellformed.list-comb app-sterm-def)
qed simp
thus ?case
by (simp add: wellformed.list-comb const-sterm-def)
qed auto

lemma (in constants) seval-shadows:
assumes rs, Γ ⊢ s t u ↓ ¬ shadows-consts t
assumes list-all (λ(-, rhs). ¬ shadows-consts rhs) rs
assumes not-shadows-consts-env Γ
shows ¬ shadows-consts u
using assms proof induction
case (const name rhs Γ)
thus ?case
unfolding srules-def
by (auto simp: list-all-iff)
next
case (comb Γ t cs u u' env pat rhs val)
hence ¬ shadows-consts (Sabs cs) ¬ shadows-consts u'
by auto
moreover from comb have (pat, rhs) ∈ set cs match pat u' = Some env
by (auto simp: find-match-elem)
ultimately have ¬ shadows-consts rhs
by (auto simp: list-ex-iff)
moreover have not-shadows-consts-env env
using comb ⟨match pat u' = -⟩ by (auto intro: shadows.match)
ultimately show ?case
using comb by blast

next
case (\(\mu\) \(\Gamma\) cs)
show \(\mu\)case
apply \(\mu\)case
apply (rule subst-shadows)
apply fact+
done

next
case (constr name \(\Gamma\) ts us)
have list-all (Not o shadows-cons) us
using (\(\mu\)list2 - -) (\(\sim\) shadows-cons (name \$$\$\) ts)
proof (induction ts us rule: list.rel-induct)
case (Cons v vs u us)
thus \(\mu\)case
using constr by (auto simp: shadows.list-comb const-sterm-def app-sterm-def)
qed simp
thus \(\mu\)case
by (auto simp: shadows.list-comb list-ex-iff list-all-iff const-sterm-def)
qed auto

lemma (in constructors) seval-list-comb-abs:
assumes rs, \(\Gamma\) \(\vdash\) name \$$\$\$$ args \$$\downarrow\$$ Sabs cs
shows name \(\in\) dom (map-of rs)
using assms
proof (induction \(\Gamma\) name \$$\$\$$ args Sabs cs arbitrary: args cs)
case (constr name' - - us)
hence Sabs cs = name' \$$\$\$$ us by simp
hence False
by (cases rule: list-comb-cases) (auto simp: const-sterm-def app-sterm-def)
thus \(\mu\)case ..
next
case (comb \(\Gamma\) \(t\) cs' \(u\)' env pat rhs)
hence strip-comb \(t\) \$$s\$$ \(u\) = strip-comb (name \$$\$\$$ args)
by simp
hence strip-comb \(t\) = (Sconst name, butlast args) \(u\) = last args
apply --
subgoal
apply (simp add: strip-list-comb-const)
apply (fold app-sterm-def const-sterm-def)
by (auto split: prod.splits)
subgoal
apply (simp add: strip-list-comb-const)
apply (fold app-sterm-def const-sterm-def)
by (auto split: prod.splits)
done
hence \(t\) = name \$$\$\$$ butlast args
apply (fold const-sterm-def)
by (metis list-strip-comb fst-conv snd-conv)

thus ?case
using comb by auto
qed (auto elim; list-comb-cases simp: const-sterm-def app-sterm-def intro; weak-map-of-SomeI)

lemma (in constructors) is-value-eval-id:
assumes is-value t closed t
shows rs, Γ ⊢ s t ↓ t
using assms proof induction
  case (abs cs)
    have rs, Γ ⊢ s (Sabs cs) ↓ Sabs (map (λ(pat, t). (pat, subst t (fmdrop-fset (frees pat) Γ))) cs)
      by (rule seval.abs)
    moreover have subst (Sabs cs) Γ = Sabs cs
      using abs by (metis subst-closed-id)
    ultimately show ?case
      by simp
next
  case (constr vs name)
    have list-all2 (seval rs Γ) vs vs
      proof (rule list.rel-refl-strong)
        fix v
        assume v ∈ set vs
        moreover hence closed v
        using constr
        unfolding closed.list-comb
        by (auto simp: list-all-iff)
      ultimately show rs, Γ ⊢ s v ↓ v
        using ⟨list-all - -⟩
        by (force simp: list-all-iff)
    qed
    with ⟨name |∈| C⟩ show ?case
      by (rule seval.constr)
  qed

lemma (in constructors) ssubst-eval:
assumes rs, Γ ⊢ s t ↓ t' Γ' ⊆_f Γ closed-env Γ value-env Γ
shows rs, Γ ⊢ s subst t Γ' ⊢ t'
using assms proof induction
  case (var Γ name val)
  show ?case
    proof (cases fmlookup Γ' name)
      case None
      thus ?thesis
        using var by (auto intro: seval.intros)
    next
      case (Some val')
with var have val' = val
  unfolding fnmsub-alt-def by force
show ?thesis
  apply simp
  apply (subst Some)
  apply (subst (val' = \_))
  apply simp
  apply (rule is-value-eval-id)
  using var by auto
qed

next
  case (abs \Gamma cs)
  hence subst (Sabs cs) \Gamma' \Gamma = subst (Sabs cs) \Gamma
    by (metis subst-twice fnmsubset-pred)
moreover have rs, \Gamma \sqsubset_s subst (Sabs cs) \Gamma' \sqsubseteq subst (Sabs cs) \Gamma' \Gamma
  apply simp
  apply (subst map-map[ symmetric ])
  apply (rule seval.abs)
  done
ultimately have rs, \Gamma \sqsubset_s subst (Sabs cs) \Gamma' \sqsubseteq subst (Sabs cs) \Gamma
  by metis
thus ?case by simp

next
  case (constr name \Gamma ts us)
  hence list-all2 (\lambda t. seval rs \Gamma (subst t \Gamma')) ts us
    by (blast intro: list.rel-mono-strong)
  with constr show ?case
    by (auto simp: subst-list-comb list-all2-map1 intro: seval.constr)
qed (auto intro: seval.intros)

lemma (in constructors) seval-agree-eq:
  assumes rs, \Gamma \sqsubseteq_s t \sqsubseteq u fnrestrict-fset S \Gamma = fnrestrict-fset S \Gamma' closed-except t
  S
  assumes S \subseteq fmdom \Gamma closed-srules rs closed-env \Gamma
  shows rs, \Gamma' \sqsubseteq_s t \sqsubseteq u
using assms proof (induction arbitrary: \Gamma' S)
  case (var \Gamma name val)
  hence name \in S
    by (simp add: closed-except-def)
  hence fnlookup \Gamma name = fnlookup \Gamma' name
    using fnrestrict-fset S \Gamma = \_
  unfolding fnfilter-alt-defs
  including fmap.lifting
  by transfer' (auto simp: map-filter-def fun-eq-iff split: if-splits)
with var show ?case
  by (auto intro: seval.var)
next
  case (abs \Gamma cs)
— Intentionally local: not really a useful lemma outside of its scope

have ∗: fmdrop-fset S (fmrestrict-fset T m) = fmrestrict-fset (T ∪ S) (fmdrop-fset S m) for S T m

unfolding fmfilter-alt-defs fmfilter-comp
by (rule fmfilter-cong) auto

{  
  fix pat t  
  assume (pat, t) ∈ set cs  
  with abs have closed-except t (S ∪ frees pat)  
  by (auto simp: Sterm.closed-except-simps list-all-iff)

  have subst t (fmdrop-fset (frees pat) (fmrestrict-fset S Γ)) = subst t (fmdrop-fset (frees pat) Γ)  
  apply (subst ∗)  
  apply (rule subst-restrict-closed)  
  apply fact  
  done

  moreover have subst t (fmdrop-fset (frees pat) (fmrestrict-fset S Γ′)) = subst t (fmdrop-fset (frees pat) Γ′)  
  apply (subst ∗)  
  apply (rule subst-restrict-closed)  
  apply fact  
  done

  ultimately have subst t (fmdrop-fset (frees pat) Γ) = subst t (fmdrop-fset (frees pat) Γ′)  
  using abs by metis

}

hence map (λ(pat, t). (pat, subst t (fmdrop-fset (frees pat) Γ))) cs =  
map (λ(pat, t). (pat, subst t (fmdrop-fset (frees pat) Γ′))) cs  
by auto

thus ?case  
by (metis seval.abs)

next

  case (comb Γ t cs u u′ env pat rhs val)
  have fmdom env = frees pat  
  apply (rule match-dom)  
  apply (rule find-match-elem)  
  apply fact  
  done

  show ?case  
  proof (rule seval.comb)
show \( rs, \Gamma' \vdash_s t \downarrow \) \( \text{Sabs} \) \( cs \) \( rs \), \( \Gamma' \vdash_s u \downarrow u' \)
using \text{comb} by (auto simp: \text{Sterm.closed-except-simps})

next
show \( rs, \Gamma' \text{++}_f \text{env} \vdash_s \text{rhs} \downarrow \text{val} \)
proof (rule \text{comb})
  have \( \text{fmrestrict-fset} \ (S \cup \text{fmdom env}) \ (\Gamma \text{++}_f \text{env}) = \text{fmrestrict-fset} \ (S \cup \text{fmdom env}) \ (\Gamma' \text{++}_f \text{env}) \)
  using \text{comb}(8)
  unfolding \text{fnfilter-alt-defs}
  including \text{fmap.lifting fset.lifting}
  by transfer' (auto simp: \text{map-filter-def fun-eq-iff map-add-def split: option.splits if-splits})
thus \( \text{fmrestrict-fset} \ (S \cup \text{frees pat}) \ (\Gamma \text{++}_f \text{env}) = \text{fmrestrict-fset} \ (S \cup \text{frees pat}) \ (\Gamma' \text{++}_f \text{env}) \)
unfolding \( \text{findom env} = \cdot \).
next
have \( \text{closed-except \ t S} \)
  using \text{comb} by (simp add: \text{Sterm.closed-except-simps})

have \( \text{closed} \ (\text{Sabs cs}) \)
apply (rule seval-closed)
apply fact+
using (\text{closed-except \ t S}) \ (S \subseteq \text{fmdom} \Gamma)
unfolding \text{closed-except-def} apply simp
done

have \( (\text{pat}, \text{rhs}) \in \text{set cs} \)
  using (\text{find-match \ - = \ -}) by (rule \text{find-match-elem})
hence \( \text{closed-except \ rhs} \ (\text{frees pat}) \)
  using (\text{closed \ (Sabs cs)}) by (auto dest: \text{closed-except-sabs})
thus \( \text{closed-except \ rhs} \ (S \cup \text{frees pat}) \)
  unfolding \text{closed-except-def} by auto
next
show \( S \cup \text{frees pat} \subseteq \text{fmdom} \ (\Gamma \text{++}_f \text{env}) \)
apply simp
apply (intro \text{conjI})
using \text{comb}(10) apply blast
unfolding \( \text{fmdom env} = \cdot \) by blast
next
have \( \text{closed-except \ u S} \)
  using \text{comb} by (auto simp: \text{closed-except-def})

show \( \text{closed-env} \ (\Gamma \text{++}_f \text{env}) \)
apply rule
apply fact
apply (rule \text{closed.match[where \ t = u' and \ pat = pat]})
subgoal
by (rule \text{find-match-elem}) fact
subgoal
apply (rule seval-closed)
apply fact+
using (closed-except u S) (S |⊆| fmdom Γ) unfolding closed-except-def

by blast

done
qed fact

qed fact

next

case (constr name Γ ts us)
show ?case
apply (rule seval.constr)
apply fact
apply (rule list.rel-mono-strong)
apply fact
using constr
unfolding closed.list-comb list-all-iff
by auto
qed (auto intro: seval.intros)

Correctness wrt srwre

category srules

private lemma seval-correct0:
assumes rs, Γ ⊢ t s u closed-except t (fmdom Γ) closed-env Γ
shows rs ⊢ subst t Γ −→∗ u
using assms proof induction

case (const name rhs Γ)

have srewrite-step rs name rhs
by (rule srewrite-stepI) fact
thus ?case
by (auto intro: srewrite.intros)

next

case (comb Γ t cs u u' env pat rhs val)
hence closed-except t (fmdom Γ) closed-except u (fmdom Γ)
by (simp add: Stem.closure-except-simps)+
moreover have closed-srules rs
using all-rules
unfolding list-all-iff by fastforce
ultimately have closed (Sabs cs) closed u'
using comb by (metis seval-closed)+

from comb have (pat, rhs) ∈ set cs match pat u' = Some env
by (auto simp: find-match-elem)
hence closed-except rhs (frees pat)
using (closed (Sabs cs)) by (auto dest: closed-except-sabs)
hence frees rhs |⊆| frees pat
by (simp add: closed-except-def)
moreover have fmdom env = frees pat
  using (match pat u' = taxes) by (auto simp: match-dom)
ultimately have frees rhs \subseteq| fmdom env
  by simp
hence subst rhs (\Gamma ++ \text{f env}) = subst rhs env
  by (rule subst-add-shadowed-env)

have \text{rs} \vdash \text{s subst t \Gamma} \rightarrow^{*} \text{Sabs cs \$ s subst u \Gamma}
  closed-except-simps)
also have \text{rs} \vdash \text{s Sabs cs \$ s subst u' \Gamma}
  using comb \langle \text{match pat u' = Some env} \rangle \langle \text{fmdom env} = \_ \rangle \langle \text{frees rhs} |\subseteq\langle \text{frees pat} \rangle \rangle
  by (auto simp: closed-except-def)
next
  show closed-env (\Gamma ++ \text{f env})
    using comb \langle \text{match pat u' = Some env} \rangle \langle \text{closed u'} \rangle
    by (blast intro: closed.match)
qed

finally show ?case by simp
next
  case (constr name \Gamma ts us)
  show ?case
    apply (simp add: subst-list-comb)
    apply (rule srewrite.rt-list-comb)
    subgoal
      apply (simp add: list.rel-map)
      apply (rule list.rel-mono-strong[OF constr(2)])
      apply clarify
      apply (elim impE)
      using constr(3) apply (erule closed.list-combE)
      apply (rule constr)+
      apply (auto simp: const-sterm-def)
    done
    subgoal by auto
    done
  qed auto

corollary seval-correct:
  assumes \text{rs, fnempty \vdash_s t \rightarrow^{*} u closed t}
  shows \text{rs \vdash_s t} \rightarrow^{*} \text{u}
proof

  have closed-except t (fndom fnempty)
  using assms by simp
  with assms have rs ⊢ subst t fnempty —> u
    by (fastforce intro: seval-correct0)
  thus ?thesis
    by simp
qed

end

theory Big-Step-Value

imports
  Big-Step-Sterm
  ../Terms/Value

begin

3.5.2 Big-step semantics evaluating to value

primrec vrule :: vrule ⇒ bool where
vrule (-, rhs) —> svwellformed rhs ∧ vclosed rhs ∧ ¬ is-Vconstr rhs

locale vrules = constants C-info fst | ' | fset-of-list rs
  for C-info and rs :: vrule list +
  assumes all-rules: list-all vrule rs
  assumes distinct: distinct (map fst rs)
  assumes not-shadows: list-all (λ(-, rhs). not-shadows-vconsts rhs) rs
  assumes vconstructor-value-rs: vconstructor-value-rs rs
  assumes vwelldefined-rs: list-all (λ(-, rhs). vwelldefined rhs) rs

begin

lemma map: is-map (set rs)
  using distinct by (rule distinct-is-map)
end

abbreviation value-to-sterm-rules :: vrule list ⇒ srule list where
value-to-sterm-rules ≡ map (map-prod id value-to-sterm)

inductive (in special-constants)
  veval :: (name × value) list ⇒ (name, value) fmap ⇒ sterm ⇒ value ⇒ bool (-,
  -/ ⊢ v, -/ : [50,0,50] 50) for rs where
  const: (name, rhs) ∈ set rs ⇒ rs, Γ ⊢ v Sconst name ⊢ rhv
  var: fnlookup Γ name = Some val ⇒ rs, Γ ⊢ v Svar name ⊢ val
  abs: rs, Γ ⊢ Sabs cs ⊢ Vabs cs Γ |
  comb:
    rs, Γ ⊢ t ⊢ Vabs cs Γ' ⇒ rs, Γ ⊢ v u ⊢ u' ⇒
    vfind-match cs u' = Some (env, -, rhs) ⇒
\[ \text{rs, } \Gamma' + \{f\} \text{ env } \vdash_v \text{ rhs } \downarrow \text{ val } \Rightarrow \]
\[ \text{rs, } \Gamma \vdash_v t \downarrow s, u \downarrow \text{ val } | \]

\text{rec-comb:}
\[ \text{rs, } \Gamma \vdash_v t \downarrow \text{ Vrecabs css name } \Gamma' \Rightarrow \]
\[ \text{fmlookup css name = Some cs } \Rightarrow \]
\[ \text{rs, } \Gamma \vdash_v u \downarrow a' \Rightarrow \]
\[ \text{vfind-match cs } u' = \text{ Some (env, } - \text{, rhs) } \Rightarrow \]
\[ \text{rs, } \Gamma' + \{f\} \text{ env } \vdash_r \text{ rhs } \downarrow \text{ val } \Rightarrow \]
\[ \text{rs, } \Gamma \vdash_v t \downarrow s, u \downarrow \text{ val } | \]

\text{constr:}
\[ \text{name } \in C \Rightarrow \]
\[ \text{list-all2 (vval rs } \Gamma) \text{ ts us } \Rightarrow \]
\[ \text{rs, } \Gamma \vdash_v \text{ name } \downarrow s, u \downarrow \text{ Vconstr name us } \]

\text{lemma (in } \nu\text{rules) vval-wellformed:}
\[ \text{assumes } \text{rs, } \Gamma \vdash_v t \downarrow v \text{ wellformed } t \text{ wellformed-venv } \Gamma \]
\[ \text{shows } \text{wwellformed } v \]
\[ \text{using } \text{assms proof induction} \]
\[ \text{case } \text{const} \]
\[ \text{thus } \text{?case} \]
\[ \text{using } \text{all-rules} \]
\[ \text{by (auto simp: list-all-iff)} \]
\[ \text{next} \]
\[ \text{case } \text{comb} \]
\[ \text{show } \text{?case} \]
\[ \text{apply (rule comb)} \]
\[ \text{using } \text{comb by (auto simp: list-all-iff dest: vfind-match-elem intro: vwell-formed.vmatch-env)} \]
\[ \text{next} \]
\[ \text{case } \text{(rec-comb } \Gamma t \text{ css name } \Gamma' \text{ cs } u \text{ } u' \text{ env pat rhs val)} \]
\[ \text{hence (pat, rhs) } \in \text{ set cs vmatch (mk-pat pat) } u' = \text{ Some env} \]
\[ \text{by (metis vfind-match-elem)}+ \]

\[ \text{show } \text{?case} \]
\[ \text{proof (rule rec-comb)} \]
\[ \text{have wellformed } t \]
\[ \text{using rec-comb by simp} \]
\[ \text{have } \text{wwellformed } (\text{Vrecabs css name } \Gamma') \]
\[ \text{by (rule rec-comb) fact+} \]
\[ \text{thus wellformed rhs} \]
\[ \text{using rec-comb (pat, rhs) } \in \text{ set cs} \]
\[ \text{by (auto simp: list-all-iff)} \]

\[ \text{have wellformed-venv } \Gamma' \]
\[ \text{using } \text{wwellformed } (\text{Vrecabs css name } \Gamma') \text{ by simp} \]
\[ \text{moreover have wellformed-venv env} \]
\[ \text{using rec-comb vmatch (mk-pat pat) } u' = \text{ Some env} \]
\[ \text{by (auto intro: wellformed.vmatch-env)} \]
\[ \text{ultimately show wellformed-venv } (\Gamma' + \{f\} \text{ env)} \]

163
by blast
qed

next

case (constr name Γ ts us)
have list-all wellformed us
  using (list-all2 - - -) (wellformed (list-comb - -))
proof (induction ts us rule: list.rel-induct)
  case (Cons v vs u us)
   with constr show ?case
   unfolding wellformed.list-comb by auto
   qed simp
   thus ?case
   by (simp add: list-all-iff)
   qed auto

lemma (in vrules) veval-closed:
  assumes rs, Γ ⊢ v t ↓ v closed-except t (fdom Γ) closed-venv Γ
  assumes wellformed t wellformed-venv Γ
  shows vclosed v
  using assms proof induction
  case (const name rhs Γ)
   hence (name, rhs) ∈ set rs
     by (auto dest: map-of-SomeD)
   thus ?case
   using const all-rules
   by (auto simp: list-all-iff)
next
  case (comb Γ t cs Γ' u' env pat rhs val)
   hence pat: (pat, rhs) ∈ set cs vmatch (mk-pat pat) u' = Some env
     by (metis vfind-match-elem)+
show ?case
  proof (rule comb)
    have vclosed u'
      using comb by (auto simp: Sterm.closed-except-simps)
    have closed-venv env
      by (rule vclosed.vmatch-env) fact+
    thus closed-venv (Γ' ++ f env)
      using comb by (auto simp: Sterm.closed-except-simps)
next
  have wellformed t
    using comb by simp
  have vwellformed (Vabs cs Γ')
    by (rule veval-wellformed) fact+
  thus wellformed rhs
    using pat by (auto simp: list-all-iff)
  have wellformed-venv Γ'
    using (vwellformed (Vabs cs Γ')) by simp
moreover have \textit{wellformed-venv} env

using \textit{comb} pat
by (auto intro: \textit{vwellformed.vmatch-env veval-wellformed})

ultimately show \textit{wellformed-venv} (\Gamma' +++ f env)
by blast

have \textit{vclosed} (Vabs cs \Gamma')
using \textit{comb} by (auto simp: \textit{Sterm.closed-except-simps})
hence closed-except rhs \textit{fmdom} \Gamma' [\cup] \textit{frees pat}
using \textit{pat} by (auto simp: list-all-iff)

moreover have \textit{fmdom env} = \textit{frees pat}
using \textit{vwellformed} (Vabs cs \Gamma')\textit{pat}
by (auto simp: \textit{vmatch-dom mk-pat-frees list-all-iff})

ultimately show \textit{closed-except rhs} \textit{fmdom} (\Gamma' +++ f env))
using \textit{vclosed} (Vabs cs \Gamma')
by simp

qed

next

\textbf{case} (\textit{rec-comb} \Gamma t css name \Gamma' cs u u' env rhs val)
hence \textit{pat} (\textit{pat}, rhs) \in set cs \textit{vmatch} (mk-pat pat) u' = Some env
by (metis \textit{vfind-match-elem}+)

show ?case

proof (rule \textit{rec-comb})

have \textit{vclosed u'}
using \textit{rec-comb} by (auto simp: \textit{Sterm.closed-except-simps})

have \textit{closed-venv env}
by (rule \textit{vclosed.vmatch-env} fact+)
thus \textit{closed-venv} (\Gamma' +++ f env)
using \textit{rec-comb} by (auto simp: \textit{Sterm.closed-except-simps})

next

have \textit{wellformed} t
using \textit{rec-comb} by simp

have \textit{vwellformed} (Vrecabs css name \Gamma')
by (rule \textit{veval-wellformed} fact+)
thus \textit{wellformed} rhs
using \textit{pat} \textit{rec-comb} by (auto simp: list-all-iff)

have \textit{wellformed-venv} \Gamma'
using \textit{vwellformed} (Vrecabs css name \Gamma') by simp

moreover have \textit{wellformed-venv} env
using \textit{rec-comb pat}
by (auto intro: \textit{vwellformed.vmatch-env veval-wellformed})

ultimately show \textit{wellformed-venv} (\Gamma' +++ f env)
by blast

have \textit{wellformed-clauses} cs
using \( \langle \text{wellformed} \ (V_{\text{recabs \ css \ name}} \ \Gamma') \rangle \langle \text{fmlookup \ css \ name} = \text{Some \ cs} \rangle \)

by auto

have \( \text{vclosed} \ (V_{\text{recabs \ css \ name}} \ \Gamma') \)

using \( \text{rec-comb by} \ (\text{auto \ simp: \ Stem.\text{closed-except-simps}}) \)

hence \( \text{closed-except \ rhs} \ (\text{fmdom} \ \Gamma' \ [\cup] \ \text{frees \ pat}) \)

using \( \text{rec-comb \ pat \ by} \ (\text{auto \ simp: \ list-all-iff}) \)

moreover have \( \text{fmdom \ env} = \text{frees \ pat} \)

using \( \langle \text{wellformed-clauses \ cs} \ \rangle \ \text{pat} \)

by \( \text{(auto \ simp: \ list-all-iff \ Sterm.\text{closed-except-simps})} \)

ultimately show \( \text{closed-except \ rhs} \ (\text{fmdom} \ (\Gamma' \ ++ \ \text{f \ env})) \)

using \( \langle \text{vclosed} \ (V_{\text{recabs \ css \ name}} \ \Gamma') \rangle \)

by simp

qed

next

case (\( \text{constr \ name} \ \Gamma \ \text{ts \ us} \))

have \( \text{list-all \ vclosed \ us} \)

using \( \langle \text{list-all2 - - -} \rangle \ \langle \text{closed-except} \ (\cdot \ \$$ \cdot\) \ \langle \text{wellformed} \ (\text{name} \ \$$ \text{ts}) \rangle \)

proof (induction \( \text{ts \ us} \))

case \( \text{Cons \ v \ vs \ u \ us} \)

with \( \text{constr} \)

show \(?case\)

unfolding \( \text{closed-list-comb} \ \text{wellformed.list-comb} \)

by \( \text{(auto \ simp: \ list-all-iff \ Stem.\text{closed-except-simps})} \)

qed simp

thus \(?case\)

by \( \text{(simp \ add: \ list-all-iff)} \)

qed (auto \ simp: \ Sterm.\text{closed-except-simps})

lemma (in vrules) veval-constructor-value:

assumes \( rs, \ \Gamma \vdash \text{v \ t \ \text{\downarrow \ v}} \) \( \text{vconstructor-value-env \ \Gamma} \)

shows \( \text{vconstructor-value} \ v \)

using \( \text{assms \ proof \ induction} \)

case \( \text{(comb} \ \Gamma \ \text{t \ cs \ \Gamma' \ u \ u' \ \text{env \ pat \ rhs \ val)} \)

hence \( \text{(pat, \ rhs) \ \in \ \text{set \ cs \ vmatch \ (mk-pat \ pat) \ u' = \text{Some \ env})} \)

by \( \text{(metis \ vfind-match-elem)+} \)

hence \( \text{vconstructor-value-env} \ (\Gamma' \ ++ \ f \ \text{env}) \)

using \( \text{comb \ by} \ (\text{auto \ intro: \ vconstructor-value.vmatch-env}) \)

thus \(?case\)

using \( \text{comb \ by \ auto} \)

next

case \( \text{constr \ name} \ \Gamma \ \text{ts \ us} \)

hence \( \text{list-all \ vconstructor-value \ us} \)

by \( \text{(auto \ elim: \ list-all2-rightE)} \)

with \( \text{constr} \)

show \(?case\)

by simp

next

case \( \text{const} \)

thus \(?case\)

using \( \text{vconstructor-value-rs} \)

qed
by (auto simp: list-all-iff vconstructor-value-rs-def)

next

case (rec-comb Γ t css name Γ' cs u u' env pat rhs val)

hence (pat, rhs) ∈ set cs vmatch (mk-pat pat) u' = Some env
  by (metis vfind-match-elem)

hence vconstructor-value-env (Γ' + f env)
  using rec-comb by (auto intro: vconstructor-value.vmatch-env)

thus ?case
  using rec-comb by auto

qed (auto simp: list-all-iff vconstructor-value-rs-def)

lemma (in vrules) veval-welldefined:
assumes rs, Γ ⊢ v t ↓ v fmpred (λ-. vwelldefined) Γ welldefined t
shows vwelldefined v
using assms proof induction
  case const
    thus ?case
      using vwelldefined-rs assms
      unfolding list-all-iff
      by (auto simp: list-all-iff)

next

case (comb Γ t cs Γ' u u' env rhs val)

hence vwelldefined (Vabs cs Γ')
  by auto

show ?case
  proof (rule comb)
    have fmpred (λ-. vwelldefined) Γ'
      using vwelldefined (Vabs cs Γ')
      by simp
    moreover have fmpred (λ-. vwelldefined) env
      apply (rule vwelldefined.vmatch-env)
      apply (rule vfind-match-elem)
      using comb by auto
    ultimately show fmpred (λ-. vwelldefined) (Γ' + f env)
      by auto

next

  have (pat, rhs) ∈ set cs
    using comb by (metis vfind-match-elem)
  thus welldefined rhs
    using vwelldefined (Vabs cs Γ')
    by (auto simp: list-all-iff)
  qed

next

case (rec-comb Γ t css name Γ' cs u u' env pat rhs val)

have (pat, rhs) ∈ set cs
  by (rule vfind-match-elem) fact

show ?case
  proof (rule rec-comb)
show \( \text{fmpred} (\lambda-. v\text{welldefined}) (\Gamma' + + f \text{env}) \)
proof
  show \( \text{fmpred} (\lambda-. v\text{welldefined}) \text{env} \)
  using rec-comb by (auto dest: vfind-match-elem intro: vwelldefined.vmatch-env)
next
  show \( \text{fmpred} (\lambda-. v\text{welldefined}) \Gamma' \)
  using rec-comb by auto
qed
next
have vwelldefined \((V\text{recabs css name} \Gamma')\)
  using rec-comb by auto
thus welldefined rhs
  apply simp
  apply (elim conjE)
  apply (drule fmpredD[where \( m = css \)])
  using \((\text{pat, rhs}) \in \text{set cs})\) rec-comb by (auto simp: list-all-iff)
qed
next
case \((\text{constr name} \Gamma' \text{ts us})\)
  have list-all vwelldefined us
  using \(\text{list-all2 - -} \rightarrow \) welldefined \( - \rightarrow \))
  proof (induction ts us rule: list.rel-induct)
    case \( \text{Cons v vs u us} \)
    with constr show \(?\text{case}\)
    unfolding welldefined.list-comb
    by auto
    qed simp
    with constr show \(?\text{case}\)
    by (simp add: list-all-iff all-consts-def)
next
case abs
  thus \(?\text{case}\)
  unfolding welldefined-sabs by auto
qed auto

Correctness wrt constructors.seval

context vrules begin

definition \( rs' :: \text{srule list where} \)
rs' = value-to-sterm-rules rs
lemma value-to-sterm-srules: \( \text{srules C-info rs'} \)
proof
  show distinct \((\text{map \( \text{fst} rs' \)})\)
  unfolding rs'-def
  using distinct by auto
next
show list-all srule rs'

unfolding rs'-def list.pred-map
apply (rule list.pred-mono-strong[OF all-rules])
apply (auto intro: vclosed.value-to-sterm vwellformed.value-to-sterm)
subgoal by (auto intro: vwellformed.value-to-sterm)
subgoal by (auto intro: vclosed.value-to-sterm)
subgoal for a b by (cases b) (auto simp: is-abs-def term-cases-def)
done

next
show fdisjnt (fst | | fset-of-list rs') C
  using vconstructor-value-rs unfolding rs'-def vconstructor-value-rs-def
  by auto
interpret c: constants - fst | | fset-of-list rs'
  by standard (fact | fact distinct-ctr)+
have all-consts: c.all-consts = all-consts
  unfolding c.all-consts-def all-consts-def
  by (simp add: rs'-def)
have shadows-consts: c.shadows-consts rhs = shadows-consts rhs for rhs :: sterm
  by (induction rhs; fastforce simp: all-consts list-ex-iff)

have list-all (λ(-, rhs). ¬ shadows-consts rhs) rs'
  unfolding rs'-def
  unfolding list.pred-map map-prod-def id-def case-prod-twice list-all-iff
  apply auto
  unfolding comp-def all-consts-def
  using not-shadows
  by (fastforce simp: list-all-iff dest: not-shadows-vconsts.value-to-sterm)
thus list-all (λ(-, rhs). ¬ c.shadows-consts rhs) rs'
  unfolding shadows-consts .

have list-all (λ(-, rhs). welldefined rhs) rs'
  unfolding rs'-def list.pred-map
  apply (rule list.pred-mono-strong[OF vwelldefined-rs])
subgoal for z
  apply (cases z; hypsubst-thin)
  apply simp
  apply (erule vwelldefined.value-to-sterm)
done
done
moreover have map fst rs = map fst rs'
  unfolding rs'-def by simp
ultimately have list-all (λ(-, rhs). welldefined rhs) rs'
  by simp
thus list-all (λ(-, rhs). c.welldefined rhs) rs'
  unfolding all-consts .
next
show distinct all-constructors
  by (fact distinct-ctr)
qed
When we evaluate terms using `veval`, the result is a value which possibly contains a closure (constructor `Vabs`). Such a closure is essentially a case-lambda (like `Sabs`), but with an additionally captured environment of type `string \rightarrow value` (which is usually called \(\Gamma'\)). The contained case-lambda might not be closed.

The proof idea is that we can always substitute with \(\Gamma'\) and obtain a regular term value. The only interesting part of the proof is the case when a case-lambda gets applied to a value, because in that process, a hidden environment is *unveiled*. That environment may not bear any relation to the active environment \(\Gamma\) at all. But pattern matching and substitution proceeds only with that hidden environment.

```plaintext
context vrules begin

private lemma veval-correct0:
  assumes rs, \(\Gamma \vdash v\)
  wellformed t wellformed-venv \(\Gamma\)
  assumes closed-except t (fmdom \(\Gamma\)) closed-venv \(\Gamma\)
  assumes econstructor-value-venv \(\Gamma\)
  shows rs', fmap value-to-sterm \(\Gamma\) \(\vdash\) t \(\downarrow\) value-to-sterm v
using assms proof induction

  case (constr name \(\Gamma\) ts us)
    have list-all2 (seval rs' (fmmapping value-to-sterm \(\Gamma\))) ts (map value-to-sterm us)
    unfolding list-all2-map2
    proof (rule list.rel-mono-strong[OF list-all2 - - -], elim conjE impE)
      fix t u
      assume t \(\in\) set ts u \(\in\) set us
      assume rs, \(\Gamma \vdash v\)
      show wellformed t closed-except t (fmdom \(\Gamma\))
        using (\(t \in\) set ts) constr
        unfolding wellformed.list-comb closed.list-comb list-all-iff
        by auto
      qed (rule constr | assumption)+

  thus \?case
    using \(\langle name |\in| C\rangle\)
    by (auto intro: seval.constr)

next
  case (comb \(\Gamma\) t cs \(\Gamma'\) u u' venv pat rhs val)
    -- We first need to establish a ton of boring side-conditions.

  hence vmatch (mk-pat pat) u' = Some venv
```

170
by (auto dest: vfind-match-elem)

have wellformed t
  using comb by simp
have vwellformed (Vabs cs Γ')
  by (rule veval-wellformed) fact+

hence
  list-all (linear o fst) cs
  wellformed-venv Γ'
  by (auto simp: list-all-iff split-beta)

have rel-option match-related (vfind-match cs u') (find-match cs (value-to-sterm u'))
  apply (rule find-match-eq)
  apply fact
  apply (rule veval-constructor-value)
  apply fact+
  done

then obtain senv
  where find-match cs (value-to-sterm u') = Some (senv, pat, rhs)
  and env-eq venv senv
  using vfind-match - - = -
  by cases auto
hence (pat, rhs) ∈ set cs match pat (value-to-sterm u') = Some senv
  by (auto dest: find-match-elem)
hence fmdom senv = frees pat
  by (simp add: match-dom)

moreover have senv = fmmap value-to-sterm venv
  using (env-eq venv senv)
  by (rule env-eq-eq)

ultimately have fmdom venv = frees pat
  by simp

have closed-except t (fmdom Γ) wellformed t
  using comb by (simp add: closed-except-def)+
have vclosed (Vabs cs Γ')
  by (rule veval-closed) fact+

have vconstructor-value (Vabs cs Γ') vconstructor-value u'
  by (rule veval-constructor-value; fact)+
hence vconstructor-value-env Γ'
  by simp
have vconstructor-value-env venv
  by (rule vconstructor-value-wmatch-env fact+)
have wellformed u
  using comb by simp
have vwellformed u′
  by (rule veval-wellformed) fact+
have wellformed-venv venv
  by (rule vwellformed.venmatch-env) fact+

have closed-except u (fmdom Γ)
  using comb by (simp add: closed-except-def)
have vclosed u′
  by (rule veval-closed) fact+
have closed-venv venv
  by (rule vclosed.venmatch-env) fact+

have closed-venv Γ′
  using 〈vclosed (Vabs cs Γ′)〉 by simp

let ?subst = λpat t. subst t (fmdrop-fset (fmdom Γ) (fmmap value-to-sterm Γ′))

1. We know the following (induction hypothesis): rs′, fmmap value-to-sterm (Γ′ ++ f venv) ⊢ s rhs ↓ value-to-sterm val

2. ... first, we can deduce using ssubst-eval that this is equivalent to substituting rhs first: rs′, fmmap value-to-sterm (Γ′ ++ f venv) ⊢ s subst rhs (fmdrop-fset (fmdom Γ) (fmmap value-to-sterm Γ′)) ↓ value-to-sterm val

3. ... second, we can replace the hidden environment Γ′ by the active environment Γ using seval-agree-eq because it does not contain useful information at this point: rs′, fmmap value-to-sterm (Γ ++ f venv) ⊢ s subst rhs (fmdrop-fset (fmdom Γ) (fmmap value-to-sterm Γ′)) ↓ value-to-sterm val

4. ... finally we can apply a step in the original semantics and arrive at the conclusion: rs′, fmmap value-to-sterm Γ ⊢ s t $u ↓ value-to-sterm val

have rs′, fmmap value-to-sterm (Γ′ ++ f venv) ⊢ s ?subst pat rhs $u ↓ value-to-sterm val

proof (rule ssubst-eval)
  show rs′, fmmap value-to-sterm (Γ′ ++ f venv) ⊢ s rhs $u ↓ value-to-sterm val
  proof (rule comb)
    have linear pat closed-except rhs (fmdom Γ′ |∪| fmdom venv) (fmmap value-to-sterm Γ′)
      using ⟨(pat, rhs) ∈ set cs⟩ (vwellformed (Vabs cs Γ′)) (vclosed (Vabs cs Γ′))
      by (auto simp: list-all-iff)
    hence closed-except rhs (fmdom Γ′ |∪| fmdom venv)
      using (vmatch (mk-pat rhs) u′ = Some venv)
      by (auto simp: mk-pat-frees vmatch-dom)
    thus closed-except rhs (fmdom (Γ′ ++ f venv))
      by simp
  next
  show wellformed-venv (Γ′ ++ f venv)

172
using \( \text{wellformed-venv} \, \Gamma \) \( \langle \text{wellformed-venv} \, \text{venv} \rangle \)
by blast
next
show \( \text{closed-venv} \, (\Gamma' + + f \, \text{venv}) \)
using \( \langle \text{closed-venv} \, \Gamma' \rangle \, \langle \text{closed-venv} \, \text{venv} \rangle \)
by blast
next
show \( \text{vconstructor-value-env} \, (\Gamma' + + f \, \text{venv}) \)
using \( \langle \text{vconstructor-value-env} \, \Gamma' \rangle \, \langle \text{vconstructor-value-env} \, \text{venv} \rangle \)
by blast
next
show \( \text{wellformed} \, \text{rhs} \)
using \( \langle \langle \text{pat}, \, \text{rhs} \rangle \in \text{set cs} \rangle \) \( \langle \text{vwellformed} \, \langle \text{Vabs cs} \, \Gamma' \rangle \rangle \)
by (fastforce simp: list-all-iff)
qed
next
have \( \text{fmdrop-fset} \, (\text{fmdom} \, \text{venv}) \) \( \Gamma' \subseteq f \, \Gamma' + + f \, \text{venv} \)
including \( \text{fmap.\text{lifting fset.\text{lifting}}} \)
by transfer' (auto simp: map-drop-set-def map-filter-def map-le-def map-add-def split: if-splits)
thus \( \text{fmdrop-fset} \, \langle \text{frees} \, \text{pat} \rangle \, (\text{fmmap} \, \text{value-to-sterm} \, \Gamma') \subseteq f \, \text{fmmap} \, \text{value-to-sterm} \, (\Gamma' + + f \, \text{venv}) \)
unfolding \( \langle \text{fmdom} \, \text{venv} \, = \, \text{frees} \, \text{pat} \rangle \)
by (metis \text{fmdrop-fset-fmmap} \text{fmmap-subset})
next
show \( \text{closed-env} \, (\text{fmmap} \, \text{value-to-sterm} \, (\Gamma' + + f \, \text{venv})) \)
apply auto
apply rule
apply (rule \text{vclosed.value-to-sterm-env, fact})+
done
next
show \( \text{value-env} \, (\text{fmmap} \, \text{value-to-sterm} \, (\Gamma' + + f \, \text{venv})) \)
apply auto
apply rule
apply (rule \text{vconstructor-value.value-to-sterm-env, fact})+
done
qed
show \(?\text{case}\)
proof (rule \text{seval.comb})
have \( \text{rs'} \, \text{fmmap} \, \text{value-to-sterm} \, \Gamma \vdash_s t \downarrow \text{value-to-sterm} \, (\text{Vabs} \, \text{cs} \, \Gamma') \)
using \text{comb} by (auto simp: closed-except-def)
thus \( \text{rs'} \, \text{fmmap} \, \text{value-to-sterm} \, \Gamma \vdash_s t \downarrow \text{Sabs} \, (\lambda(\text{pat}, \, t). \, (\text{pat}, \, \text{?subst pat t}) \, \text{cs}) \)
by simp
next
show \( \text{rs'} \, \text{fmmap} \, \text{value-to-sterm} \, \Gamma \vdash_s u \downarrow \text{value-to-sterm} \, u' \)
using \text{comb} by (simp add: closed-except-def)
next
  show $rs'$, fmmap value-to-sterm $\Gamma' ++ f$ $venv \vdash_s ?\text{subst pat}$ $rhs \downarrow$ value-to-sterm $val$

  proof (rule seval-agree-eq)
  show $rs'$, fmmap value-to-sterm $\Gamma' ++ f$ fmmap value-to-sterm $venv \vdash_s ?\text{subst pat}$ $rhs \downarrow$ value-to-sterm $val$
    using $rs'$, fmmap value-to-sterm $(\Gamma' ++ f$ $venv)$ $\vdash_s ?\text{subst pat}$ $rhs \downarrow$ value-to-sterm $val$; by simp
  next
    show fmrestrict-fset (frees pat) (fmmap value-to-sterm $\Gamma' ++ f$ fmmap value-to-sterm $venv$) =
      fmrestrict-fset (frees pat) (fmmap value-to-sterm $\Gamma' ++ f$ $venv$)
    unfolding (senv = -)
    apply (subst (fmdom $venv$ = frees pat); symmetric)+
    apply (subst fmadd-restrict-right-dom)
    apply (subst fmadd-restrict-right-dom)
    apply simp
    done
  next
    have closed (value-to-sterm (Vabs cs $\Gamma'$))
      using (vclosed (Vabs cs $\Gamma'$)); by (rule vclosed.value-to-sterm)
    thus closed-except (subst rhs (fmdrop-fset (frees pat) (fmmap value-to-sterm $\Gamma'$))) (frees pat)
      using (pat, rhs) $\in$ set cs
      by (auto simp: Sterm.closed-except-simps list-all-iff)
  next
    show closed-env (fmmap value-to-sterm $\Gamma' ++ f$ fmmap value-to-sterm $venv$)
      using (closed-venv $\Gamma'$ (closed-venv $venv$)
      by (auto intro: vclosed.value-to-sterm-env)
  next
    show frees pat $\subseteq$ fmdom (fmmap value-to-sterm $\Gamma' ++ f$ fmmap value-to-sterm $venv$)
      using (fmdom $venv$ = frees pat)
      by fastforce
  next
    show closed-srules $rs'$
      using all-rules
      unfolding $rs'$-def list-all-iff
      by (fastforce intro: vclosed.value-to-sterm)
  qed

next
  show find-match (map ($\lambda$(pat, t). (pat, ?$\text{subst pat}$ t)) cs) (value-to-sterm $u'$) = Some (senv, pat, ?$\text{subst pat}$ $rhs$)
    using (find-match - - = -)
    by (auto simp: find-match-map)
qed
next
— Basically a verbatim copy from the comb case

\textbf{case} (\texttt{rec-comb} $\Gamma$ $t$ css name $\Gamma'$ $cs$ $u$ $u'$ venv $rhs$ $val$)

\textbf{hence} \texttt{vmatch} (\texttt{mk-pat} $pat$) $u'$ = \texttt{Some} venv
\hspace{1em} by (auto dest: \texttt{vfind-match-elem})

\textbf{have} $cs$ = the (\texttt{fmlookup} css name)
\hspace{1em} using \texttt{rec-comb} by simp

\textbf{have} wellformed $t$
\hspace{1em} using \texttt{rec-comb} by simp
\textbf{have} \texttt{vwellformed} (\texttt{Vrecabs} css name $\Gamma'$)
\hspace{1em} by (rule veval-wellformed) fact+
\textbf{hence} \texttt{vwellformed} (\texttt{Vabs} $cs$ $\Gamma'$) — convenient hack: $cs$ is not really part of a \texttt{Vabs}
\hspace{1em} using \texttt{rec-comb} by auto

\textbf{hence}
\hspace{1em} list-all (linear $\circ$ \texttt{fst}) $cs$
\hspace{1em} wellformed-venv $\Gamma'$
\hspace{1em} by (auto simp: list-all-iff split-beta)

\textbf{have} rel-option match-related (\texttt{vfind-match} $cs$ $u'$) (\texttt{find-match} $cs$ (\texttt{value-to-sterm} $u'$))
\hspace{1em} apply (rule \texttt{find-match-eq})
\hspace{1em} apply \texttt{fact}
\hspace{1em} apply (rule veval-constructor-value)
\hspace{1em} apply \texttt{fact+}
\hspace{1em} done

\textbf{then obtain} \texttt{senv}
\hspace{1em} where \texttt{find-match} $cs$ (\texttt{value-to-sterm} $u'$) = \texttt{Some} ($\texttt{senv}$, $\texttt{pat}$, $\texttt{rhs}$)
\hspace{1em} and \texttt{env-eq} venv \texttt{senv}
\hspace{1em} using (\texttt{vfind-match} - - = -)
\hspace{1em} by cases auto
\textbf{hence} ($\texttt{pat}$, $\texttt{rhs}$) $\in$ set $cs$ match $pat$ (\texttt{value-to-sterm} $u'$) = \texttt{Some} \texttt{senv}
\hspace{1em} by (auto dest: \texttt{find-match-elem})
\textbf{hence} \texttt{fmdom} \texttt{senv} = \texttt{frees} $\texttt{pat}$
\hspace{1em} by (simp add: \texttt{match-dom})

\textbf{moreover have} \texttt{senv} = \texttt{fnmap} \texttt{value-to-sterm} \texttt{venv}
\hspace{1em} using (\texttt{env-eq} \texttt{venv} \texttt{senv})
\hspace{1em} by (rule \texttt{env-eq-eq})

\textbf{ultimately have} \texttt{fmdom} \texttt{venv} = \texttt{frees} \texttt{pat}
\hspace{1em} by \texttt{simp}
have \( \text{closed-except } t \ (\text{fmdom } \Gamma) \text{ wellformed } t \)
using \( \text{rec-comb by (simp add: closed-except-def)} \)
have \( \text{vclosed (Vreccabs css name } \Gamma') \)
by (rule veval-closed) fact+
hence \( \text{vclosed (Vabs cs } \Gamma') \)
using \( \text{rec-comb by auto} \)

have \( \text{vconstructor-value } u' \)
by (rule veval-constructor-value) fact+
have \( \text{vconstructor-value (Vreccabs css name } \Gamma') \)
by (rule veval-constructor-value) fact+
hence \( \text{vconstructor-value-env } \Gamma' \)
by simp
have \( \text{vconstructor-value-env venv} \)
by (rule veval-constructor-value.env) fact+

have wellformed \( u \)
using \( \text{rec-comb by simp} \)
have \( \text{vwellformed } u' \)
by (rule veval-wellformed) fact+
have \( \text{wellformed-venv venv} \)
by (rule vclosed.vmatch-env) fact+

have \( \text{closed-except } u \ (\text{fmdom } \Gamma) \)
using \( \text{rec-comb by (simp add: closed-except-def)} \)
have \( \text{vclosed } u' \)
by (rule veval-closed) fact+
have \( \text{closed-venv venv} \)
by (rule vclosed.vmatch-env) fact+

have \( \text{closed-venv } \Gamma' \)
using \( \text{vclosed (Vabs cs } \Gamma') \)
by simp

let \(?\text{subst} = \lambda \text{pat t. subst t (fmdrop-fset (frees pat) (fnmap value-to-sterm } \Gamma')})\)
have \( \text{rs', fnmap value-to-sterm (}\Gamma' + + f \text{ venv) } \vdash \ _s \ ?\text{subst pat rhs } \downarrow \ \text{value-to-sterm val} \)
proof (rule ss subst-eval)
  show \( \text{rs', fnmap value-to-sterm (}\Gamma' + + f \text{ venv) } \vdash \ _s \ \text{rhs } \downarrow \ \text{value-to-sterm val} \)
proof (rule rec comb)
  have \( \text{linear pat closed-except rhs (fmdom } \Gamma' |\cup| \text{ frees pat}) \)
  using \((\text{pat, rhs) } \in \text{ set cs}) \ \text{vwellformed (Vabs cs } \Gamma') \ \text{vclosed (Vabs cs } \Gamma')\)
by (auto simp: list all iff)
hence \( \text{closed-except rhs (fmdom } \Gamma' |\cup| \text{ fmdom venv}) \)
using \( \text{vmatch (mk-pat pat) } u' = \text{ Some venv} \)
by (auto simp: mk-pat frees vmatch dom)
thus \( \text{closed-except rhs (fmdom (}\Gamma' + + f \text{ venv})} \)
by simp
show wellformed-venv \( (\Gamma' + + f \ venv) \)
using \( \langle \text{wellformed-venv} \ \Gamma' \langle \text{wellformed-venv} \ venv \rangle \) by blast

show closed-venv \( (\Gamma' + + f \ venv) \)
using \( \langle \text{closed-venv} \ \Gamma' \langle \text{closed-venv} \ venv \rangle \) by blast

show vconstructor-value-env \( (\Gamma' + + f \ venv) \)
using \( \langle \text{vconstructor-value-env} \ \Gamma' \langle \text{vconstructor-value-env} \ venv \rangle \) by blast

show wellformed rhs
using \( \langle \text{(pat, rhs)} \in \text{set cs} \rangle \langle \text{vwellformed} \ (\text{Vabs cs} \ \Gamma') \rangle \)
by (fastforce simp: list-all-iff)

qed

have fmdrop-fset \( (\text{fmdom venv}) \) \( \Gamma' \subseteq f \ \Gamma' + + f \ venv \)
including fmap.lifting fset.lifting
by transfer'
(auto simp: map-drop-set-def map-filter-def map-le-def map-add-def split: if-splits)
thus fmdrop-fset \( (\text{frees pat}) \) \( \text{fmmap value-to-sterm} \ \Gamma' \subseteq f \ \text{fmmap value-to-sterm} \ (\Gamma' + + f \ venv) \)
unfolding \( \langle \text{fmdom venv} = \text{frees pat} \rangle \)
by (metis fmdrop-fset-fmmap fmmap-subset)

show closed-env \( \text{fmmap value-to-sterm} \ (\Gamma' + + f \ venv) \)
apply auto
applypule
apply (rule vclosed.value-to-sterm-env, fact)+
done

next

next

show value-env \( \text{fmmap value-to-sterm} \ (\Gamma' + + f \ venv) \)
apply auto
applypule
apply (rule vconstructor-value.value-to-sterm-env, fact)+
done

qed

show ?case

proof (rule seval.comb)

have rs', fmmap value-to-sterm \( \Gamma \vdash_s \ t \downarrow \text{value-to-sterm} \ (Vrecabs \ cs \ \text{name} \ \Gamma') \)
using rec-comb by (auto simp: closed-except-def)
thus rs', fmmap value-to-sterm \( \Gamma \vdash_s \ t \downarrow \text{Sabs} \ (\lambda (\text{pat}, \ t). \ (\text{pat}, \ ?\text{subst} \ \text{pat} \ t)) \ cs \)
unfolding \( \langle cs = - \rangle \) by simp
next show \( rs', \text{fmmap value-to-sterm} \Gamma \vdash_s u \downarrow \text{value-to-sterm} u' \)
using rec-comb by (simp add: closed-except-def)

next show \( rs', \text{fmmap value-to-sterm} \Gamma' ++ f \text{fmmmap value-to-sterm} venv \vdash_s \) subst pat rhs \( \downarrow \) value-to-sterm val

proof (rule seval-agree-eq)
show \( rs', \text{fmmap value-to-sterm} \Gamma' ++ f \text{fmmmap value-to-sterm} venv \vdash_s \) subst pat rhs \( \downarrow \) value-to-sterm val
using \( rs', \text{fmmap value-to-sterm} (\Gamma' ++ f \text{venv}) \vdash_s \) subst pat rhs \( \downarrow \) value-to-sterm val by simp

next show \( \text{fmmrestrict-fset (frees pat)} (\text{fmmap value-to-sterm} \Gamma' ++ f \text{fmmmap value-to-sterm} \text{venv}) = \)
\( \text{fmmrestrict-fset (frees pat)} (\text{fmmap value-to-sterm} \Gamma' ++ f \text{fmmmap value-to-sterm} \text{venv}) \)

unfolding (venv = -)
apply (subst (fmdom venv = frees pat)[symmetric])+
apply (subst fmdom-map[symmetric])
apply (subst fnadd-restrict-right-dom)
apply (subst fmdom-map[symmetric])
apply (subst fnadd-restrict-right-dom)
apply simp
done

next have closed (value-to-sterm (Vabs cs \( \Gamma' \)))
using (vclosed (Vabs cs \( \Gamma' \)))
by (rule vclosed.value-to-sterm)
thus closed-except (subst rhs (fndrop-fset (frees pat)) (fmmap value-to-sterm \( \Gamma' \)))(frees pat)
using ((pat, rhs) \in set cs)
by (auto simp: Sterm.closed-except-simps list-all-iff)

next show closed-env (fmmap value-to-sterm \( \Gamma' ++ f \text{fmmmap value-to-sterm} \text{venv} \))
using (closed-venv \( \Gamma' \) (closed-venv venv)
by (auto intro: vclosed.value-to-sterm-env)
next show frees pat \( \subseteq \) fndom (fmmap value-to-sterm \( \Gamma' ++ f \text{fmmmap value-to-sterm} \text{venv} \))
using (fndom venv = frees pat)
by fastforce
next show closed-srules rs'
using all-rules
unfolding rs'-def list-all-iff
by (fastforce intro: vclosed.value-to-sterm)
qed

next show find-match (map (\( \lambda \) (pat, t), (pat, subst pat t)) cs) (value-to-sterm u')
Some (senv, pat, ?subst pat rhs)
    using (find-match - - = ~)
    by (auto simp: find-match-map)
qed
next
case (const name rhs Γ)
  show ?case
    apply (rule seval.const)
    unfolding rs′-def
    using (name, rhs) ∈ ~ by force
next
case abs
  show ?case
    by (auto simp del: fmdrop-fset-fmmap intro: seval.abs)
qed (auto intro: seval.var seval.abs)

lemma veval-correct:
assumes rs, fmempty ⊢ v t ↓ v wellformed t closed t
shows rs′, fmempty ⊢ s t ↓ value-to-sterm v
proof
  have rs′, fmmap value-to-sterm fmempty ⊢ s t ↓ value-to-sterm v
  using assms
    by (auto intro: veval-correct0 simp del: fmmap-empty)
  thus ?thesis
    by simp
qed

end end

3.5.3 Big-step semantics with conflation of constants and
variables

theory Big-Step-Value-ML
imports Big-Step-Value
begin

definition mk-rec-env :: (name, sclauses) fmap ⇒ (name, value) fmap ⇒ (name, value) fmap where
  mk-rec-env css Γ′ = fmmap-keys (λname cs. Vrecabs css name Γ′) css

context special-constants begin

inductive veval′ :: (name, value) fmap ⇒ sterm ⇒ value ⇒ bool (~/ ~/ ~/ ~ ~/ ~ ~/ [50,0,50] 50) where
const: name |∅| C ⇒ fmlookup Γ name = Some val ⇒ Γ ⊢⊥ Sconst name ⊥ val
| var: fmlookup Γ name = Some val ⇒ Γ ⊢⊥ Svar name ⊥ val |

abs: $\Gamma \vdash v \text{sabs}\hspace{0.5em}c\text{s} \downarrow V\text{sabs}\hspace{0.5em}c\text{s} \hspace{0.5em}\Gamma$ 

comb:
$$\Gamma \vdash v t \downarrow V\text{sabs}\hspace{0.5em}c\text{s} \hspace{0.5em}\Gamma \vdash v u \downarrow u' \implies$$

$v\text{find-match}\hspace{0.5em}c\text{s} u' = \text{Some}(\text{env, } -, \text{ rhs}) \implies$

$\Gamma' \vdash v t \downarrow v' \implies \Gamma \vdash v t \downarrow v \hspace{0.5em}|$

rec-comb:
$$\Gamma \vdash v t \downarrow V\text{rec-abs}\hspace{0.5em}c\text{s}s\hspace{0.5em}n \hspace{0.5em}\Gamma \vdash v u \downarrow u' \implies$$

$\Gamma' \vdash v t \downarrow v' \implies \Gamma \vdash v t \downarrow v \hspace{0.5em}|$

constr:
$$\text{name} \in C \implies \text{list-all2}(v\text{eval'} \hspace{0.5em}\Gamma) \hspace{0.5em}ts \hspace{0.5em}us \implies \Gamma \vdash v \text{name} \hspace{0.5em}$$

lemma $v\text{eval'}\text{-sabs-svarE}$:

assumes $\Gamma \vdash v \text{sabs}\hspace{0.5em}c\text{s} S\text{var}\hspace{0.5em}n \downarrow v$

obtains $u' \hspace{0.5em}\text{env}\hspace{0.5em}\text{pat}\hspace{0.5em}\text{rhs}$

where $\text{fnlookup}\hspace{0.5em}\Gamma \hspace{0.5em}n = \text{Some} u'$

$v\text{find-match}\hspace{0.5em}c\text{s} u' = \text{Some}(\text{env, } -, \text{ rhs}) \implies$

$\Gamma' \vdash v t \downarrow v' \hspace{0.5em}|$

using $\text{assms} \hspace{0.5em}\text{proof} \hspace{0.5em}\text{cases}$

case $(\text{constr}\hspace{0.5em}\text{name}\hspace{0.5em}ts)$

hence $\text{strip-comb}(v\text{sabs}\hspace{0.5em}c\text{s} S\text{var}\hspace{0.5em}n) = \text{strip-comb}(\text{name} \hspace{0.5em}$$

by simp

hence False

apply $(\text{fold}\hspace{0.5em}\text{app-sterm-def})$

apply $(\text{simp add:}\hspace{0.5em}\text{strip-list-comb-const})$

apply $(\text{simp add:}\hspace{0.5em}\text{const-sterm-def})$

done

thus $?\text{thesis} \hspace{0.5em}by\hspace{0.5em} \text{simp}$

next
case $\text{rec-comb}$

hence False by cases

thus $?\text{thesis} \hspace{0.5em}by\hspace{0.5em} \text{simp}$

next
case $(\text{comb}\hspace{0.5em}c\text{s}' \hspace{0.5em}\Gamma' \hspace{0.5em}u' \hspace{0.5em}\text{env}\hspace{0.5em}\text{pat}\hspace{0.5em}\text{rhs})$

moreover have $\text{fnlookup}\hspace{0.5em}\Gamma \hspace{0.5em}n = \text{Some} u'$

using $\Gamma \vdash v \text{Svar}\hspace{0.5em}n \downarrow u'$

proof cases

case $(\text{constr}\hspace{0.5em}\text{name}\hspace{0.5em}ts)$

hence False

by $(\text{fold}\hspace{0.5em}\text{free-sterm-def}) \hspace{0.5em}\text{simp}$

thus $?\text{thesis} \hspace{0.5em}by\hspace{0.5em} \text{simp}$

qed auto

moreover have $c\text{s} = c\text{s}' \hspace{0.5em}\Gamma = \Gamma'$

using $\Gamma \vdash v \text{sabs}\hspace{0.5em}c\text{s} \downarrow V\text{sabs}\hspace{0.5em}c\text{s}' \hspace{0.5em}\Gamma'$

by $(\text{cases;}\hspace{0.5em}auto)+$
ultimately show thesis
using that by auto

qed

lemma vwell'-wellformed:
assumes Γ ⊢_v t ↓ v wellformed t wellformed-venv Γ
shows vwellformed v
using assms proof induction
  case comb
  show ?case
    apply (rule comb)
    using comb by (auto simp: list-all-iff dest: vfind-match-elem intro: vwellformed.venv)
next
case (rec-comb Γ t css name Γ' cs u u' env pat rhs val)
have (pat, rhs) ∈ set cs
  by (rule vfind-match-elem) fact
show ?case
  proof (rule rec-comb)
    show wellformed-venv (Γ' ++_f mk-rec-env css Γ' ++_f env)
      proof (intro fnmpred-add)
        show wellformed-venv Γ'
          using rec-comb by auto
      next
    show wellformed-venv env
      using rec-comb by (auto dest: vfind-match-elem intro: vwellformed.venv)
next
    show wellformed-venv (mk-rec-env css Γ')
      unfolding mk-rec-env-def
      using rec-comb by (auto intro: fmdomI)
  qed
next
  have vwellformed (Vrecabs css name Γ')
    unfolding mk-rec-env-def
    using rec-comb by (auto intro: fmdom'I)
  thus wellformed rhs
    using ⟨(pat, rhs) ∈ set cs⟩ rec-comb by (auto simp: list-all-iff)
  qed
next
case (constr name Γ ts us)
  have list-all vwellformed us
    using ⟨list-all2 - - -⟩ ⟨wellformed (- $S$ -)⟩
    proof (induction ts us rule: list.rel-induct)
      case (Cons v vs u us)
      thus ?case
        using constr by (auto simp: app-sterm-def wellformed.list-comb)
    qed simp
  thus ?case
by (simp add: list-all-iff)

qed auto

lemma (in constants) veval′-shadows:
  assumes Γ ⊢ v t \downarrow v not-shadows-vconsts-env Γ ∼ shadows-consts t
  shows not-shadows-vconsts v
  using assms proof induction
  case comb
  show ?case
    apply (rule comb)
    using comb by (auto simp: list-all-iff dest: vfind-match-elem intro: not-shadows-vconsts.vmatch-env)
  next
  case (rec-comb Γ t css name Γ′ cs u u′ env pat rhs val)
  have (pat, rhs) ∈ set cs
    by (rule vfind-match-elem) fact
  show ?case
    proof (rule rec-comb)
      show not-shadows-vconsts-env (Γ′ ++ f mk-rec-env css Γ′ ++ f env)
        proof (intro fnpred-add)
      show not-shadows-vconsts-env env
        using rec-comb by (auto dest: vfind-match-elem intro: not-shadows-vconsts.vmatch-env)
      next
      show not-shadows-vconsts-env (mk-rec-env css Γ′)
        unfolding mk-rec-env_def
      using rec-comb by (auto intro: fmdomI)
      next
      show not-shadows-vconsts-env Γ′
        using rec-comb by auto
      qed
      next
      have not-shadows-vconsts (Vrecabs css name Γ′)
        using rec-comb by auto
      thus ∼ shadows-consts rhs
      using ⟨(pat, rhs) ∈ set cs⟩ rec-comb by (auto simp: list-all-iff)
      qed
      next
      case (constr name Γ ts us)
      have list-all (not-shadows-vconsts) us
        using (list-all2 - - -) (∼ shadows-consts (name $$ ts))
        proof (induction ts us rule: list.rel-induct)
        case (Cons v vs u us)
        thus ?case
          using constr by (auto simp: shadows.list-comb app-sterm-def)
        qed simp
        thus ?case
          by (simp add: list-all-iff)
      qed (auto simp: list-all-iff list-ex-iff)

lemma veval′-closed:
assumes $\Gamma \vdash_v t \downarrow v$ closed-except $t$ ($\text{fmdom } \Gamma$) closed-venv $\Gamma$
assumes wellformed $t$ wellformed-venv $\Gamma$
shows $\text{vclosed } v$
using assms proof induction
  case $(\text{comb } \Gamma' t \text{ cs } \Gamma' u' \text{ env } \text{rhs } \text{val})$
hence $\text{vclosed } (\text{Vabs cs } \Gamma')$
    by (auto simp: closed-except-def)
  have $(\text{pat}, \text{rhs}) \in \text{set } \text{cs} \text{ vmatch (mk-pat } \text{pat} \text{)} u' = \text{Some env}$
    by (rule vfind-match-elem; fact)+
hence $\text{fmdom env} = \text{patvars (mk-pat } \text{pat})$
    by (simp add: vmatch-dom)
  have $\text{vwellformed } (\text{Vabs cs } \Gamma')$
    apply (rule veval'-wellformed)
    using comb by auto
  hence linear $\text{pat}$
    using $(\text{pat}, \text{rhs}) \in \text{set } \text{cs}$
    by (auto simp: list-all-iff)
  hence $\text{fmdom env} = \text{frees pat}$
    unfolding $\text{fmdom env} = -$)
    by (simp add: mk-pat-frees)
  show $\text{case}$
    proof (rule comb)
      show wellformed $\text{rhs}$
        using $(\text{pat}, \text{rhs}) \in \text{set } \text{cs}$
        (vwellformed $(\text{Vabs cs } \Gamma')$
          by (auto simp: list-all-iff)
      next
      show $\text{closed-venv } (\Gamma' ++_f \text{ env})$
        apply rule
        using $\text{vclosed } (\text{Vabs cs } \Gamma')$
          by (auto simp: closed-except-def)
      next
      show wellformed-venv $(\Gamma' ++_f \text{ env})$
        apply rule
        using vwellformed $(\text{Vabs cs } \Gamma')$
          by (auto simp: vwellformed_vmatch-env)
        apply (rule vfind-match-elem)
        using comb by (auto)
      qed

next

\textbf{case} \((\text{rec-comb} \, \Gamma \, t \, \text{css} \, \text{name} \, \Gamma' \, \text{cs} \, \text{u} \, \text{u}' \, \text{env} \, \text{rhs} \, \text{val})\)

\textbf{have} \((\text{pat}, \, \text{rhs}) \in \text{set} \, \text{cs} \, \text{vmatch} \, \text{(mk-pat} \, \text{pat}) \, \text{u}' = \text{Some} \, \text{env}\)

\textbf{by} \((\text{rule vfind-match-elem}; \, \text{fact}+)\)

\textbf{hence} \(\text{fmdom} \, \text{env} = \text{patvars} \, \text{(mk-pat} \, \text{pat})\)

\textbf{by} \((\text{simp add}: \, \text{vmatch-dom})\)

\textbf{have} \(\text{vwellformed} \, (\text{Vrecabs} \, \text{css} \, \text{name} \, \Gamma')\)

\textbf{apply} \((\text{rule veval'}\text{-wellformed})\)

\textbf{using} \(\text{rec-comb by auto}\)

\textbf{hence} \(\text{wellformed-clauses} \, \text{cs}\)

\textbf{using} \(\text{rec-comb by auto}\)

\textbf{hence} \(\text{linear} \, \text{pat}\)

\textbf{using} \((\text{pat}, \, \text{rhs}) \in \text{set} \, \text{cs}\)

\textbf{by} \((\text{auto simp}: \, \text{list-all-iff})\)

\textbf{hence} \(\text{fmdom} \, \text{env} = \text{frees} \, \text{pat}\)

\textbf{unfolding} \(\text{fmdom} \, \text{env} = \neg\)

\textbf{by} \((\text{simp add}: \, \text{mk-pat-frees})\)

\textbf{show} \(?\text{case}\)

\textbf{proof} \((\text{rule rec-comb})\)

\textbf{show} \(\text{closed-venv} \, (\Gamma'++f \, \text{mk-rec-env} \, \text{css} \, \Gamma'++f \, \text{env})\)

\textbf{proof} \((\text{intro fnmpred-add})\)

\textbf{show} \(\text{closed-venv} \, \Gamma'\)

\textbf{using} \(\text{rec-comb by (auto simp: closed-except-def)}\)

\textbf{next}\n
\textbf{show} \(\text{closed-venv} \, \text{env}\)

\textbf{using} \(\text{rec-comb by (auto simp: closed-except-def dest: vfind-match-elem intro: vclosed.vmatch-env)}\)

\textbf{next}\n
\textbf{show} \(\text{closed-venv} \, \text{(mk-rec-env} \, \text{css} \, \Gamma')\)

\textbf{unfolding} \(\text{mk-rec-env-def}\)

\textbf{using} \(\text{rec-comb by (auto simp: closed-except-def intro: fmdomI)}\)

\textbf{qed}\n
\textbf{next}\n
\textbf{have} \(\text{vclosed} \, (\text{Vrecabs} \, \text{css} \, \text{name} \, \Gamma')\)

\textbf{using} \(\text{mk-rec-env-def}\)

\textbf{using} \(\text{rec-comb by (auto simp: closed-except-def intro: fmdomI)}\)

\textbf{hence} \(\text{closed-except} \, \text{rhs} \, (\text{fmdom} \, \text{Gamma'} \, | \cup | \, \text{frees} \, \text{pat})\)

\textbf{apply} \(\text{simp}\)

\textbf{apply} \((\text{elim conjE})\)

\textbf{apply} \((\text{drule fnmpredD[where} \, \text{m} = \text{css}]\))

\textbf{apply} \((\text{rule rec-comb})\)

\textbf{using} \((\text{pat}, \, \text{rhs}) \in \text{set} \, \text{cs}\)

\textbf{unfolding} \(\text{list-all-iff} \, \text{by auto}\)

\textbf{thus} \(\text{closed-except} \, \text{rhs} \, (\text{fmdom} \, (\Gamma'++f \, \text{mk-rec-env} \, \text{css} \, \Gamma'++f \, \text{env}))\)

\textbf{unfolding} \(\text{closed-except-def}\)

\textbf{using} \((\text{fmdom} \, \text{env} = \text{frees} \, \text{pat})\)

\textbf{by} \(\text{auto}\)
next
  show wellformed rhs
  using \langle wellformed-clauses cs \rangle \langle pat, rhs \rangle \in set cs
  by (auto simp: list-all-iff)
next
  show wellformed-venv (\Gamma' ++ f mk-rec-env css \Gamma' ++ f env)
  proof (intro fnpred-add)
    show wellformed-venv \Gamma'
    using \langle wellformed (Vrecabs css name \Gamma') \rangle by auto
next
  show wellformed-venv env
  using rec-comb by (auto dest: vfind-match-elem intro: veval'-wellformed
vwellformed.venv-env)
next
  show wellformed-venv (mk-rec-env css \Gamma' ++ f \Gamma)
  unfolding mk-rec-env-def
  using \langle wellformed (Vrecabs css name \Gamma') \rangle by (auto intro: fmdomI)
qed

next
  case (constr name \Gamma ts us)
  have list-all vclosed us
    using \langle list-all2 - - - \rangle \langle closed-except (- $$ - $$) \rangle \langle wellformed (- $$ $$) \rangle
    proof (induction ts us rule: list.rel-induct)
      case (Cons v vs u us)
      with constr show ?case
      unfolding closed.list-comb wellformed.list-comb
      by (auto simp: Sterm.closed-except-simps)
      qed simp
    thus ?case
      by (simp add: list-all-iff)
  qed (auto simp: Sterm.closed-except-simps)

primrec vwelldefined' :: value \Rightarrow bool where
vwelldefined' (Vconstr name vs) \longleftrightarrow list-all vwelldefined' vs |
vwelldefined' (Vabs cs \Gamma) \longleftrightarrow
  pred-fmap id (fmmap vwelldefined' \Gamma) \land
  list-all (\lambda(pat, t). consts t |\subseteq| (fmdom \Gamma |\cup| C)) cs \land
  fdisjnt C (fmdom \Gamma) |
vwelldefined' (Vrecabs css name \Gamma) \longleftrightarrow
  pred-fmap id (fmmap vwelldefined' \Gamma) \land
  pred-fmap (\lambda cs.
    list-all (\lambda(pat, t). consts t |\subseteq| (fmdom \Gamma |\cup| C |\cup| fmdom css)) cs \land
    fdisjnt C (fmdom \Gamma)) css \land
  name |\in| fmdom css \land
  fdisjnt C (fmdom css)

lemma vmatch-welldefined':
  assumes vmatch pat v = Some env vwelldefined' v
shows \( \text{fmpred} \ (\lambda- \text{vwelldefined}') \) \( \text{env} \)
using \( \text{assms} \) proof (induction \( \text{pat} \ v \) arbitrary; \( \text{env} \) rule: \text{vmatch-induct})
case (constr \( \text{name} \) \( \text{ps} \) \( \text{name}' \) \( \text{vs} \))
hence  
\( \text{map-option} \ (\text{foldl} \ ((++ f)) \text{fmempty}) \ (\text{those} \ (\text{map2} \text{vmatch} \text{ps} \text{vs})) = \text{Some} \ \text{env} \)  
name = name' length ps = length vs  
by (auto split: if-splits)
then obtain \( \text{envs} \) where \( \text{env} = \text{foldl} \ ((++ f)) \text{fmempty} \ \text{envs} \)  
map2 \text{vmatch} \text{ps} \text{vs} = \text{map} \text{Some} \ \text{envs}  
by (blast dest: those-someD)
moreover have \( \text{fmpred} \ (\lambda- \text{vwelldefined}') \) \( \text{env} \) if \( \text{env} \in \text{set} \ \text{envs} \) for \( \text{env} \)  
proof --  
from that have \( \text{Some} \ \text{env} \in \text{set} \ (\text{map2} \text{vmatch} \text{ps} \text{vs}) \)  
unfolding \( \text{map2} \ - - - \ = - \) by simp  
then obtain \( p \ v \) where \( p \in \text{set} \ \text{ps} \ v \in \text{set} \ \text{vs} \ \text{vmatch} \ p \ v = \text{Some} \ \text{env} \)  
by (auto elim: \text{map2-elemE})  
hence \( \text{vwelldefined}' \) \( v \)  
using constr by (simp add: list-all-iff)  
show \( \text{thesis} \)  
by (rule constr; safe?) fact+  
qed
ultimately show \( \text{?case} \)  
by auto
qed auto

lemma \( \text{sconsts-list-comb} \):  
\begin{align*} 
\text{consts} \ (\text{list-comb} \ f \ \text{xs}) \ |\subseteq| \ S & \implies \text{consts} \ f \ |\subseteq| \ S \land \text{list-all} \ (\lambda x. \ \text{consts} \ x \ |\subseteq| \ S) \ 
\text{xs} 
\end{align*}
by (induction \( \text{xs} \) arbitrary: \( f \)) auto

lemma \( \text{sconsts-sabs} \):  
\begin{align*} 
\text{consts} \ (\text{Sabs} \ cs) \ |\subseteq| \ S & \implies \text{list-all} \ (\lambda (\cdot, \ t). \ \text{consts} \ t \ |\subseteq| \ S) \ cs 
\end{align*}
apply (auto simp: list-all-iff fUnion-alt-def dest: fUnion-least-rev)
apply (auto simp add: list-all-iff fset-of-list-elem)
done

lemma (in \( \text{constants} \)) \( \text{veval}' \)-welldefined':  
assumes \( \Gamma \vdash_v t \downarrow \) \( v \) \text{fdisjnt} \( C \) (fdom \( \Gamma \))  
assumes \( \text{consts} \ t \ |\subseteq| \ \text{fdom} \ \Gamma \ |\cup| \ C \ \text{fpred} \ (\lambda-. \ \text{vwelldefined}') \) \( \Gamma \)  
assumes \( \text{wellformed} \ t \ \text{wellformed-venv} \ \Gamma \)  
assumes \( \neg \text{shadows-consts} \ t \ \text{not-shadows-vconsts-env} \ \Gamma \)  
shows \( \text{vwelldefined}' \) \( v \)  
using \( \text{assms} \) proof induction  
case (abs \( \Gamma \ cs \))  
thus \( \text{?case} \)

186
unfolding sconsts-sabs
   by (auto simp: list-all-iff list-ex-iff)

next
case (comb Γ t cs Γ′ u u′ env pat rhs val)
  hence (pat, rhs) ∈ set cs
     by (auto dest: vfind-match-elem)
moreover have vwelldefined′ (Vabs cs Γ′)
   using comb by auto
ultimately have consts rhs |⊆| fmdom Γ′ |∪| C
     by (auto simp: list-all-iff)

have vwellformed (Vabs cs Γ′)
  apply (rule veval′-wellformed)
  using comb by auto
hence linear pat
  using ⟨(pat, rhs) ∈ set cs⟩
  by (auto simp: list-all-iff)
hence frees pat = patvars (mk-pat pat)
  by (simp add: mk-pat-frees)
  hence fdom env = frees pat
apply simp
apply (rule vmatch-dom)
apply (rule vfind-match-elem)
apply (rule comb)
done

have not-shadows-vconsts (Vabs cs Γ′)
  apply (rule veval′-shadows)
  using comb by auto

have vwelldefined′ (Vabs cs Γ′)
  using comb by auto

show ?case
proof (rule comb)
  show consts rhs |⊆| fmdom (Γ′ ++ f env) |∪| C
    using ⟨consts rhs |⊆| fmdom Γ′ |∪| C⟩
    by auto
next
  have vwelldefined′ u u′
    using comb by auto
  hence fmpred (λ-. vwelldefined′) env
    using comb
    by (auto intro: vmatch-welldefined′ dest: vfind-match-elem)
  thus fmpred (λ-. vwelldefined′) (Γ′ ++ f env)
    using ⟨vwelldefined′ (Vabs cs Γ′)⟩ by auto
next
  have fdisjnt C (fmdom Γ′)
    using ⟨vwelldefined′ (Vabs cs Γ′)⟩
by simp
moreover have \( \text{fdisjnt } C \ (\text{fdom env}) \)

unfolding \( \text{fdom env} = \text{frees pat} \)
using \( (\text{pat}, \text{rhs}) \in \text{set cs} \) \( \langle \text{not-shadows-vconsts} \ (Vabs cs \Gamma') \rangle \)
by (auto simp: list-all-iff all-consts-def fdisjnt-alt-def)
ultimately show \( \text{fdisjnt } C \ (\text{fdom} \ (\Gamma' +\!+ f \text{ env})) \)

next
show wellformed rhs
using \( (\text{pat}, \text{rhs}) \in \text{set cs} \) \( \langle \text{vwellformed} \ (Vabs cs \Gamma') \rangle \)
by (auto simp: list-all-iff)

next
have wellformed-venv \( \Gamma' \)
using \( \langle \text{vwellformed} \ (Vabs cs \Gamma') \rangle \)
by simp

moreover have wellformed-venv env
apply (rule vwellformed.vmatch-env)
apply (rule vfind-match-elem)
apply fact
apply (rule veval’.wellformed)
using comb by auto
ultimately show wellformed-venv \( (\Gamma' +\!+ f \text{ env}) \)
by blast

next
have not-shadows-vconsts-env \( \Gamma' \)
using \( \langle \text{not-shadows-vconsts} \ (Vabs cs \Gamma') \rangle \)
by simp
moreover have not-shadows-vconsts-env env
apply (rule not-shadows-vconsts.vmatch-env)
apply (rule vfind-match-elem)
apply fact
apply (rule veval’.shadows)
using comb by auto
ultimately show not-shadows-vconsts-env \( (\Gamma' +\!+ f \text{ env}) \)
by blast

next
show \( \neg \text{shadows-consts } \text{rhs} \)
using \( \langle \text{not-shadows-vconsts} \ (Vabs cs \Gamma') \rangle \) \( (\text{pat}, \text{rhs}) \in \text{set cs} \)
by (auto simp: list-all-iff)

next

case (rec-comb \( \Gamma \ t \ css \ name \ \Gamma' \ cs \ u \ u' \ env \ pat \ rhs \ val \))
hence \( (\text{pat}, \text{rhs}) \in \text{set cs} \)
by (auto dest: vfind-match-elem)

moreover have vwelldefined’ \( (Vrecabs css \ name \ \Gamma') \)
using rec-comb by auto
ultimately have \( \text{consts } \text{rhs} \ \subseteq \ \text{fdom} \ \Gamma' \ |\cup| \ (C \ |\cup| \ \text{fdom } css) \)
using \( \text{fmlookup } css \ name = \text{Some } cs \)
by (auto simp: list-all-iff dest!: fmpredD[where \( m = \text{css} \])

have vwellformed \( (Vrecabs css \ name \ \Gamma') \)
apply (rule veval′-wellformed)
using rec-comb by auto
hence wellformed-clauses cs
using rec-comb by auto
hence linear pat
using \((\text{pat}, \text{rhs}) \in \text{set cs}\)
by (auto simp: list-all-iff)
hence frees pat = patvars (mk-pat pat)
by (simp add: mk-pat-frees)
hence fmdom env = frees pat
apply simp
apply (rule vmatch-dom)
apply (rule vfind-match-elem)
apply (rule rec-comb)
done

have not-shadows-vconsts (Vrecabs css name \(\Gamma′\))
apply (rule veval′-shadows)
using rec-comb by auto
have vwelldefined′ (Vrecabs css name \(\Gamma′\))
using rec-comb by auto

show ?case
proof (rule rec-comb)
  show \(\text{consts rhs} \subseteq \text{fmdom}\ (\Gamma′ + + f \text{mk-rec-env css} \Gamma′ + + f \text{env}) \cup C\)
  using \(\text{consts rhs} \subseteq \cdot\) unfolding mk-rec-env-def
  by auto
next
have fnmpred (\(\lambda\cdot. \text{vwelldefined′}\)) \(\Gamma′\)
  using (\text{vwelldefined′} (Vrecabs css name \(\Gamma′\))) by auto
moreover have fnmpred (\(\lambda\cdot. \text{vwelldefined′}\)) (mk-rec-env css \(\Gamma′\))
  unfolding mk-rec-env-def
  using rec-comb by (auto intro: fmdomI)
moreover have fnmpred (\(\lambda\cdot. \text{vwelldefined′}\)) env
  using rec-comb by (auto dest: vfind-match-elem intro: vmatch-welldefined′)
ultimately show fnmpred (\(\lambda\cdot. \text{vwelldefined′}\)) (\(\Gamma′ + + f \text{mk-rec-env css} \Gamma′ + + f \text{env}\))
  by blast
next
have fdisjnt C (fmdom \(\Gamma′\))
  using rec-comb by auto
moreover have fdisjnt C (fmdom env)
  unfolding (fmdom env = frees pat)
  using \(\text{fmlookup css name = Some cs}\) \((\text{pat}, \text{rhs}) \in \text{set cs}\) (not-shadows-vconsts (Vrecabs css name \(\Gamma′\))
  apply auto
  apply (drule fnmpredD[where \(m = \text{css}\)])
  by (auto simp: list-all-iff allconsts-def fdisjnt-alt-def)
moreover have \( f_{\text{disjnt}} C (f_{\text{mdom}} (mk_{\text{-}env} css \Gamma')) \)

unfolding \( mk_{\text{-}rec}_{\text{-}env}_{\text{-}def} \)
using \( \langle \text{welldefined}' (V_{\text{recabs}} css \text{ name } \Gamma') \rangle \)
by simp

ultimately show \( f_{\text{disjnt}} C (f_{\text{mdom}} (\Gamma' + +f mk_{\text{-}rec}_{\text{-}env} css \Gamma' + +f \text{ env})) \)

unfolding \( f_{\text{disjnt-alt}_{\text{-}def}} \) by auto

next

show wellformed rhs
using \( \langle \text{pat}, \text{rhs} \rangle \in \text{set cs} \langle \text{wellformed-clauses cs} \rangle \)
by (auto simp: list-all-iff)

next

have wellformed-env \( \Gamma' \)
using \( \langle \text{wellformed} (V_{\text{recabs}} css \text{ name } \Gamma') \rangle \) by simp

moreover have wellformed-env \( (mk_{\text{-}rec}_{\text{-}env} css \Gamma') \)

unfolding \( mk_{\text{-}rec}_{\text{-}env}_{\text{-}def} \)
using \( \langle \text{wellformed} (V_{\text{recabs}} css \text{ name } \Gamma') \rangle \)
by (auto intro: fmdomI)

moreover have wellformed-env env

apply (rule vwellformed, vmatch-env)
apply (rule vfind-match-elem)
apply fact
apply (rule venv', wellformed)

using rec-comb by auto

ultimately show wellformed-env \( (\Gamma' + +f mk_{\text{-}rec}_{\text{-}env} css \Gamma' + +f \text{ env}) \)
by blast

next

have not-shadows-\text{-}vconsts\text{-}env \( \Gamma' \)
using \( \langle \text{not-shadows-\text{-}vconsts} (V_{\text{recabs}} css \text{ name } \Gamma') \rangle \) by simp

moreover have not-shadows-\text{-}vconsts\text{-}env \( (mk_{\text{-}rec}_{\text{-}env} css \Gamma') \)

unfolding \( mk_{\text{-}rec}_{\text{-}env}_{\text{-}def} \)
using \( \langle \text{not-shadows-\text{-}vconsts} (V_{\text{recabs}} css \text{ name } \Gamma') \rangle \)
by (auto intro: fmdomI)

moreover have not-shadows-\text{-}vconsts\text{-}env env

apply (rule not-shadows-\text{-}vconsts, vmatch-env)
apply (rule vfind-match-elem)
apply fact
apply (rule venv', shadows)

using rec-comb by auto

ultimately show not-shadows-\text{-}vconsts\text{-}env \( (\Gamma' + +f mk_{\text{-}rec}_{\text{-}env} css \Gamma' + +f \text{ env}) \)
by blast

next

show \( \neg \text{shadows-consts} \text{ rhs} \)
using rec-comb \( \langle \text{not-shadows-\text{-}vconsts} (V_{\text{recabs}} css \text{ name } \Gamma') \rangle \langle \text{pat}, \text{rhs} \rangle \in \text{set cs} \)
by (auto simp: list-all-iff)

qed

next

case \( (\text{constr name } \Gamma \ ts us) \)
have list-all 'welldefined' us
  using \( \langle \text{list-all2} \ - \ - \ \rangle \ (\text{consts} \ (\text{name} \ $$ ts) \ |\subseteq| \ -) \)
  using \( \langle \text{wellformed} \ (\text{name} \ $$ ts); \ (\neg \text{shadows-consts} \ (\text{name} \ $$ ts)) \rangle \)
  proof (induction \( ts \ us \) rule: list.rel-induct)
  case (Cons \( v \ us \))
  with constr show ?case
  unfolding wellformed.list-comb shadows.list-comb
  by (auto simp: consts-list-comb)
  qed simp
  thus ?case by (simp add: list-all-iff)
  qed auto
end

Correctness wrt veval
context vrules begin
The following relation can be characterized as follows:

- Values have to have the same structure. (We prove an interpretation of value-struct-rel.)
- For closures, the captured environments must agree on constants and variables occurring in the body. The first value argument is from veval (i.e. from CakeML-Codegen.Big-Step-Value), the second from veval'.

coinductive vrelated :: value ⇒ value ⇒ bool (\( \vdash v/ - \approx [-0, 50] 50 \)) where
constr: list-all2 vrelated \( ts \ us \) \( \Rightarrow \vdash v \ \text{Vconstr name ts} \approx V\text{constr name us} \ |
abs:
  \text{fmrel-on-fset} \ (\text{frees} (\text{Sabs cs})) \ vrelated \( \Gamma_1 \ \Gamma_2 \Rightarrow \vdash v \ \text{Vabs cs} \ \Gamma_1 \approx V\text{abs cs} \ \Gamma_2 \ |
rec-abs:
  \text{pred-fmap} \ (\lambda cs.
    \text{fmrel-on-fset} \ (\text{frees} (\text{Sabs cs})) \ vrelated \ (\text{fmap-of-list rs}) \ \Gamma_2 \Rightarrow
    \vdash v \ \text{Vrecabs css name} \ \Gamma_1 \approx V\text{recabs css name} \ \Gamma_2 \)

Perhaps unexpectedly, vrelated is not reflexive. The reason is that it does not just check syntactic equality including captured environments, but also adherence to the external rules.

sublocale vrelated: value-struct-rel vrelated
proof
  fix \( t_1 \ t_2\)
assume $\vdash v \approx t_1 \Rightarrow t_2$

thus $veq-structure t_1 t_2$

apply (induction $t_1$ arbitrary: $t_2$)
  apply (erule vrelated.cases; auto)
  apply (erule list.rel-mono-strong)
  apply simp
  apply (erule vrelated.cases; auto)
  apply (erule vrelated.cases; auto)
  done

next

fix name name’ and $ts us :: value list$

show $\vdash v Vconstr name ts \approx Vconstr name’ us \iff (name = name’ \land list-all2 vrelated ts us)$

proof safe
  assume $\vdash v Vconstr name ts \approx Vconstr name’ us$
  thus $name = name’ \land list-all2 vrelated ts us$
  by (cases; auto)+

qed (auto intro: vrelated.intros)

qed

The technically involved relation $vrelated$ implies a weaker, but more intuitive property: If $\vdash v t \approx u$ then $t$ and $u$ are equal after termification (i.e. conversion with $value-to-sterm$). In fact, if both terms are ground terms, it collapses to equality. This follows directly from the interpretation of $value-struct-rel$.

lemma $veval’$-correct:

assumes $\Gamma_2 \vdash v t \downarrow v_2$ wellformed $t$ wellformed-venv $\Gamma_2$
assumes $\neg$ shadows-consts $t$ not-shadows-vconsts-env $\Gamma_2$
assumes welldefined $t$
assumes fnpred ($\lambda$. vwelldefined) $\Gamma_1$
assumes fnrel-on-fset (frees $t$) vrelated $\Gamma_1 \Gamma_2$
assumes fnrel-on-fset (consts $t$) vrelated $\Gamma_1 \Gamma_2$

obtains $v_1$ where $rs, \Gamma_1 \vdash v t \downarrow v_1 \vdash v_1 v_2 \approx v_2$

apply atomize-elim

using assms proof (induction arbitrary: $\Gamma_1$)

case (const name $\Gamma_2 \vdash val_2$)

hence fnrel-on-fset $\{|name|\}$ vrelated (fnmap-of-list $rs$) $\Gamma_2$

by simp

have rel-option vrelated (fnlookup (fnmap-of-list $rs$) $name$) (fnlookup $\Gamma_2$ $name$)

apply (rule fnrel-on-fsetD[where $S = \{|name|\}$])

apply simp

apply fact

done

then obtain $val_1$ where fnlookup (fnmap-of-list $rs$) $name = Some val_1 \vdash v val_1$

$\approx val_2$

using const by cases auto

hence $(name, val_1) \in set rs$

by (auto dest: fnmap-of-list-SomeD)
show ?case
  apply (intro conjI exI)
  apply (rule veval.const)
  apply fact+
done

next
case (var Γ₂ name val₂)
hence fmrel-on-fset \{\{name\}\} vrelated Γ₁ Γ₂
  by simp
have rel-option vrelated \(\text{fmlookup } Γ₁ \text{ name}) \(\text{fmlookup } Γ₂ \text{ name})
  apply (rule \text{fmrel-on-fsetD where} \(S = \{\{\text{name}\}\}\))
  apply simp
  apply fact
  done
then obtain \(val₁ \text{ where } \text{fmlookup } Γ₁ \text{ name} = \text{Some } val₁ \equiv val₂\)
  using var by cases auto
show ?case
  apply (intro conjI exI)
  apply (rule veval.var)
  apply fact+
  done

next
case abs
thus ?case
  by (auto intro: veval.abs vrelated.abs)

next
case (comb Γ₂ t cs Γ₂' u u₂' env₂ pat rhs val₂)
hence \(∃ v. rs, Γ₁ ⊢ v \downarrow v \land v ≈ \text{Vabs } cs Γ₂'\)
  by (auto intro: fmrel-on-fsubset)
then obtain \(v \text{ where } Γ₁ ⊢ v \downarrow v ≈ \text{Vabs } cs Γ₂' \text{ rs, } Γ₁ ⊢ v \downarrow v\)
  by blast
moreover then obtain Γ₁'
  where \(v = \text{Vabs } cs Γ₁'\)
    and \(\text{fmrel-on-fset (frees (Sabs cs)) vrelated } Γ₁' Γ₂\)
    and \(\text{fmrel-on-fset (consts (Sabs cs)) vrelated (fmap-of-list rs) } Γ₂'\)
  by cases auto
ultimately have \(rs, Γ₁ ⊢ t \downarrow \text{Vabs } cs Γ₁'\)
  by simp
have \(∃ u₁', rs, Γ₁ ⊢ u \downarrow u₁' \land Γ₁ ⊢ u₁' ≈ u₂'\)
  using comb by (auto intro: fmrel-on-fsubset)
then obtain \(u₁' \text{ where } Γ₁ ⊢ u₁' ≈ u₂' \text{ rs, } Γ₁ ⊢ u \downarrow u₁'\)
  by blast

have rel-option (rel-prod (fmrel vrelated) (=)) \(\text{vfind-match cs } u₁'\) \(\text{vfind-match cs } u₂'\)
  by (rule vrelated.vfind-match-rel') fact
then obtain \(env₁ \text{ where } \text{vfind-match cs } u₁' = \text{Some } (env₁, pat, rhs) \text{ fmrel vrelated env₁ env₂}\)
using \langle v\text{find-match} \ cs \ u' \_2 = \_ \rangle 
by \ cases \ auto 

have \ (\text{pat}, \text{rhs}) \in \set \ cs 
by \ (\text{rule } \text{vfind-match-elem}) \ fact 

have \ v\text{wellformed} \ (V\text{abs} \ cs \ \Gamma' \_2) 
apply \ (\text{rule } v\text{eval}'-\text{wellformed}) 
apply \ fact 
using \ (\text{wellformed} \ (t \$_a \ u)) \ apply \ \text{simp} 
apply \ fact+ 
done 
hence \ \text{wellformed-env} \ \Gamma' \_2 
by \ \text{simp} 

have \ v\text{welldefined} \ v 
apply \ (\text{rule } v\text{eval-welldefined}) 
apply \ fact+ 
using \ \text{comb} \ by \ \text{simp} 
hence \ v\text{welldefined} \ (V\text{abs} \ cs \ \Gamma' \_1) 
unfolding \ (v = \_). 

have \ \text{linear} \ \text{pat} 
using \ \langle (\text{pat}, \text{rhs}) \in \set \ cs \ \rangle \ \langle \text{vwellformed} \ (V\text{abs} \ cs \ \Gamma' \_2) \rangle 
by \ (auto \ \text{simp}: \ \text{list-all-iff}) 

have \ f\text{mdom} \ env_1 = \text{patvars} \ (mk \text{-pat} \ \text{pat}) 
apply \ (\text{rule } v\text{match-dom}) 
apply \ (\text{rule } \text{vfind-match-elem}) 
apply \ fact 
done 
with \ \langle \text{linear} \ \text{pat} \rangle \ have \ f\text{mdom} \ env_1 = \text{frees} \ \text{pat} 
by \ (\text{simp add}: \ \text{mk-pat-frees}) 

have \ f\text{mdom} \ env_2 = \text{patvars} \ (mk \text{-pat} \ \text{pat}) 
apply \ (\text{rule } v\text{match-dom}) 
apply \ (\text{rule } \text{vfind-match-elem}) 
apply \ fact 
done 
with \ \langle \text{linear} \ \text{pat} \rangle \ have \ f\text{mdom} \ env_2 = \text{frees} \ \text{pat} 
by \ (\text{simp add}: \ \text{mk-pat-frees}) 

note \ f\text{set-of-list-map}[\text{simp del}] 
have \ \exists \ \text{val}_1. \ \text{rs}, \ \Gamma'_1 ++f \ \text{env}_1 \vdash_v \ \text{rhs} \ \downarrow \ \text{val}_1 \ \land \ \vdash_v \ \text{val}_1 \ \approx \ \text{val}_2 
proof \ (\text{rule } \text{comb}) 
show \ f\text{rel-on-fset} \ (\text{frees} \ \text{rhs}) \ \text{vrelated} \ (\Gamma'_1 ++f \ \text{env}_1) \ (\Gamma'_2 ++f \ \text{env}_2) 
proof 
fix \ \text{name} 
assume \ \text{name} \in \ \text{frees} \ \text{rhs} 

194
show rel-option vrelated (fmlookup (Γ’₁ ++f env₁) name) (fmlookup (Γ’₂ ++f env₂) name)
proof (cases name |∈| frees pat)
  case True
  hence name |∈| fmdom env₁ name |∈| fmdom env₂
    using (fmdom env₁ = frees pat) (fmdom env₂ = frees pat)
    by simp+
  hence fmlookup (Γ’₁ ++f env₁) name = fmlookup env₁ name fmlookup (Γ’₂ ++f env₂) name = fmlookup env₂ name
    by auto
  thus ?thesis
    using :fmrel vrelated env₁ env₂
    by auto
next
  case False
  hence name |∉| fmdom env₁ name |∉| fmdom env₂
    using (fmdom env₁ = frees pat) (fmdom env₂ = frees pat)
    by simp+
  hence fmlookup (Γ’₁ ++f env₁) name = fmlookup Γ’₁ name fmlookup (Γ’₂ ++f env₂) name = fmlookup Γ’₂ name
    by auto
moreover have name |∈| frees (Sabs cs)
  using False (name |∈| frees rhs) (pat, rhs) ∈ set cs
  apply (auto simp: ffUnion-alt-def)
  apply (rule fBexI[where x = frees rhs |−| frees pat!])
  apply (auto simp: fset-of-list-elem)
  done
ultimately show ?thesis
  using :fmrel-on-fset (frees (Sabs cs)) vrelated Γ’₁ Γ’₂
    by (auto dest: fmrel-on-fsetD)
qed
qed
next
have not-shadows-vconsts (Vabs cs Γ’₂)
  apply (rule veval’-shadows)
  apply fact+
  using comb by auto
thus ¬ shadows-consts rhs
  using (pat, rhs) ∈ set cs
  by (auto simp: list-all-iff)
show not-shadows-vconsts-env (Γ’₂ ++f env₂)
  apply rule
  using (not-shadows-vconsts (Vabs cs Γ’₂)); apply simp
  apply (rule not-shadows-vconsts.vmatch-env)
  apply (rule vfind-match-elem)
  apply fact
apply (rule veval'-shadows)
apply fact
apply fact
using comb by auto

show fmrel-on-fset (consts rhs) vrelated (fmap-of-list rs) (Γ′₂ ++f env₂)
proof
fix name
assume name ∈| consts rhs
hence name ∈| consts (Sabs cs)
using ((pat, rhs) ∈ set cs)
by (auto intro: fBexI simp: fset-of-list-elm fUnion-alt-def)
hence rel-option vrelated (fmlookup (fmap-of-list rs) name) (fmlookup Γ′₂ name)

using (fmrel-on-fset (consts (Sabs cs)) vrelated (fmap-of-list rs) Γ′₂)
by (auto dest: fmrel-on-fsetD)
mOREOVER have name /∈| fmdom env₂
proof
assume name ∈| fmdom env₂
hence fmlookup env₂ name ≠ None
by (meson fmdom-notI)
then obtain v where fmlookup env₂ name = Some v
by blast
hence name ∈| fmdom env₂
by (auto intro: fdomI)
hence name ∈| frees pat
using (fdom env₂ = frees pat)
by simp

have welldefined rhs
using :welldefined (Vabs cs Γ′₁) ((pat, rhs) ∈ set cs)
by (auto simp: list-all-iff)
hence name ∈| fst (| fset-of-list rs |∪| C)
using :name ∈| consts rhs)
by (auto simp: all-consts-def)
mOREOVER have ¬ shadows-consts pat
using :not-shadows-vconsts (Vabs cs Γ′₂) ((pat, rhs) ∈ set cs)
by (auto simp: list-all-iff shadows-consts-def all-consts-def)
ultimately show False
using :name ∈| frees pat

unfolding shadows-consts-def fdisjnt-alt-def all-consts-def
by auto
qed
ultimately show rel-option vrelated (fmlookup (fmap-of-list rs) name)
(fmlookup (Γ′₂ ++f env₂) name)
by simp
qed
next
show wellformed rhs
using \((\text{pat}, \text{rhs}) \in \text{set } cs\) \(\langle \text{vwellformed } (Vabs \text{ cs } \Gamma'2) \rangle\)
by \((\text{auto simp: list-all-iff})\)

next
have wellformed-venv \(\Gamma'2\)
by fact
moreover have wellformed-venv \(\text{env}_2\)
apply (rule vwellformed.vmatch-env)
apply (rule vfind-match-elem)
apply fact
apply (rule veval'-wellformed)
apply fact
using \(\langle \text{wellformed } (t \text{ $u$ } u) \rangle\) apply simp
apply fact
done
ultimately show wellformed-venv \((\Gamma'2 ++f \text{ env}_2)\)
by blast

next
show welldefined \(\text{rhs}\)
using \(\langle \text{vwelldefined } (Vabs \text{ cs } \Gamma'1) \rangle \langle (\text{pat}, \text{rhs}) \in \text{set } cs \rangle\)
by \((\text{auto simp: list-all-iff})\)

next
have fmpred \((\lambda\cdot. \text{vwelldefined}) \Gamma'1\)
using \(\langle \text{vwelldefined } (Vabs \text{ cs } \Gamma'1) \rangle\) by simp
moreover have fmpred \((\lambda\cdot. \text{vwelldefined}) \text{ env}_1\)
apply (rule vwelldefined.vmatch-env)
apply (rule vfind-match-elem)
apply fact
apply (rule veval-welldefined)
apply fact
apply fact
using comb apply simp
done
ultimately show fmpred \((\lambda\cdot. \text{vwelldefined}) \ (\Gamma'1 ++f \text{ env}_1)\)
by blast

qed

then obtain \(\text{val}_1\) where \(\text{rs}, \Gamma'1 ++f \text{ env}_1 \vdash_v \text{rhs} \downarrow \text{val}_1 \vdash_v \text{val}_1 \approx \text{val}_2\)
by blast

show ?case
apply (intro conj1 exI)
apply (rule veval.comb)
apply fact+
done
next
— Almost verbatim copy from \text{comb case}.
case (rec-comb \(\Gamma'2\) \(t\) \text{css name} \(\Gamma'2\) \text{cs } \text{u} \text{u}_2' \text{env}_2\) \text{pat} \text{rhs} \text{val}_2)
hence \( \exists v, rs, \Gamma_1 \vdash v \downarrow v \land \vdash v \approx \text{Vrecabs css name } \Gamma_2 \)
by (auto intro: fnrel-on-fsubset)
then obtain \( v \) where \( \vdash v \approx \text{Vrecabs css name } \Gamma_2 \) \( rs, \Gamma_1 \vdash t \downarrow v \)
by blast
moreover then obtain \( \Gamma_1' \)
where \( v = \text{Vrecabs css name } \Gamma_1' \)
and \( \text{fnrel-on-fset (frees (Sabs cs)) vrelated } \Gamma_1' \Gamma_2' \)
and \( \text{fnrel-on-fset (consts (Sabs cs)) vrelated (fmap-of-list rs) (} \Gamma_2' \++ f \text{mk-rec-env css } \Gamma_2' \))
using (fnlookup css name = Some cs)
by cases auto
ultimately have \( rs, \Gamma_1 \vdash t \downarrow \text{Vrecabs css name } \Gamma_1' \)
by simp
have \( \exists u_1', rs, \Gamma_1 \vdash u \downarrow u_1' \land \vdash u_1' \approx u_2' \)
using rec-comb by (auto intro: fnrel-on-fsubset)
then obtain \( u_1' \) where \( \vdash u_1' \approx u_2' \) \( rs, \Gamma_1 \vdash u \downarrow u_1' \)
by blast
have \( \text{rel-option (rel-prod (fnrel vrelated) (=)) (vfind-match cs u_1') (vfind-match cs u_2')} \)
by (rule vrelated.vfind-match-rel') fact
then obtain \( \text{env}_1 \) where \( \text{vfind-match cs u_1'} = \text{Some (env}_1, \text{pat}, \text{rhs}) \) fnrel
vrelated \( \text{env}_1 \text{ env}_2 \\
using (vfind-match cs u_2' = \_
by cases auto
have \( (\text{pat}, \text{rhs}) \in \text{set cs} \)
by (rule vfind-match-elem) fact
have \( \text{vwellformed (Vrecabs css name } \Gamma_2' \))
apply (rule veval'-wellformed)
apply fact
using (wellformed (t $ a u)) apply simp
apply fact+
done
hence \( \text{wellformed-env } \Gamma_2' \text{ vwellformed (Vabs cs } \Gamma_2') \)
using rec-comb by auto
have \( \text{vwelldefined v} \)
apply (rule veval-welldefined)
apply fact+
using rec-comb by simp
hence \( \text{vwelldefined (Vrecabs css name } \Gamma_1' \))
unfolding \( \langle v = \_ \rangle \).
hence \( \text{vwelldefined (Vabs cs } \Gamma_1') \)
using rec-comb by auto
have linear pat
using \((\text{pat, rhs}) \in \text{set cs} \) \(\text{vwellformed (Vabs cs} \ \Gamma'_2)\):
by (auto simp: list-all-iff)

have \(\text{fdom env}_1 = \text{patvars (mk-pat pat)}\)
apply (rule vmatch-dom)
apply (rule vfind-match-elem)
apply fact
done
with (linear pat) have \(\text{fdom env}_1 = \text{frees pat}\)
by (simp add: mk-pat-frees)

have \(\text{fdom env}_2 = \text{patvars (mk-pat pat)}\)
apply (rule vmatch-dom)
apply (rule vfind-match-elem)
apply fact
done
with (linear pat) have \(\text{fdom env}_2 = \text{frees pat}\)
by (simp add: mk-pat-frees)

note \(\text{fset-of-list-map[simp del]}\)
have \(\exists \text{val}_1, \text{rs}, \Gamma'_1 \ + + f \ \text{env}_1 \vdash_\text{v} \text{rhs} \downarrow \text{val}_1 \land \vdash_\text{v} \text{val}_1 \approx \text{val}_2\)
proof (rule rec-comb)
  have \(\text{not-shadows-v consts (Vrecabs css name} \ \Gamma'_2)\)
  apply (rule veval’-shadows)
  apply fact+
  using rec-comb by auto
  thus \(\neg \text{shadows-c ons ts rhs}\)
  using \((\text{pat, rhs}) \in \text{set cs} \) rec-comb
  by (auto simp: list-all-iff)
  hence \(\text{fdisjnt all-consts (frees rhs)}\)
  by (rule shadows-consts-frees)

  have \(\text{not-shadows-v consts-env} \ \Gamma'_2\)
  using \((\text{not-shadows-v consts (Vrecabs css name} \ \Gamma'_2)\)
  by simp

moreover have \(\text{not-shadows-v consts-env (mk-rec-env css} \ \Gamma'_2)\)
unfolding \(\text{mk-rec-env-def}\)
using \((\text{not-shadows-v consts (Vrecabs css name} \ \Gamma'_2)\)
by (auto intro: fdomI)

moreover have \(\text{not-shadows-v consts-env env}_2\)
apply (rule not-shadows-v consts.veeval-consts)
apply (rule vfind-match-elem)
apply fact
apply (rule veval’-shadows)
apply fact
apply fact
using rec-comb by auto

199
ultimately show not-shadows-vconsts-env (Γ′₂ ++f mk-rec-env css Γ′₂ ++f env₂)
by auto

have not-shadows-vconsts (Vabs cs Γ′₂)
using ⟨not-shadows-vconsts (Vrecabs - - -), rec-comb⟩
by auto

show fmrel-on-fset (frees rhs) vrelated (Γ′₁ ++f env₁) (Γ′₂ ++f mk-rec-env css Γ′₂ ++f env₂)
proof
fix name
assume name |∈| frees rhs
moreover have fmdom css |⊆| allconsts
using (welldefined (Vrecabs - - -): unfolding all-consts-def)
by auto
ultimately have name |∉| fmdom css
using ⟨fmdisjnt - (frees rhs): unfolding fdisjnt-alt-def⟩
by (metis (full-types) fempty-iff finterI fset-rev-mp)

show rel-option vrelated (fmlookup (Γ′₁ ++f env₁) name) (fmlookup (Γ′₂ ++f mk-rec-env css Γ′₂ ++f env₂) name)
proof (cases name |∈| frees pat)
case True
hence name |∈| fmdom env₁ name |∈| fmdom env₂
using (fmdom env₁ = frees pat) (fmdom env₂ = frees pat)
by simp+
hence
fmlookup (Γ′₁ ++f env₁) name = fmlookup env₁ name
fmlookup (Γ′₂ ++f mk-rec-env css Γ′₂ ++f env₂) name = fmlookup env₂ name
by auto
thus ?thesis
using ⟨fmrel vrelated env₁ env₂⟩
by auto
next
case False
hence name |∉| fmdom env₁ name |∉| fmdom env₂
using (fmdom env₁ = frees pat) (fmdom env₂ = frees pat)
by simp+
hence
fmlookup (Γ′₁ ++f env₁) name = fmlookup Γ′₁ name
fmlookup (Γ′₂ ++f mk-rec-env css Γ′₂ ++f env₂) name = fmlookup Γ′₂ name
unfolding mk-rec-env-def using ⟨name |∉| fmdom css⟩
by auto
moreover have \( \text{name} \in \text{frees} (Sabs \ \text{cs}) \)
using \( \text{False} (\text{name} \in \text{frees} \ \text{rhs}) \)
\((\text{pat}, \ \text{rhs}) \in \text{set} \ \text{cs}) \)
apply (auto simp: \text{ffUnion-alt-def})
apply (rule \text{fBexI}[\text{where} \ x = \text{frees} \ \text{rhs} |\text{-|frees} \ \text{pat}])
apply (auto simp: \text{fset-of-list-elem})
done

ultimately show \( \text{thesis} \)
using \( \text{fmrel-on-fset} (\text{frees} (Sabs \ \text{cs})) \ \text{vrelated} \ \Gamma_1 \ \Gamma_2 \)
by (auto dest: \text{fnrel-on-fsetD})
qed
qed

show \( \text{fnrel-on-fset} (\text{consts} \ \text{rhs}) \ \text{vrelated} \ \text{fmap-of-list} \ \text{rs} \) (\Gamma_2 \ +\ +_f \ \text{mk-rec-env} \\
\text{css} \ \Gamma_2 \ +\ +_f \ \text{env}_2)
proof
fix name
assume name \( \in \text{consts} \ \text{rhs} \)

hence name \( \in \text{consts} (Sabs \ \text{cs}) \)
using \( (\text{pat}, \ \text{rhs}) \in \text{set} \ \text{cs}) \)
by (auto intro: \text{fBexI} simp: \text{fset-of-list-elem} \text{ffUnion-alt-def})

hence rel-option \text{vrelated} (\text{fnlookup} \ \text{fmap-of-list} \ \text{rs} \ \text{name}) (\text{fnlookup} \ (\Gamma_2 \\
+\ +_f \ \text{mk-rec-env} \ \text{css} \ \Gamma_2) \ \text{name})
using \( \text{fnrel-on-fset} (\text{consts} (Sabs \ \text{cs})) \ \text{vrelated} \ \text{fmap-of-list} \ \text{rs} \) (\Gamma_2 \\
+\ +_f \ \text{mk-rec-env} \ \text{css} \ \Gamma_2)
by (auto dest: \text{fnrel-on-fsetD})

moreover have name \( \in \text{fmdom} \ \text{env}_2 \)

proof
assume name \( \in \text{fmdom} \ \text{env}_2 \)

hence \text{fnlookup} \ \text{env}_2 \ \text{name} \neq \text{None}
by (meson \text{fmdom-notI})

then obtain \( v \) \text{where} \text{fnlookup} \ \text{env}_2 \ \text{name} = \text{Some} \ v
by blast

hence name \( \in \text{fmdom} \ \text{env}_2 \)
by (auto intro: \text{fmdomI})

hence name \( \in \text{frees} \ \text{pat} \)
using \( \text{fnmdom} \ \text{env}_2 = \text{frees} \ \text{pat} \)
by simp

have \( \text{vwelldefined} (Vabs \ \text{cs} \ \Gamma_1) \)
using \( \text{vwelldefined} (Vrecabs \ \text{css} - \Gamma_1) \)
using \text{rec-comb}
by auto

hence \( \text{welldefined} \ \text{rhs} \)
using \( (\text{pat}, \ \text{rhs}) \in \text{set} \ \text{cs}; \ \text{rec-comb} \)
by (auto simp: \text{list-all-iff})

hence name \( \in \text{fst} \ \text{fset-of-list} \ \text{rs} \ \text{\text{\cup}} \ \text{C} \)
using \( \text{name} \in \text{consts} \ \text{rhs}; \ \text{all-consts-def} \)

201
by blast
moreover have \( \neg \text{shadows-consts } \text{pat} \)
using \( \langle \text{not-shadows-vconsts } (Vabs \ cs \ \Gamma'_{2}) \rangle \langle (\text{pat}, \ rhs) \in \text{set } cs \rangle \)
by (auto simp: list-all-iff shadows-consts-def all-consts-def)
ultimately show \( \text{False} \)
using \( \langle \text{name } |\in| \text{frees } \text{pat} \rangle \)
unfolding shadows-consts-def fdisjnt-alt-def all-consts-def
by auto
qed
ultimately show rel-option vrelated (fmlookup (fmap-of-list rs) name)
(\( \text{fmlookup } (\Gamma'_{2} + + f \text{ mk-rec-env } css \ \Gamma'_{2} + + f \text{ env}_{2}) \text{ name} \))
by simp
qed
next
have wellformed-venv \( \Gamma'_{2} \)
by fact
moreover have wellformed-venv env\(_{2}\)
apply (rule vwellformed.vmatch-env)
apply (rule vfind-match-elem)
apply fact
apply (rule veval’.wellformed)
apply fact
using \( \langle \text{wellformed } (t \ \$_{s} \ u) \rangle \) apply simp
apply fact
done
moreover have wellformed-venv (mk-rec-env css \( \Gamma'_{2} \))
unfolding mk-rec-env-def
using \( \langle \text{vwellformed } (Vrecabs \ \cdots \ \cdots) \rangle \)
by (auto intro: fmdomI)
ultimately show wellformed-venv (\( \Gamma'_{2} + + f \text{ mk-rec-env } css \ \Gamma'_{2} + + f \text{ env}_{2} \))
by blast
next
show welldefined rhs
using \( \langle \text{vwelldefined } (Vabs \ cs \ \Gamma'_{1}) \rangle \langle (\text{pat}, \ rhs) \in \text{set } cs \rangle \)
by (auto simp: list-all-iff)
next
have fmpred (\( \lambda \text{- vwelldefined} \)) \( \Gamma'_{1} \)
using \( \langle \text{vwelldefined } (Vabs \ cs \ \Gamma'_{1}) \rangle \) by simp
moreover have fmpred (\( \lambda \text{- vwelldefined} \)) env\(_{1}\)
apply (rule vwelldefined.vmatch-env)
apply (rule vfind-match-elem)
apply fact
apply (rule veval-welldefined)
apply fact
next
apply fact
using rec-comb apply simp
done

ultimately show fmpred (λ-. welldefined) (Γ' 1 ++ f env)
by blast
qed

then obtain val where rs, Γ' 1 ++ f env ⊢ rhs ⊥ val ⊥ val 1 ≈ val 2
by blast

show ?case
apply (intro conjI exI)
apply (rule veval.rec-comb)
apply fact+
done
next
case (constr name Γ 2 ts us 2)

have list-all2 ((∃ u 2. (∃ u 1. rs, Γ 1 ⊢ t ↓ u 1 ∧ Γ v u 1 ≈ u 2)) ts us)
using ⟨list-all2 - ts us⟩
proof (rule list.rel-mono-strong, elim conjE impE allE exE)
fix t u 2
assume t ∈ set ts u 2 ∈ set us
assume Γ 2 ⊢ v t ↓ u 2
show wellformed t welldefined t ¬ shadows-consts t
using constr (t ∈ set ts)
unfolding welldefined.list-comb wellformed.list-comb shadows.list-comb
by (auto simp: list-all-iff list-ex-iff)

show wellformed-venv Γ 2
not-shadows-vconsts-env Γ 2
fmpred (λ-. welldefined) Γ 1
by fact+

have consts t |∈| fset-of-list (map consts ts)
using ⟨t ∈ set ts⟩ by (simp add: fset-of-list-elem)
hence consts t |⊂| consts (name $$ ts)
unfolding consts-list-comb
by (metis ffUnion-subset-elem le-supI2)
thus fmrel-on-fset (consts t) vrelated (fmap-of-list rs) Γ 2
using constr by (blast intro: fmrel-on-fsubset)

have frees t |∈| fset-of-list (map frees ts)
using ⟨t ∈ set ts⟩ by (simp add: fset-of-list-elem)
hence frees t |⊂| frees (name $$ ts)
unfolding frees-list-comb const-sterm-def freess-def
by (auto intro!: ffUnion-subset-elem)
thus \textsf{fmrel-on-fset} (\textsf{frees} \, t) \, \textsf{vrelated} \, \Gamma_1, \Gamma_2
using constr by (blast intro: \textsf{fmrel-on-fsubset})
qed auto

then obtain \textsf{us}_1 \, \textsf{where} \, \textsf{list-all2} \, (\textsf{veval} \, rs \, \Gamma_1) \, ts \, \textsf{us}_1 \, \textsf{list-all2} \, \textsf{vrelated} \, \textsf{us}_1 \, \textsf{us}_2
by induction auto

thus \textsf{?case}
using constr
by (auto intro: \textsf{veval}.constr \textsf{vrelated}.constr)
qed

\textbf{Preservation of extensional equality}

\textbf{lemma (in constants) \textsf{veval’-agree-eq}:}
\assumes \Gamma_2 \vdash \un \, v \, \textsf{wellformed} \, t \, \textsf{wellformed-venv} \, \Gamma_2
\assumes \neg \, \textsf{shadows-consts} \, t \, \textsf{not-shadows-vconsts-env} \, \Gamma_2
\assumes \textsf{welldefined} \, t
\assumes \textsf{closed} \, t
\assumes \textsf{fmrel-on-fset} (\textsf{consts} \, t) \, \textsf{vrelated} \, (\textsf{fmap-of-list} \, rs) \, \Gamma_2
\obtains \, v_1 \, \textsf{where} \, rs, \, \textsf{fmempty} \, \Gamma \, \vdash \, v_1 \, \approx \, v_2
\proof (\text{rule \textsf{veval’-correct}[where \, \Gamma_1 = \textsf{fnempty}])
\show \textsf{fmpred} (\lambda v. \, \textsf{vwelldefined} \, \textsf{fmempty} \, \textsf{by} \, \text{simp}
\next
\show \textsf{fmrel-on-fset} (\textsf{frees} \, t) \, \textsf{vrelated} \, \textsf{fmempty} \, \Gamma_2
\using (\textsf{closed} \, t) \, \text{unfolding} \, \textsf{closed-except-def} \, \text{by} \, \text{auto}
qed (\text{rule \textsf{assms}}) +

end
next
case (var Γ name val)
hence name |∈| ids (Svar name)
  unfolding ids-def by simp
with var have rel-option erelated (fmlookup Γ' name) (fmlookup Γ name)
  by (auto dest: fmrel-on-fsetD)
then obtain val' where fmlookup Γ' name = Some val' val' ≈c val
  using fmlookup Γ name = Some val
  by cases auto
thus ?case
  using var by (auto intro: veval'.var)
next
case (abs Γ cs)
hence Vabs cs Γ' ≈c Vabs cs Γ
  by (auto intro: erelated.abs)
thus ?case
  using abs by (auto intro: veval'.abs)
next
case (comb Γ t cs ΓΛ u v2 env pat rhs val)
  have fmrel-on-fset (ids t) erelated Γ' Γ
    apply (rule fmrel-on-fsubset)
    apply fact
    unfolding ids-def by auto
  then obtain v1' where Γ' t ↓ v1' v1' ≈c Vabs cs ΓΛ
    using comb by (auto simp: closed-except-def)
  then obtain ΓΛ' where v1' = Vabs cs ΓΛ' fmrel-on-fset (ids (Sabs cs)) erelated
    ΓΛ' ΓΛ
    by (auto elim: erelated.cases)
  have fmrel-on-fset (ids u) erelated Γ' Γ
    apply (rule fmrel-on-fsubset)
    apply fact
    unfolding ids-def by auto
  then obtain v2' where Γ' u ↓ v2' v2' ≈c v2
    apply –
    apply (erule comb.IH(2))
    using comb by (auto simp: closed-except-def)
  have rel-option (rel-prod (fmrel erelated) (=)) (vfind-match cs v2') (vfind-match cs v2)
    using (v2' ≈c v2) by (rule erelated.vfind-match-rel')
  then obtain env' where fmrel erelated env' env vfind-match cs v2' = Some
    (env', pat, rhs)
    using comb by cases auto
  have vclosed (Vabs cs ΓΛ)
    apply (rule veval'.closed)
    using comb by (auto simp: closed-except-def)
have \( \text{vclosed} v_2 \)
apply (rule \( \text{veval}'-\text{closed} \))
using \( \text{comb} \) by (auto simp: \( \text{closed-except-def} \))

have \( \text{closed-except} (\text{Sabs} \ cs) (\text{fmdom} \ \Gamma_\Lambda) \)
using (\( \text{vclosed} (\text{Vabs} \ cs \ \Gamma_\Lambda) \)) by (auto simp: \( \text{Sterm} . \text{closed-except-simps} \))
hence \( \text{frees} (\text{Sabs} \ cs) \subseteq fmdom \ \Gamma_\Lambda \)
unfolding \( \text{closed-except-def} . \)

have \( \text{vwellformed} (\text{Vabs} \ cs \ \Gamma_\Lambda) \)
apply (rule \( \text{veval}'-\text{wellformed} \))
apply fact
using \( \text{comb} \) by auto

have \((\text{pat}, \text{rhs}) \in \text{set} \ cs\)
by (rule \( \text{vfind-match-elem} \)) fact
hence \( \text{linear} \ \text{pat} \)
using (\( \text{vwellformed} (\text{Vabs} \ cs \ \Gamma_\Lambda) \))
by (auto simp: \( \text{list-all-iff} \))
hence \( \text{frees} \ \text{pat} = \text{patvars} (\text{mk-pat} \ \text{pat}) \)
by (simp add: \( \text{mk-pat-frees} \))
hence \( \text{fmdom} \ \text{env} = \text{frees} \ \text{pat} \)
apply simp
apply (rule \( \text{vmatch-dom} \))
apply (rule \( \text{vfind-match-elem} \))
apply (rule \( \text{comb} \))
done

have \( \text{vwelldefined}' (\text{Vabs} \ cs \ \Gamma_\Lambda) \)
apply (rule \( \text{veval}'-\text{welldefined}' \))
apply fact
using \( \text{comb} \) by auto
hence \( \text{consts} \ \text{rhs} \ \subseteq fmdom \ \Gamma_\Lambda \ \cup \ C \ \text{fdisjnt} \ C \ (\text{fmdom} \ \Gamma_\Lambda) \)
using (\( \text{pat}, \text{rhs} \in \text{set} \ cs \))
by (auto simp: \( \text{list-all-iff} \))

have \( \text{not-shadows-vconsts} (\text{Vabs} \ cs \ \Gamma_\Lambda) \)
apply (rule \( \text{veval}'-\text{shadows} \))
using \( \text{comb} \) by auto
obtain \( \text{val}' \) where \( \Gamma_\Lambda' ++ f \ \text{env}' \vdash v \ \text{rhs} \ \downarrow \ \text{val}' \ \approx_e \ \text{val} \)
proof (erule \( \text{comb.\text{IH}} \))
show \( \text{closed-venv} (\Gamma_\Lambda ++ f \ \text{env}) \)
apply rule
using (\( \text{vclosed} (\text{Vabs} \ cs \ \Gamma_\Lambda) \))
apply simp
apply (rule \( \text{vclosed.\text{vmatch-env}} \))
apply (rule \( \text{vfind-match-elem} \))
apply fact
apply fact

206
done

next

show wellformed rhs
  using ((pat, rhs) ∈ set cs) (vwellformed (Vabs cs Γ₁))
  by (auto simp: list-all-iff)

next

show wellformed-venv (Γ₁ ++ f env)
  apply rule
  using (vwellformed (Vabs cs Γ₁)) apply simp
  apply (rule vwellformed_vmatch-env)
  apply (rule vfind-match-elem)
  apply (rule comb)
  apply (rule veval'-wellformed)
  apply fact
  using comb by auto

next

show closed-except rhs (fmrel Γ₂ ++ f env)
  using ((pat, rhs) ∈ set cs) (vclosed (Vabs cs Γ₂)) (fmrel env = frees pat)
  by (auto simp: list-all-iff)

have fmrel env = fmrel env'
  using (fmrel env = frees pat)
proof
fix id
assume id |∈| ids rhs
thus rel-option erelated (fmlookup (Γ₁ ++ f env') id) (fmlookup (Γ₁ ++ f env) id)
  unfolding ids-def
proof (cases rule: funion-strictE)
case A
  hence id |∈| fmrel env |∪| fmrel Γ₂
    using (closed-except rhs (fmrel Γ₂ ++ f env'))
    unfolding closed-except-def
    by auto
  thus ?thesis
    proof (cases rule: funion-strictE)
case A
  hence id |∈| fmrel env'
    using (fmrel env = frees pat) (fmrel env = fmrel env')
    by simp
    with A show ?thesis
      using (fmrel env = frees pat) (fmrel env = fmrel env') by auto
next
case B

207
hence \( \text{id} \in \text{frees} \text{pat} \)
using \( \text{fmdom env} = \text{frees pat} \) by simp

hence \( \text{id} \in \text{frees} (\text{Sabs cs}) \)
apply auto
unfolding \( \text{ffUnion-alt-def} \)
apply simp
apply (rule \( \text{fBexI[where x = (pat, rhs)]} \))
using \( \text{id} \in \text{frees rhs} \) apply simp
unfolding \( \text{fset-of-list-elem} \)
apply (rule \( \text{(pat, rhs) \in set cs} \))
done

hence \( \text{id} \in \text{ids} (\text{Sabs cs}) \)
unfolding \( \text{ids-def} \) by simp

have \( \text{id} \in \text{consts} (\text{Sabs cs}) \)
apply auto
unfolding \( \text{ffUnion-alt-def} \)
apply simp
apply (rule \( \text{fBexI[where x = (pat, rhs)]} \))
apply simp
apply fact
unfolding \( \text{fset-of-list-elem} \)
apply (rule \( \text{(pat, rhs) \in set cs} \))
done

hence \( \text{id} \in \text{ids} (\text{Sabs cs}) \)
unfolding \( \text{ids-def} \) by auto

show \( \text{thesis} \)
using \( \text{fmdom env} = \text{fmdom env'} \)
apply auto
apply (rule \( \text{fmrelD} \))
apply (rule \( \text{fmrel ereleted env' env} \))
apply (rule \( \text{fmrel-on-fsetD} \))
apply (rule \( \text{id} \in \text{ids (Sabs cs)} \))
apply (rule \( \text{fmrel-on-fset (ids (Sabs cs)) ereleted \; \Gamma_A' \; \Gamma_{\Lambda'} } \))
done

qed

next

next case B

have \( \text{id} \in \text{consts (Sabs cs)} \)
apply auto
unfolding \( \text{ffUnion-alt-def} \)
apply simp
apply (rule \( \text{fBexI[where x = (pat, rhs)]} \))
apply simp
apply fact
unfolding \( \text{fset-of-list-elem} \)
apply (rule \( \text{(pat, rhs) \in set cs} \))
done

hence \( \text{id} \in \text{ids (Sabs cs)} \)
unfolding \( \text{ids-def} \) by auto

show \( \text{thesis} \)
using \( \text{fmdom env} = \text{fmdom env'} \)
apply auto
apply (rule \( \text{fmrelD} \))
apply (rule \( \text{fmrel ereleted env' env} \))
apply (rule \( \text{fmrel-on-fsetD} \))
apply (rule \( \text{id} \in \text{ids (Sabs cs)} \))
apply (rule \( \text{fmrel-on-fset (ids (Sabs cs)) ereleted \; \Gamma_A' \; \Gamma_{\Lambda'} } \))
done

qed

208
qed

next

show fnpred (\lambda-, vwelldefined') (\Gamma \Lambda ++f env)

proof
have vwelldefined' (Vabs cs \Gamma \Lambda)
apply (rule veval'-welldefined')
apply fact
using comb by auto
thus fnpred (\lambda-, vwelldefined') \Gamma \Lambda
by simp
next
have vwelldefined' v_2
apply (rule veval'-welldefined')
apply fact
using comb by auto

show fnpred (\lambda-, vwelldefined') env
apply (rule vmatch-welldefined')
apply (rule vfind-match-elm)
apply fact+
done

qed

next

have fdisjnt C (fmdom \Gamma \Lambda)
using ⟨vwelldefined' (Vabs cs \Gamma \Lambda)⟩ by simp
moreover have fdisjnt C (fmdom env)
unfolding ⟨fmdom env = -⟩
using ⟨(pat, rhs) ∈ set cs⟩ ⟨not-shadows-vconsts (Vabs cs \Gamma \Lambda)⟩
by (auto simp; list-all-iff all-consts-def fdisjnt-alt-def)
ultimately show fdisjnt C (fmdom (\Gamma \Lambda ++f env))
unfolding fdisjnt-alt-def by auto
next

show ¬ shadows-consts rhs
using ⟨(pat, rhs) ∈ set cs⟩ ⟨not-shadows-vconsts (Vabs cs \Gamma \Lambda)⟩
by (auto simp; list-all-iff)
next

have not-shadows-vconsts-env \Gamma \Lambda
using ⟨not-shadows-vconsts (Vabs cs \Gamma \Lambda)⟩ by auto
moreover have not-shadows-vconsts-env env
apply (rule not-shadows-vconsts.vmatch-env)
apply (rule vfind-match-elm)
apply fact
apply (rule veval'-shadows)
using comb by auto
ultimately show not-shadows-vconsts-env (\Gamma \Lambda ++f env)
by blast
next

show consts rhs |⊆| fmdom (\Gamma \Lambda ++f env) |∪| C
using ⟨consts rhs |⊆| -⟩
by auto

qed

moreover have $\Gamma' \vdash v \mathcal{S}_s \uparrow \mathcal{V} val'$
proof (rule veval',comb)
show $\Gamma' \vdash \mathcal{V} abs \mathcal{C} \binom{\Lambda'}{\Gamma}$
using $\Gamma' \vdash v_1'$
unfolding $\langle v_1' = \mathcal{V} \rangle$.
qed fact+

ultimately show $?case$
using comb by metis

next

\textbf{case (rec-comb $\Gamma$ $t$ $\mathcal{C} \binom{\Lambda}{\Gamma}$ $cs$ $u$ $v_2$ $env$ $pat$ $rhs$ $val$)}

have $\text{fmrel-on-fset (ids $t$)}$ $\text{erelated} \Gamma' \Gamma$
apply (rule $\text{fnrel-on-fsubset}$)
apply $\text{fact}$
unfolding $\text{ids-def by auto}$
then obtain $v_1'$ where $\Gamma' \vdash v_1'$ $\mathcal{V} \text{recabs cs name } \Gamma_A$
using $\text{rec-comb by (auto simp: closed-except-def)}$
then obtain $\Gamma_A'$
where $v_1' = \mathcal{V} \text{recabs cs name } \Gamma_A'$
and $\text{pred-fmap (\lambda cs. fmrel-on-fset (ids (Sabs cs)) erelated } \Gamma_A' \Gamma_A) \text{ cs}$
by (auto elim: erelated_cases)

have $\text{fmrel-on-fset (ids $u$)}$ $\text{erelated} \Gamma' \Gamma$
apply (rule $\text{fnrel-on-fsubset}$)
apply $\text{fact}$
unfolding $\text{ids-def by auto}$
then obtain $v_2'$ where $\Gamma' \vdash v_2'$ $\mathcal{V} \text{find-match cs } v_2'$
apply $-$
apply (erule $\text{rec-comb.IH (2)}$)
using $\text{rec-comb by (auto simp: closed-except-def)}$

have $\text{rel-option (rel-prod (fmrel erelated) (=) (vfind-match cs v_2')) (vffind-match cs v_2)}$
using $\langle v_2' \approx e v_2' \rangle$ by (rule erelated, vfind-match-rel')
then obtain $\text{env'}$ where $\text{fmrel erelated env'} \text{ env vfind-match cs } v_2' = \text{Some}$
(\text{env'}, pat, rhs)
using $\text{rec-comb by cases auto}$

have $\text{vclosed } (\mathcal{V} \text{recabs cs name } \Gamma_A)$
apply (rule $\text{veval'}$-closed)
using $\text{rec-comb by (auto simp: closed-except-def)}$

hence $\text{vclosed } (\mathcal{V} \text{abs cs } \Gamma_A)$
using $\text{rec-comb by (auto simp: closed-except-def)}$

have $\text{vclosed } v_2$
apply (rule $\text{veval'}$-closed)
using rec-comb by (auto simp: closed-except-def)

have closed-except \((Sabs cs)\) \((\text{fmdom } \Gamma_\Lambda)\)
  using \((\text{vclosed} \ (Vabs cs \Gamma_\Lambda))\) by (auto simp: Stem.closed-except-simps)
hence frees \((Sabs cs)\) \(\subseteq\) \(\text{fmdom } \Gamma_\Lambda\)
unfolding closed-except-def.

have \(\text{vwellformed} \ (Vrecabs css \text{ name } \Gamma_\Lambda)\)
  apply (rule veval\'-wellformed)
  apply fact
  using rec-comb by auto
hence \(\text{vwellformed} \ (Vabs cs \Gamma_\Lambda)\)
  using rec-comb by (auto simp: closed-except-def)

have \((\text{pat}, \text{rhs})\) \(\in\) set \(cs\)
  by (rule vfind-match-elem) fact
hence linear \(\text{pat}\)
  using \((\text{vwellformed} \ (Vabs cs \Gamma_\Lambda))\)
  by (auto simp: list-all-iff)
hence frees \(\text{pat}\) = \(\text{patvars} \ (\text{mk-pat } \text{pat})\)
  by (simp add: mk-pat-frees)
hence \(\text{fmdom } \text{env} = \text{frees } \text{pat}\)
  apply simp
  apply (rule vmatch-dom)
  apply (rule vfind-match-elem)
  apply (rule rec-comb)
done

have \(\text{vwelldefined}' \ (Vrecabs css \text{ name } \Gamma_\Lambda)\)
  apply (rule veval\'-welldefined')
  apply fact
  using rec-comb by auto
hence \(\text{consts } \text{rhs} \subseteq \text{fmdom } \Gamma_\Lambda | \bigcup | \text{(C | \bigcup | } \text{fmdom } \text{css}) | \text{fdisjnt } C \ (\text{fmdom } \Gamma_\Lambda)\)
  using \((\text{pat}, \text{rhs})\) \(\in\) set \(cs\) \(\text{fmlookup css name} = \text{Some } cs\)
  by (auto simp: list-all-iff dest!: fmpredD [where \(m = css\)]

have not-shadows-vconsts \((Vrecabs css \text{ name } \Gamma_\Lambda)\)
  apply (rule veval\'-shadows)
  using rec-comb by auto
hence not-shadows-vconsts \((Vabs cs \Gamma_\Lambda)\)
  using rec-comb by auto

obtain \(\text{val}' \ \text{where } \Gamma_\Lambda' \ ++_f \ \text{mk-rec-env } \text{css } \Gamma_\Lambda' \ ++_f \ \text{env}' \vdash_v \text{rhs} \downarrow \text{val}' \ \text{val} \approx_c \text{val}\)
proof (erule rec-comb.IH)
show closed-venv \((\Gamma_\Lambda \ ++_f \ \text{mk-rec-env } \text{css } \Gamma_\Lambda \ ++_f \ \text{env})\)
  apply rule
  apply rule
  using \((\text{vclosed} \ (Vabs cs \Gamma_\Lambda))\) apply simp

211
unfolding \textit{mk-rec-env-def} \\
using \langle \textit{vclosed} (Vrecabs css name \Gamma_{\Lambda}) \rangle \ apply \ (\textit{auto intro: fmdomI})[] \\
apply \ (\textit{rule vclosed.vmatch-env}) \\
apply \ (\textit{rule vfind-match-elem}) \\
apply \ fact \\
apply \ fact \\
done \\
next \\
show \textit{wellformed \ rhs} \\
using \ ((\textit{pat}, rhs) \in \textit{set \ cs}) \ (\textit{vwellformed} (Vabs cs \Gamma_{\Lambda})) \\
by \ (\textit{auto simp: list-all-iff}) \\
next \\
show \textit{wellformed-venv} (\Gamma_{\Lambda} ++ f \textit{mk-rec-env css} \Gamma_{\Lambda} ++ f \textit{env}) \\
apply \ \textit{rule} \\
apply \ \textit{rule} \\
using \ (\textit{vwellformed} (Vabs cs \Gamma_{\Lambda})) \ apply \ \textit{simp} \\
unfolding \ \textit{mk-rec-env-def} \\
using \ (\textit{vwellformed} (Vrecabs css name \Gamma_{\Lambda})) \ apply \ (\textit{auto intro: fmdomI})[] \\
apply \ (\textit{rule vwellformed.vmatch-env}) \\
apply \ fact \\
apply \ (\textit{rule vwellformed.vmatch-env}) \\
apply \ fact \\
apply \ \textit{fact} \\
using \ \textit{rec-comb by auto} \\
next \\
have \ \textit{closed-except rhs} (fmdom (\Gamma_{\Lambda} ++ f \textit{env})) \\
using \ ((\textit{pat}, rhs) \in \textit{set \ cs}) \ (\textit{vclosed} (Vabs cs \Gamma_{\Lambda})); \ (fmdom \ env = \textit{frees \ pat}) \\
by \ (\textit{auto simp: list-all-iff closed-except-def}) \\
thus \ \textit{closed-except rhs} (fmdom (\Gamma_{\Lambda} ++ f \textit{mk-rec-env css} \Gamma_{\Lambda} ++ f \textit{env})) \\
unfolding \ \textit{closed-except-def} \\
by \ \textit{auto} \\

have \ \textit{fmdom \ env} = \textit{fmdom \ env'} \\
using \ (\textit{fmrel \ erelated \ env' \ env}) \\
by \ (\textit{metis \ fmrel-fmdom-eq}) \\

have \ \textit{fmrel-on-fset} (\textit{ids \ rhs}) \ \textit{erelated} (\textit{mk-rec-env css} \Gamma_{\Lambda'}) (\textit{mk-rec-env css} \Gamma_{\Lambda}) \\
unfolding \ \textit{mk-rec-env-def} \\
apply \ \textit{rule} \\
apply \ \textit{simp} \\
unfolding \ \textit{option.rel-map} \\
apply \ (\textit{rule option.rel-refl}) \\
apply \ (\textit{rule \ erelated.intro}) \\
apply \ (\textit{rule \ pred-fmap} (\lambda \textit{cs}. \ \textit{fmrel-on-fset} (\textit{ids} (\textit{Sabs \ cs})) \ \textit{erelated} \ \Gamma_{\Lambda'} \Gamma_{\Lambda}) \ \textit{css}) \\
done \\

have \ \textit{fmrel-on-fset} (\textit{ids} (\textit{Sabs \ cs})) \ \textit{erelated} \ \Gamma_{\Lambda'} \Gamma_{\Lambda} \\
using \ (\textit{pred-fmap} (\lambda \textit{cs}. \ \textit{fmrel-on-fset} (\textit{ids} (\textit{Sabs \ cs})) \ \textit{erelated} \ \Gamma_{\Lambda'} \Gamma_{\Lambda}) \ \textit{css}) \\

212
rec-comb
   by auto

have fmdom (mk-rec-env css Γₐ) = fmdom (mk-rec-env css Γₐ')
unfolding mk-rec-env-def by auto

show fmrel-on-fset (ids rhs) erelated (Γₐ' ++ₐ mk-rec-env css Γₐ' ++ₐ env')
(Γₐ ++ₐ mk-rec-env css Γₐ ++ₐ env)
proof
  fix id
  assume id |∈| ids rhs

  thus rel-option erelated (fmlookup (Γₐ' ++ₐ mk-rec-env css Γₐ' ++ₐ env') id)
   (fmlookup (Γₐ ++ₐ mk-rec-env css Γₐ ++ₐ env) id)
unfolding ids-def
proof (cases rule: funion-strictE)
  case A
  hence id |∈| fmdom env |∪| fmdom Γₐ
     using ⟨closed-except rhs (fmdom (Γₐ ++ₐ env))⟩
  unfolding closed-except-def
  by auto

thus ?thesis
proof (cases rule: funion-strictE)
  case A
  hence id |∈| fmdom env'
     using ⟨fmdom env = frees pat: fmdom env = fmdom env'⟩
     by simp
with A show ?thesis
  using ⟨fmrel erelated env' env⟩ by auto
next
  case B
  hence id |∉| frees pat
     using ⟨fmdom env = frees pat⟩ by simp
  hence id |∈| frees (Sabs cs)
  apply auto
  unfolding fUnion-alt-def
  apply simp
  apply (rule fBexI[where x = (pat, rhs)])
  using id |∈| frees rhs apply simp
  unfolding fset-of-list-elem
  apply (rule f([pat, rhs] ∈ set cs))
done
  hence id |∈| ids (Sabs cs)
  unfolding ids-def by simp

have id |∉| fmdom env'
  using B unfolding ⟨fmdom env = fmdom env'⟩ by simp
thus ?thesis
using \(\langle \text{id} \mid \notin \rangle \text{fmdom env} \langle \text{fmdom (mk-rec-env css } \Gamma_\Lambda) = \text{fmdom (mk-rec-env css } \Gamma_\Lambda') \rangle\):

apply auto
apply (rule fmrel-on-fsetD)
apply (rule \(\langle \text{id} \mid \in \rangle \text{ids rhs} \rangle\))

apply \(\langle \text{fmdom (mk-rec-env css } \Gamma_\Lambda) = \text{fmdom (mk-rec-env css } \Gamma_\Lambda') \rangle\):

apply (rule fmrel-on-fsetD)
apply (rule \(\langle \text{id} \mid \in \rangle \text{ids (Sabs cs)} \rangle\))

apply (rule \(\langle \text{fmrel-on-fset (ids (Sabs cs)) erelated } \Gamma_\Lambda' \Gamma_\Lambda' \rangle\))
done

qed

next

\begin{align*}
\text{next} & \\
\text{case } B \\
\text{have } & \text{id } \mid \in \text{ consts } (Sabs cs) \\
\text{apply } & \text{auto} \\
\text{unfolding } & \text{ffUnion-alt-def} \\
\text{apply } & \text{simp} \\
\text{apply } & \text{(rule } \text{fBexI[where } x = (\text{pat, rhs})]) \\
\text{apply } & \text{simp} \\
\text{apply } & \text{fact} \\
\text{unfolding } & \text{fset-of-list-elem} \\
\text{apply } & \text{(rule } (\text{pat, rhs}) \in \text{set cs}) \\
\text{done} \\
\text{hence } & \text{id } \mid \in \text{ ids } (Sabs cs) \\
\text{unfolding } & \text{ids-def by auto} \\
\text{show } & \text{?thesis} \\
\text{using } & \text{(fmdom env } = \text{fmdom env'} \langle \text{fmdom (mk-rec-env css } \Gamma_\Lambda) = \text{fmdom (mk-rec-env css } \Gamma_\Lambda') \rangle\):
\end{align*}

apply auto
apply (rule fnrelD)
apply (rule \(\langle \text{fmrel erelated env'} env \rangle\))
apply (rule fnrel-on-fsetD)
apply (rule \(\langle \text{id} \mid \in \rangle \text{ids rhs} \rangle\))
apply (rule \(\langle \text{fmrel-on-fset (ids rhs) erelated } \Gamma_\Lambda' \Gamma_\Lambda' \rangle\))
done

qed

next

\begin{align*}
\text{show } & \text{fmpred } (\lambda-. \text{vwelldefined'}) (\Gamma_\Lambda ++f \text{ mk-rec-env css } \Gamma_\Lambda ++f \text{ env}) \\
\text{proof (intro } \text{fmpred-add}) \\
\text{have } & \text{vwelldefined'} (Vrecabs css name } \Gamma_\Lambda) \\
\end{align*}
apply (rule veval'-'welldefined')
apply fact
using rec-comb by auto
thus fmpred (λ-. vwelldefined') Γ fmpred (λ-. vwelldefined') (mk-rec-env css Γ)

unfolding mk-rec-env-def by (auto intro: fmdomI)

next
have vwelldefined' v2
apply (rule veval'-'welldefined')
apply fact
using rec-comb by auto

show fmpred (λ-. vwelldefined') env
apply (rule vmatch-welldefined')
apply (rule vfind-match-elem)
apply fact+
done

qed

next
have fdisjnt C (fmdom env)
unfolding (fmdom env = -)
using ((pat, rhs) ∈ set cs) (not-shadows-vconsts (Vabs cs Γ))
by (auto simp: list-all-iff all-consts-def fdisjnt-alt-def)
moreover have fdisjnt C (fmdom css)
using (vwelldefined' (Vrecabs css name Γ)) by simp
ultimately show fdisjnt C (fmdom (Γ ++ f mk-rec-env css Γ ++ f env))
using (fdisjnt C (fmdom Γ))
unfolding fdisjnt-alt-def mk-rec-env-def by auto

next
show ¬ shadows-consts rhs
using ((pat, rhs) ∈ set cs) (not-shadows-vconsts (Vabs cs Γ))
by (auto simp: list-all-iff)

next
have not-shadows-vconsts-env Γ
using (not-shadows-vconsts (Vabs cs Γ)) by auto
moreover have not-shadows-vconsts-env env
apply (rule not-shadows-vconsts.vmatch-env)
apply (rule vfind-match-elem)
apply fact
apply (rule veval'-'shadows)
using rec-comb by auto
moreover have not-shadows-vconsts-env (mk-rec-env css Γ)
unfolding mk-rec-env-def
using (not-shadows-vconsts (Vrecabs css name Γ))
by (auto intro: fmdomI)
ultimately show not-shadows-vconsts-env (Γ ++ f mk-rec-env css Γ ++ f env)
by blast
next

show consts rhs \subseteq fmdom (\Gamma \Lambda ++ f mk-rec-env css \Gamma \Lambda ++ f env) |\cup| C
using :consts rhs \subseteq unfolding mk-rec-env-def
by auto
qed

moreover have \Gamma' \vdash v_1 \Downarrow \text{val}'
proof (rule veval'.rec-comb)
  show \Gamma' \vdash v \Downarrow \text{Vrecabs css name} \Gamma_\Lambda'
  using \Gamma' \vdash v \Downarrow v_1'
  unfolding \langle v_1' = \circ \rangle.
qed fact+

ultimately show ?case
using rec-comb by metis
next
case (constr name \Gamma ts us)

have list-all (\lambda t. fmrel-on-fset (ids t) erelated \Gamma' \Gamma \land closed-except t (fmdom \Gamma) \land wellformed t \land \text{consts t} \mid \subseteq| fmdom \Gamma \mid \cup| C \land \neg \text{shadows-consts t}) ts
apply (rule list-allI)
apply rule
apply (rule fmrel-on-fsubset)
apply (rule constr)
subgoal
unfolding ids-list-comb
by (induct ts; auto)
subgoal
apply (intro conjI)
subgoal
using (closed-except (name $$ ts) (fmdom \Gamma))
unfolding closed.list-comb by (auto simp: list-all-iff)
subgoal
using (wellformed (name $$ ts))
unfolding wellformed.list-comb by (auto simp: list-all-iff)
subgoal
using (consts (name $$ ts) \mid \subseteq| fmdom \Gamma \mid \cup| C)
unfolding consts-list-comb
by (metis Ball-set constr.prems(8) special-constants.sconsts-list-comb)
subgoal
using (\neg \text{shadows-consts} (name $$ ts))
unfolding shadows.list-comb by (auto simp: list-ex-iff)
done
done

obtain us' where list-all3 (\lambda t u u'. \Gamma' \vdash v \Downarrow u' \land u' \approx_e u) ts us us'
using \langle list-all2 - - \rangle \langle list-all - ts \rangle;
proof (induction arbitrary: thesis rule: list.rel-induct)
case (Cons t ts u us)

216
then obtain \(u'\) where
\[\text{list-all3 } (\lambda t u u'. \Gamma' \vdash v t \Downarrow u' \land u' \approx_e u) \; ts \; us \; us'\]

by auto

have
\[
\text{fmrel-on-fset } (\text{ids } t) \;
\text{erelated } \Gamma' \; \Gamma \;
\text{closed-except } t \; (\text{fmdom } \Gamma) \\
\text{wellformed } t \; \text{consts } t \; \subseteq \; \text{fmdom } \Gamma \; |\cup| C \sim \text{shadows-consts } t \\
\text{using } \text{Cons by auto}
\]

then obtain \(u'\) where
\[\Gamma' \vdash v t \Downarrow u' \; u' \approx_e u\]

using \(\langle \text{closed-venv } \Gamma \rangle \; \langle \text{wellformed-venv } \Gamma \rangle \; \langle \text{fdisjnt } C \; (\text{fmdom } \Gamma) \rangle \; \langle \text{fmpred } (\lambda-. \; \text{vwelldefined'}) \; \Gamma \rangle\)

using \(\langle \text{not-shadows-vconsts-env } \Gamma \rangle \; \text{Cons.hyps}\)

by blast

show \(?\text{case}\)

apply (rule Cons.prems)

apply (rule list-all3-cons)

apply fact

apply (rule conjI)

apply fact+

done

qed auto

show \(?\text{case}\)

apply (rule constr.prems)

apply (rule veval'.constr[where \(us = us'\)])

apply fact

using \(\langle \text{list-all3 } - \; ts \; us \; us'\rangle\)

apply (induct; auto)

apply (rule erelated.intros)

using \(\langle \text{list-all3 } - \; ts \; us \; us'\rangle\)

apply (induct; auto)

done

qed

end
Chapter 4

Preprocessing of code equations

theory Doc-Preproc
imports Main
begin
end

4.1 A type class for correspondence between HOL expressions and terms

theory Eval-Class
imports
  ../Rewriting/Rewriting-Term
  ../Utils/ML-Utils
  Deriving.Derive-Manager
  Dict-Construction.Dict-Construction
begin

no-notation Mpat-Antiquot.mpaq-App (infixl $900$
hide-const (open) Strong-Term.wellformed
declare Strong-Term.wellformed-term-def[simp del]

class evaluate =
  fixes eval :: rule fset ⇒ term ⇒ 'a ⇒ bool (-/ ⊢/ (- ≈/ -) [50,0,50] 50)
  assumes eval-wellformed: rs ⊢ t ≈ a ⇒ wellformed t
begin

definition eval' :: rule fset ⇒ term ⇒ 'a ⇒ bool (-/ ⊢/ (- ⊢/ -) [50,0,50] 50)
where
  rs ⊢ t ⊢ a ←→ wellformed t ∧ (∃ t'. rs ⊢ t →→ t' ∧ rs ⊢ t' ≈ a)

lemma eval'I[intro]:

218
assumes wellformed $t \triangleright t' \rightarrow t' \triangleright a$
shows $\triangleright t \downarrow a$
using assms unfolding eval'-def by auto

lemma eval'E[elim]:
assumes $\triangleright t \downarrow a$
obtains $t'$ where wellformed $t \triangleright t' \rightarrow t' \triangleright a$
using assms unfolding eval'-def by auto

lemma eval-trivI: $\triangleright t \approx a \Rightarrow \triangleright t \downarrow a$
by (auto dest: eval-wellformed)

lemma eval-compose:
assumes wellformed $t \triangleright t' \rightarrow t'' \triangleright a$
shows $\triangleright t \downarrow a$
proof -
from $\triangleright t \downarrow a$ obtain $t''$ where wellformed $t \triangleright t'' \rightarrow t'' \triangleright a$
by blast
moreover hence $\triangleright t \rightarrow t''$
using assms by auto
ultimately show $\triangleright t \downarrow a$
using assms by auto
qed

end

instantiation fun :: (evaluate, evaluate) evaluate begin

  definition eval-fun where
  eval-fun $rs \triangleright t \approx f \triangleright t$ x $\triangleright t \rightarrow t$ x $\rightarrow rs \triangleright t$ $\rightarrow t$ x $\rightarrow a$ x

  instance
  by standard (simp add: eval-fun-def)

end

corollary eval-funD:
assumes $\triangleright t \approx f \triangleright t$ x $\triangleright t$ x $\rightarrow x$
shows $\triangleright t$ $\rightarrow t$ x $\rightarrow f$ x
using assms unfolding eval-fun-def by blast

corollary eval'-funD:
assumes $\triangleright t \rightarrow f \triangleright t$ x $\rightarrow t$ x $\rightarrow x$
shows $\triangleright t$ $\rightarrow t$ x $\rightarrow f$ x
proof -
from assms obtain $t'$ where $\triangleright t \rightarrow t' \rightarrow t' \rightarrow f$
by auto
have wellformed $(t \rightarrow t)$
using assms by auto
moreover have \( rs \vdash t \xrightarrow{\ast} t_x \)
using \( \langle rs \vdash t \xrightarrow{\ast} t'_x \rangle \) by (rule rewrite.rt-fun[unfolded app-term-def])

moreover have \( rs \vdash t' \approx f \) by (rule eval-funD)
ultimately show \( rs \vdash t \approx a \)
by (rule eval-compose)

qed

lemma eval-ext:
assumes wellformed f \( \forall x t. rs \vdash t \downarrow x \implies rs \vdash f \downarrow a x \)
shows \( rs \vdash f \approx a \)
using assms unfolding eval-fun-def by blast

lemma eval'-ext:
assumes wellformed f \( \forall x t. rs \vdash t \downarrow x \implies rs \vdash f \downarrow a x \)
shows \( rs \vdash f \downarrow a \)
apply (rule eval'[I[OF \( \langle \text{wellformed } f \rangle \]})
apply (rule rtranclp.rtrancl-refl)
apply (rule eval-ext)
using assms by auto

lemma eval'-ext-alt:
fixes f :: '\a::evaluate \Rightarrow \b::evaluate
assumes wellformed' f \( \forall x t u. rs \vdash u \downarrow x \implies rs \vdash t[u]_\beta \downarrow f x \)
shows \( rs \vdash \Lambda t u \downarrow f \)
proof (rule eval'-ext)
show wellformed (\( \Lambda t u \))
using assms by simp
next
fix x :: '\a and u
assume \( rs \vdash u \downarrow x \)
show \( rs \vdash \Lambda t u \downarrow f x \)
proof (rule eval-compose)
show wellformed (\( \Lambda t u \))
using assms (\( rs \vdash u \downarrow x \)) by auto
next
show \( rs \vdash \Lambda t u \downarrow f x \)
using (\( rs \vdash u \downarrow x \)) by (auto intro: rewrite.beta)
next
show \( rs \vdash t[u]_\beta \downarrow f x \)
using assms (\( rs \vdash u \downarrow x \)) by auto
qed

qed

lemma eval-impl-wellformed[dest]: \( rs \vdash t \approx a \implies \text{wellformed'} n t \)
by (auto dest: wellformed-inc eval-wellformed[unfolded wellformed-term-def])

lemma eval'-impl-wellformed[dest]: \( rs \vdash t \downarrow a \implies \text{wellformed'} n t \)
unfolding eval'-def by (auto dest: wellformed-inc)
lemma wellformed-unpack:
\[ \text{wellformed}' \ n \ (t \ u) \implies \text{wellformed}' \ n \ t \]
\[ \text{wellformed}' \ n \ (t \ u) \implies \text{wellformed}' \ n \ u \]
\[ \text{wellformed}' \ n \ (\lambda t) \implies \text{wellformed}' \ (\text{Suc} \ n) \ t \]
by auto

lemma replace-bound-aux:
\[ n < 0 \iff \text{False} \]
\[ \text{Suc} \ n < \text{Suc} \ m \iff n < m \]
\[ 0 < \text{Suc} \ n \iff \text{True} \]
\[ ((0::\text{nat}) = 0) \iff \text{True} \]
\[ (0 = \text{Suc} \ m) \iff \text{False} \]
\[ (\text{Suc} \ m = \text{Suc} \ n) \iff n = m \]
\[ (\text{Suc} \ m = 0) \iff \text{False} \]
\[ (\text{if True then } P \text{ else } Q) = P \]
\[ (\text{if False then } P \text{ else } Q) = Q \]
\[ \text{int} \ (0::\text{nat}) = 0 \]
by auto

named-theorems eval-data-intros
named-theorems eval-data-elims

context begin

private definition rewrite-step-term :: term \times term \Rightarrow term \Rightarrow term option
where
rewrite-step-term = rewrite-step

private lemmas rewrite-rt-fun = rewrite.rt-fun[unfolded app-term-def]
private lemmas rewrite-rt-arg = rewrite.rt-arg[unfolded app-term-def]

ML-file tactics.ML

end

method-setup wellformed = \langle Scan.succeed (SIMPLE-METHOD' o Tactics.wellformed-tac) \rangle

end

4.2 Deep embedding of Pure terms into term-rewriting logic

theory Embed
imports
Constructor-Funs.Constructor-Funs
../Utils/Code-Utils

221
Eval-Class

**keywords** embed :: thy-decl

**begin**

```plaintext
fun non-overlapping' :: term ⇒ term ⇒ bool where
  non-overlapping' (Const x) (Const y) ⟷ x ≠ y |
  non-overlapping' (Const _) (- $ -) ⟷ True |
  non-overlapping' (- $ -) (Const -) ⟷ True |
  non-overlapping' (t1 $ t2) (u1 $ u2) ⟷ non-overlapping' t1 u1 ∨ non-overlapping' t2 u2 |
  non-overlapping' - - ⟷ False
```

**lemma** non-overlapping-approx:
  assumes non-overlapping' t u
  shows non-overlapping t u
  using assms
  by (induct t u rule: non-overlapping'.induct) fastforce+

```plaintext
fun pattern-compatible' :: term ⇒ term ⇒ bool where
  pattern-compatible' (t1 $ t2) (u1 $ u2) ⟷ pattern-compatible' t1 u1 ∧ (t1 = u1 ⟷ pattern-compatible' t2 u2) |
  pattern-compatible' t u ⟷ t = u ∨ non-overlapping' t u
```

**lemma** pattern-compatible-approx:
  assumes pattern-compatible' t u
  shows pattern-compatible t u
  using assms
  proof (induction t u rule: pattern-compatible.induct)
  case 2-1
  thus ?case
  by (force simp: non-overlapping-approx)
next
case 2-5
  thus ?case
  by (force simp: non-overlapping-approx)
qed auto

**abbreviation** pattern-compatibles' :: (term × 'a) fset ⇒ bool where
  pattern-compatibles' ≡ fpairwise (λ(lhs1, -) (lhs2, -). pattern-compatible' lhs1 lhs2)

**definition** rules' :: C-info ⇒ rule fset ⇒ bool where
  rules' C-info rs ⟷
  fBall rs rule ∧
  arity-compatibles rs ∧
  is-fmap rs ∧
  pattern-compatibles' rs ∧
  rs ≠ {} ∧
  fBall rs (λ(lhs, -). ¬ pre-constants.shadows-consts C-info (heads-of rs) lhs) ∧
  fdisjnt (heads-of rs) (constructors.C C-info) ∧
```

222
lemma rules-approx:
  assumes rules' C-info rs
  shows rules C-info rs
proof
  show fBall rs rule arity-compatibles rs is-fmap rs rs ≠ ||
  and fBall rs (λ(lhs, -). ¬ pre-constants.shadows consts C-info (heads-of rs) lhs)
  and fBall rs (λ(-, rhs). pre-constants.welldefined C-info (heads-of rs) rhs)
  and fdisjnt (heads-of rs) (constructors.C C-info)
  and distinct (constructors.all-constructors C-info)
  using assms unfolding rules'-def by simp+
next
  have pattern-compatibles' rs
    using assms unfolding rules'-def by simp
  thus pattern-compatibles rs
    by (rule fpairwise-weak) (blast intro: pattern-compatible-approx)
qed

lemma embed-ext: f ≡ g =⇒ f x ≡ g x
by auto

ML-file embed.ML

consts lift-term :: 'a ⇒ term ((-))

setup:
  let
    fun embed ((Const (@{const-name lift-term}, -)) $ t) = HOL-Term.mk-term false t
    | embed (t $ u) = embed t $ embed u
    | embed t = t
  in Context.theory-map (Syntax-Phases.term-check 99 lift (K (map embed))) end

end

4.3 Default instances

theory Eval-Instances
imports Embed
begin

ML-file eval-instances.ML

setup (Eval-Instances.setup)

derive evaluate nat bool list unit prod sum option char num name term

223
end
Chapter 5

Final stage: Translation to CakeML

5.1 Basic CakeML setup

theory Doc-Backend
imports Main
begin
end

global-interpretation name: rekey Name
rewrites inv Name = as-string
proof –
  have bij Name
    by (metis bijI name.exhaust name.inject)
  show rekey Name
    by standard fact

  show inv Name = as-string
    by (metis inv-equality name.exhaust-sel name.sel)
qed

global-interpretation name-as-string: rekey as-string
  by (rule name.inv)

hide-const (open) Lem-string.concat
hide-const (open) sem-env.c
hide-const (open) sem-env.v

definition empty-locn :: locn where
empty-locn = (row = 0, col = 0, offset = 0)

definition empty-locs :: locs where
empty-locs = (empty-locn, empty-locn)

definition empty-state :: unit SemanticPrimitives.state where
empty-state = (clock = 0, refs = [], ffi = empty-ffi-state, defined-types = {},
defined-mods = {})

fun fmap-of-ns :: ('b, string, 'a) namespace ⇒ (name, 'a) fmap where
fmap-of-ns (Bind xs -) = fmap-of-list (map (map-prod Name id) xs)

lemma fnlookup-ns[simp]: fnlookup (fmap-of-ns ns) k = cupcake-nsLookup ns (as-string k)
by (cases ns) (simp add: fnlookup-of-list map-prod-def name.map-of-rekey option.map-ident)

lemma fmap-of-nsBind[simp]: fmap-of-ns (nsBind (as-string k) v0 ns) = fn upd k v0 (fmap-of-ns ns)
by (cases ns) auto

lemma fmap-of-nsAppend[simp]: fmap-of-ns (nsAppend ns1 ns2) = fmap-of-ns ns2 ++ f (fmap-of-ns ns1)
by (cases ns1; cases ns2) simp

lemma fmap-of-alist-to-ns[simp]: fmap-of-ns (alist-to-ns xs) = fmap-of-list (map (map-prod Name id) xs)
unfolding alist-to-ns-def by simp

lemma fmap-of-nsEmpty[simp]: fmap-of-ns nsEmpty = fnempty
unfolding nsEmpty-def by simp

context begin

private lemma build-rec-env-fmap0:
fmap-of-ns (foldr (λ(f, x, e). nsBind f (Recclosure envA fans' f)) fans env) =
fmap-of-ns env ++ f (fmap-of-list (map (λ(f, -). (Name f, Recclosure envA fans' f)) fans))
apply (induction fans arbitrary: env)
apply auto
by (metis (no-types, lifting) fmap-of-nsBind name.sel)

definition cake-mk-rec-env where
cake-mk-rec-env fans env = fmap-of-list (map (λ(f, -). (Name f, Recclosure env fans f)) fans)
lemma build-rec-env-fmap:
\[ \text{fmap-of-ns}\ (\text{build-rec-env}\ \text{funs}\ \text{env}) = \text{fmap-of-ns}\ \text{env} ++ \text{f}\ \text{cake-mk-rec-env}\ \text{funs}\ \text{env} \]

unfolding build-rec-env-def cake-mk-rec-env-def
by (rule build-rec-env-fmap0)

end

5.2 Constructors according to CakeML

definition cake-tctor :: string ⇒ tctor where
cake-tctor name = (if name = "fun" then Ast.TC-fn else Ast.TC-name (Short name))

primrec typ-to-t :: typ ⇒ Ast.t where
typ-to-t (TVar name) = Ast.Tvar (as-string name) |
typ-to-t (TApp name args) = Ast.Tapp (map typ-to-t args) (cake-tctor (as-string name))

context constructors begin

definition as-static-cenv :: c-ns where
as-static-cenv = Bind (rev (map (map prod id (map prod id (TypeId ◦ Short)))) flat-C-info) []

lemma as-static-cenv-cakeml-static-env: cakeml-static-env as-static-cenv
unfolding cakeml-static-env-def as-static-cenv-def
by (auto simp: list.pred-map split: prod.splits)

sublocale cake-static-env?: cakeml-static-env as-static-cenv
by (rule as-static-cenv-cakeml-static-env)

definition as-cake-type-def :: Ast.type-def where
as-cake-type-def =
map (λ(name, dt-def). (map as-string (tparams dt-def), as-string name, map (λ(C, params). (as-string C, map typ-to-t params)) (sorted-list-of-fmap (constructors dt-def))))
(sorted-list-of-fmap C-info)

definition cake-dt-prelude :: Ast.dec where
cake-dt-prelude = Ast.Dtype empty-locs as-cake-type-def

definition cake-all-types :: tid-or-exn set where
cake-all-types = (TypeId ◦ Short ◦ as-string) ’fset all-tdefs

definition empty-state-with-types :: unit SemanticPrimitives.state where
empty-state-with-types =
(\(\) clock = 0, refs = [], ffi = empty-ffi-state, defined-types = cake-all-types,
lemma empty-state-with-types-alt-def:
  empty-state-with-types = empty-state (| defined-types := cake-all-types |
unfolding empty-state-with-types-def empty-state-def
by (auto simp: datatype-record-update)
end

5.2.1 Running the generated type declarations through the semantics

context constants begin

context begin

private lemma state-types-update:
  update-defined-types (λ-. cake-all-types ∪ defined-types empty-state) empty-state
= empty-state-with-types
unfolding empty-state-with-types-def empty-state-def
by (simp add: datatype-record-update)

private lemma env-types-update: build-tdefs [] as-cake-type-def = as-static-cenv
unfolding as-cake-type-def-def as-static-cenv-def build-tdefs-def alist-to-ns-def flat-C-info-def
apply (auto simp: List.bind-def map-concat)
apply (rule arg-cong[where f = concat])
by (auto simp: map-concat comp-def split-beta)

private lemmas evaluate-type =
  evaluate-dec.dtype1 |
  where new-tdecs = cake-all-types and s = empty-state and mn = [] and tds
= as-cake-type-def,
  unfolded state-types-update env-types-update,
  folded empty-sem-env-def]

private lemma type-defs-to-new-tdecs:
  type-defs-to-new-tdecs [] as-cake-type-def =
  set (map (λname. TType (Short (as-string name))) (sorted-list-of-fset (fdom C-info)))
unfolding cake-all-types type-defs-to-new-tdecs-def as-cake-type-def-def all-tdefs-def
by (simp add: case-prod-twice sorted-list-of-fmap-def)

private lemma cakeml-convoluted1: foldr (λ(n, ts). (#) n) ys xs = map fst ys @ xs
by (induction ys arbitrary: xs) auto
private lemma cakeml-convoluted\textsubscript{2}: \texttt{foldr } (\lambda x y. f x \diamond y) \texttt{ xs ys} = \texttt{concat} (\texttt{map} f \texttt{ xs}) @ \texttt{ys} \\
by \ (\text{induction} \texttt{xs} \text{ arbitrary}: \texttt{ys}) \ \texttt{auto}

private lemma check-dup-ctors-alt-def: \texttt{check-dup-ctors} \texttt{ tds} \longleftrightarrow \texttt{distinct} (\texttt{tds} \geq (\lambda(\cdot, \cdot, \texttt{cons}). \texttt{map} \texttt{fst} \texttt{cons})) \\
unfolding \texttt{check-dup-ctors-def} \\
apply \texttt{simp} \\
apply \ (\text{rule arg-cong[where} f = \texttt{distinct]}) \\
apply \ (\text{subst} \texttt{foldr-cong}\ [\texttt{OF} \texttt{refl} \texttt{refl}, \texttt{where} g = \lambda x a. \texttt{map} \texttt{fst} (\texttt{snd} (\texttt{snd} x)) @ a]) \\
subgoal \\
apply \ (\text{subst} \texttt{split-beta}) \\
apply \ (\text{subst} \texttt{split-beta}) \\
by \ (\text{rule} \texttt{cakeml-convoluted1}) \\
subgoal \\
apply \ (\text{subst} \texttt{cakeml-convoluted2}) \\
apply \ \texttt{auto} \\
unfolding \texttt{List.bind-def} \\
apply \ (\text{rule arg-cong[where} f = \texttt{concat]}) \\
by \ \texttt{auto} \\
done

lemma evaluate-dec-prelude: \\
\text{evaluate-dec} \ t \ [] \ \texttt{env} \ \texttt{empty-state} \ \texttt{cake-dt-prelude} (\texttt{empty-state-with-types, Real empty-sem-env}) \\
unfolding \texttt{cake-dt-prelude-def} \\
proof \ (\text{rule evaluate-type, intro conjI}) \\
show \texttt{check-dup-ctors} \texttt{ as-cake-type-def} \\
using \texttt{distinct-ctr'} \\
unfolding \texttt{check-dup-ctors-alt-def} \texttt{List.bind-def as-cake-type-def-def all-constructors-def} \\
by \ (\texttt{auto simp: comp-def split-beta map-concat}) \\
next \\
show \texttt{allDistinct} (\texttt{map} (\lambda x. \texttt{case} \texttt{x} \texttt{of} (\texttt{tvs}, \texttt{tn}, \texttt{ctors}) \Rightarrow \texttt{tn}) \texttt{as-cake-type-def}) \\
unfolding \texttt{all-distinct-alt-def} \texttt{as-cake-type-def-def} \\
apply \ (\texttt{auto simp: comp-def case-prod-twice}) \\
apply \ (\texttt{rule name-as-string.fst-distinct}) \\
unfolding \texttt{sorted-list-of-fmap-def} \\
by \ (\texttt{auto simp: comp-def}) \\
next \\
show \texttt{cake-all-types} = \texttt{type-defs-to-new-tdecs} \ [] \texttt{as-cake-type-def} \\
unfolding \texttt{cake-all-types-def} \texttt{type-defs-to-new-tdecs all-tdefs-def} \\
by \ \texttt{simp} \\
next \\
show \texttt{disjnt} \texttt{cake-all-types} (\texttt{defined-types} \texttt{empty-state}) \\
unfolding \texttt{empty-state-def disjnt-def} \by \ \texttt{simp} \\
qed
Computability

\texttt{declare constructors.as-static-cenv-def[code]}
\texttt{declare constructors.as-cake-type-def-def[code]}
\texttt{declare constructors.cake-dt-prelude-def[code]}
\texttt{export-code constructors.as-static-cenv constructors.cake-dt-prelude}
\texttt{checking Scala}

end

\section{CakeML backend}

theory \textit{CakeML-Backend}

\texttt{imports}
\texttt{CakeML-Setup}
\texttt{../Terms/Value}
\texttt{../Rewriting/Rewriting-Sterm}

\begin{itemize}
\item 5.3.1 Compilation
\end{itemize}

fun \texttt{mk-ml-pat :: pat \Rightarrow Ast.pat where}
\texttt{mk-ml-pat (Patvar s) = Ast.Pvar (as-string s) |}
\texttt{mk-ml-pat (Patconstr s args) = Ast.Pcon (Some (Short (as-string s))) (map mk-ml-pat args)}

\texttt{lemma mk-pat-cupcake[intra]: is-cupcake-pat (mk-ml-pat pat)
by (induct pat) (auto simp: list-all-iff)}

context begin

private fun \texttt{frees': :: term \Rightarrow name list where}
\texttt{frees' (Free x) = [x] |}
\texttt{frees' (t_1 \& t_2) = frees' t_2 \&@ frees' t_1 |}
\texttt{frees' (\Lambda t) = frees' t |}
\texttt{frees' - = []}

\texttt{private lemma frees'-eq[simp]: fset-of-list (frees' t) = frees t
by (induction t) auto}

\texttt{private lemma frees'-list-comb: frees' (list-comb f xs) = concat (rev (map frees' xs)) \&@ frees' f
by (induction xs arbitrary: f) (auto simp: app-term-def)}

\texttt{private lemma frees'-distinct: linear pat \Rightarrow distinct (frees' pat)}
proof (induction pat)
case (App t u)
  hence distinct (frees' u @ frees' t)
    by (fastforce intro; distinct-append-fset fdisjnt-swap)
thus ?case
  by simp
qed auto

private fun pat-bindings' :: Ast.pat => name list where
  pat-bindings' (Ast.Pvar n) = [Name n] |
  pat-bindings' (Ast.Pcon - ps) = concat (rev (map pat-bindings' ps)) |
  pat-bindings' (Ast.Pref p) = pat-bindings' p |
  pat-bindings' (Ast.Pannot p -) = pat-bindings' p |
  pat-bindings' - = []

private lemma pat-bindings'-eq:
  map Name (pats-bindings ps xs) = concat (rev (map pat-bindings' ps)) @ map Name xs
  map Name (pat-bindings p xs) = pat-bindings' p @ map Name xs
by (induction ps xs and p xs rule: pats-bindings-pat-bindings.induct) (auto simp: ac-simps)

private lemma pat-bindings'-empty-eq: map Name (pat-bindings p []) = pat-bindings' p
by (simp add: pat-bindings'-eq)

private lemma pat-bindings'-eq-frees: linear p => pat-bindings' (mk-ml-pat (mk-pat p)) = frees' p
proof (induction rule: mk-pat.induct)
case (1 t)
show ?case
  using (linear t) proof (cases rule: linear-strip-comb-cases)
    case (comb s args)
    have map (pat-bindings' o mk-ml-pat o mk-pat) args = map frees' args
      proof (rule list.map-cong0, unfold comp-apply)
        fix x
        assume x ∈ set args
        moreover hence linear x
        using 1 comb by (metis linears-linear linears-strip-comb snd-conv)
      ultimately show pat-bindings' (mk-ml-pat (mk-pat x)) = frees' x
        using 1 comb by auto
    qed

    hence concat (rev (map (pat-bindings' o mk-ml-pat o mk-pat) args)) = concat (rev (map frees' args))
      by metis
    with comb show ?thesis
      apply (fold const-term-def)
apply (auto simp: strip-list-comb-const frees'-list-comb comp-assoc)
apply (unfold const-term-def)
apply simp
done
qed auto

lemma mk-pat-distinct: linear pat ⇒ distinct (pat-bindings (mk-ml-pat (mk-pat pat)) [])
by (metis pat-bindings'-eq-frees pat-bindings'-empty-eq frees'-distinct distinct-map)

end

locale cakeml =
begin

fun mk-exp :: name fset ⇒ sterm ⇒ exp and
mk-clauses :: name fset ⇒ (term × sterm) list ⇒ (Ast.pat × exp) list and
mk-con :: name fset ⇒ sterm ⇒ exp where

mk-exp - (Svar s) = Ast.Var (Short (as-string s))

| mk-exp - (Sconst s) = (if s |∈| C then Ast.Con (Some (Short (as-string s))) [] else Ast.Var (Short (as-string s))) |

| mk-exp S (t₁ $ₘ t₂) = Ast.App Ast.Opapp [mk-con S t₁, mk-con S t₂] |

| mk-exp S (Sabs cs) = (let n = fresh-fNext S in Ast.Fun (as-string n) (Ast.Mat (Ast.Var (Short (as-string n)))) (mk-clauses S cs)) |

| mk-con S t =
  (case strip-comb t of
    (Sconst c, args) ⇒
      if c |∈| C then Ast.Con (Some (Short (as-string c))) (map (mk-con S) args)
    else mk-exp S t |
    | - ⇒ mk-exp S t) |

| mk-clauses S cs = map (λ(pat, t). (mk-ml-pat (mk-pat pat), mk-con (frees pat |∪| S) t)) cs

context begin

private lemma mk-exp-cupcake0:
  wellformed t ⇒ is-cupcake-exp (mk-exp S t)

[wellformed-clauses cs ⇒ cupcake-clauses (mk-clauses S cs) ∧ cake-linear-clauses (mk-clauses S cs)]]

proof (induction rule: mk-exp-mk-clauses-mk-con.induct)
case (5 S t)
show ?case
  apply (simp split!: prod.splits sterm.splits if-split) | subgoal premises prems for args c

232
proof
  from prems have t = c $\$ args
  apply (fold const-sterm-def)
  by (metis fst-conv list-strip-comb snd-conv)
show ?thesis
  apply (auto simp: list-all-iff simp del: mk-con.simps)
  apply (rule 5(1))
  apply (rule prems(1)[symmetric])
  apply (rule refl)
  apply (rule prems)
  apply assumption
  using ⟨wellformed t⟩ ⟨t = −⟩
  apply (auto simp: wellformed list-comb list-all-iff)
  done
qed
using 5 by (auto split: prod.splits sterm.splits)
qed (auto simp: Let-def list-all-iff intro: mk-pat-distinct)
declare mk-con.simps[simp del]

lemma mk-exp-cupcake:
  wellformed t $\Rightarrow$ is-cupcake-exp (mk-exp S t)
  wellformed t $\Rightarrow$ is-cupcake-exp (mk-con S t)
by (metis mk-exp-cupcake0)+
end

definition mk-letrec-body where
  mk-letrec-body S rs = (
    map (λ(name, rhs).
      (as-string name, (let n = fresh-fNext S in
        (as-string n, Ast.Mat (Ast.Var (Short (as-string n))) (mk-clauses S (sterm.clauses rhs)))))) rs)
)
definition compile-group :: name fset $\Rightarrow$ srule list $\Rightarrow$ Ast.dec where
  compile-group S rs = Ast.Dletrec empty-locs (mk-letrec-body S rs)
definition compile :: srule list $\Rightarrow$ Ast.prog where
  compile rs = [Ast.Tdec (compile-group all-consts rs)]
end
declare cakeml.mk-con.simps[code]
declare cakeml.mk-exp.simps[code]
declare cakeml.mk-clauses.simps[code]
declare cakeml.mk-letrec-body-def [code]
declare cakeml.compile-group-def [code]
**5.3.2 Computability**

```
declare cakeml.compile-def [code]
locale cakeml' = cakeml + constants
context srules begin
sublocale srules-as-cake?: cakeml' C-info fst |' | fset-of-list rs by standard

lemma mk-letrec-cupcake:
  list-all (λ(_, _, exp). is-cupcake-exp exp) (mk-letrec-body S rs)
unfolding mk-letrec-body-def
using all-rules
apply (auto simp: Let-def list-all-iff intro: mk-pat-cupcake mk-exp-cupcake mk-pat-distinct)
subgoal for a b
  apply (erule ballE[where x = (a, b)]; cases b)
  apply (auto simp: list-all-iff is-abs-def term-cases-def)
done
subgoal for a b
  apply (erule ballE[where x = (a, b)]; cases b)
  apply (auto simp: list-all-iff is-abs-def term-cases-def)
done
done

end

definition compile' where
compile' C-info rs = cakeml.compile C-info (fst |' | fset-of-list rs) rs

lemma (in srules) compile'-compile-eq: compile' C-info rs = compile rs
unfolding compile'-def ..
```

**5.3.3 Correctness of semantic functions**

```
abbreviation related-pat :: term ⇒ Ast.pat ⇒ bool where
related-pat t p ≡ (p = mk-ml-pat (mk-pat t))
context cakeml' begin

inductive related-exp :: sterm ⇒ exp ⇒ bool where
var: related-exp (Svar name) (Ast.Var (Short (as-string name))) |
const: name ⋄ C ⇒ related-exp (Sconst name) (Ast.Var (Short (as-string name))) |
constr: name ⋄ C ⇒ list-all2 related-exp ts es ⇒
  related-exp (name §§ ts) (Ast.Con (Some (Short (as-string name))) es) |
```
lemma related-exp-is-cupcake:
assumes related-exp t e wellformed t
shows is-cupcake-exp e
using assms proof induction

next

next

\textbf{case} (\texttt{constr name ts es})

\textbf{hence} list-all wellformed ts
  \textbf{by} (simp add: wellformed.list-comb)

\textbf{with} :\text{list-all2} - ts es; \textbf{have} list-all is-cupcake-exp es
  \textbf{by} induction auto

\textbf{thus} ?case
  \textbf{by} simp

\textbf{qed} auto

\textbf{definition} related-fun :: (\texttt{term} \times \texttt{sterm}) list \Rightarrow \texttt{name} \Rightarrow \texttt{exp} \Rightarrow \texttt{bool}
\textbf{where}
related-fun cs n e \iff
\begin{align*}
  n |\notin| ids (\text{Sabs} cs) & \land n |\notin| \text{all-consts} \land (\text{case} e \text{ of} \\
  \text{Ast.Mat} (\text{Ast.Var} (\text{Short} n')) \text{ ml-cs}) \Rightarrow \\
  n = \text{Name} n' \land \text{list-all2} (\text{rel-prod related-pat related-exp}) \text{ cs ml-cs} \\
  | - \Rightarrow \text{False}
\end{align*}

\textbf{lemma} related-fun-alt-def:
related-fun cs n (\text{Ast.Mat} (\text{Ast.Var} (\text{Short} (\text{as-string} n)))) \text{ ml-cs} \iff
\begin{align*}
  \text{list-all2} (\text{rel-prod related-pat related-exp}) \text{ cs ml-cs} \land \\
  n |\notin| ids (\text{Sabs} cs) \land n |\notin| \text{all-consts}
\end{align*}

\textbf{unfolding} related-fun-def
\textbf{by} auto

\textbf{lemma} related-funE:
\begin{align*}
  \text{assumes} \ & \text{related-fun} \text{ cs n e} \\
  \text{obtains} \ & \text{ml-cs} \\
  \text{where} \ & e = \text{Ast.Mat} (\text{Ast.Var} (\text{Short} (\text{as-string} n))) \text{ ml-cs} \land n |\notin| ids (\text{Sabs} cs) \\
  \text{and} \ & \text{list-all2} (\text{rel-prod related-pat related-exp}) \text{ cs ml-cs}
\end{align*}
\textbf{using} \text{assms} \textbf{unfolding} related-fun-def
\textbf{by} (simp split: exp0.splits id0.splits)

\textbf{lemma} related-exp-fun:
related-fun cs n e \iff related-exp (\text{Sabs cs}) (\text{Ast.Fun} (\text{as-string} n) e) \land n |\notin|
\begin{align*}
  \text{ids} (\text{Sabs cs}) \land n |\notin| \text{all-consts} \\
  \{ \text{is} \ ?\text{lhs} \leftrightarrow ?\text{rhs} \}
\end{align*}
\textbf{proof}
\begin{align*}
  \text{assume} \ & ?\text{rhs} \\
  \text{hence} \ & \text{related-exp} (\text{Sabs cs}) (\text{Ast.Fun} (\text{as-string} n) e) \text{ by simp} \\
  \text{thus} \ & ?\text{lhs} \\
  \text{by} \ & \text{cases} (\text{auto simp: related-fun-def dest: name.expand})
\end{align*}

\textbf{next}
\begin{align*}
  \text{assume} \ & ?\text{lhs} \\
  \text{thus} \ & ?\text{rhs} \\
  \text{by} \ & (\text{auto intro: related-exp.fun elim: related-funE})
\end{align*}
\textbf{qed}

\textbf{inductive} related-v :: \texttt{value} \Rightarrow \texttt{v} \Rightarrow \texttt{bool} \textbf{where}

236
conv:
list-all2 related-v us vs ⇒
related-v (Vconstr name us) (Conv (Some (as-string name, -)) vs) |
closure:
related-fun cs n e ⇒
fmrel-on-fset (ids (Sabs cs)) related-v Γ (fmap-of-ns (sem-env.v env)) ⇒
related-v (Vabs cs Γ) (Closure env (as-string n) e) |
rec-closure:
fmrel-on-fset (fbind (fmran css) (ids ◦ Sabs)) related-v Γ (fmap-of-ns (sem-env.v env)) ⇒
fmrel (λcs. λ(n, e). related-fun cs n e) css (fmap-of-list (map (map-prod Name (map-prod Name id)) exps)) ⇒
related-v (Vrecabs css name Γ) (Reclosure env exps (as-string name))

abbreviation var-env :: (name, value) fmap ⇒ (string × v) list ⇒ bool where
var-env Γ ns ≡ fmrel related-v Γ (fmap-of-list (map (map-prod Name id) ns))

lemma related-v-ext:
assumes related-v v ml-v
assumes v′ ≈_e v
shows related-v v′ ml-v
using assms proof (induction arbitrary: v′)
case (conv us ml-us name)
obtain ts where v′ = Vconstr name ts list-all2 erelated ts us
using (v′ ≈_e Vconstr name us)
by cases auto
have list-all2 related-v ts ml-us
by (rule list-all2-trans[OF - ⟨list-all2 erelated ts us⟩ conv(1)]) auto
thus ?case
using conv unfolding ⟨v′ = -⟩
by (auto intro: related-v.conv)

next
case (closure cs n e Γ_2 env)
obtain Γ_1 where v′ = Vabs cs Γ_1 fmrel-on-fset (ids (Sabs cs)) erelated Γ_1 Γ_2
using (v′ ≈_e -)
by cases auto
have fmrel-on-fset (ids (Sabs cs)) related-v Γ_1 (fmap-of-ns (sem-env.v env))
apply rule
subgoal premises prems for x
apply (insert prems)
apply (drule fmrel-on-fsetD)
apply (rule closure)
subgoal
apply (insert prems)
apply (drule fmrel-on-fsetD)
apply (rule (fmrel-on-fset (ids (Sabs cs)) erelated Γ_1 Γ_2))
apply (metis (mono-tags, lifting) option.rel-sel)
done
done
done

show ?case
  unfolding ⟨v' = -⟩
  by (rule related-v.closure) fact+
next
  case (rec-closure css Γ_2 env exps name)
  obtain Γ_1 where v' = Vrecabs css name Γ_1 pred-fmap (λcs. fmrel-on-fset (ids (Sabs cs)) related Γ_1 Γ_2) css
    using ⟨v' ≈e -⟩
  by cases auto

  have fmrel-on-fset (fbind (fmran css) (ids ∘ Sabs)) related-v Γ_1 (fmap-of-ns (sem-env.v env))
    apply (rule fmrel-on-fsetI)
  subgoal premises prems for x
    apply (insert prems)
    apply (drule fmrel-on-fsetD)
    apply (rule rec-closure)
  subgoal
    apply (insert prems)
    apply (erule fbindE)
    apply (drule pred-fmapD[OF 'pred-fmap - css'])
    unfolding comp-apply
    apply (drule fmrel-on-fsetD)
    apply assumption
    apply (metis (mono-tags, lifting) option.rel-sel)
  done
done
done

show ?case
  unfolding ⟨v' = -⟩
  by (rule related-v.rec-closure) fact+
qed

context begin

private inductive match-result-related :: (string × v) list ⇒ (string × v) list
  match-result ⇒ (name, value) fmap option ⇒ bool for eenv where
  no-match: match-result-related eenv No-match None |
  error: match-result-related eenv Match-type-error - |
  match: var-env Γ eenv-m ⟷ match-result-related eenv (Match (eenv-m @ eenv))
  (Some Γ)

private corollary match-result-related-empty: match-result-related eenv (Match
proof –
  have match-result-related eenv (Match ([] @ eenv)) (Some fnempty)
    by (rule match-result-related.match) auto
  thus ?thesis
    by simp
qed

private fun is-Match :: 'a match-result ⇒ bool where
  is-Match (Match _) ⟷ True |
  is-Match _ ⟷ False

lemma cupcake-pmatch-related:
  assumes related-v v ml-v
  shows match-result-related eenv (cupcake-pmatch as-static-cenv (mk-ml-pat pat) ml-v eenv) (vmatch pat v)
  using assms proof (induction pat arbitrary: v ml-v eenv)
  case (Patvar name)
    have var-env (fmap-of-list [(name, v)]) [(as-string name, ml-v)]
      using Patvar by auto
    hence match-result-related eenv (Match [(as-string name, ml-v)] @ eenv)) (Some (fmap-of-list [(name, v)]))
      by (rule match-result-related.match)
    thus ?case
      by simp
next
  case (Patconstr name ps v0)
  show ?case
    using Patconstr.prems
  proof (cases rule: related-v.cases)
    case (conv us vs name)
    define f where
      f p v m =
        (case m of
          Match env ⇒ cupcake-pmatch as-static-cenv p v env
          | m ⇒ m) for p v m
    {
      assume name = name'
      assume length ps = length us
      hence list-all2 (λ- _. True) us ps
        by (induct rule: list-induct2) auto
      hence list-all3 (λ p v. related-v t v) us ps vs
        using list-all2 related-v us vs;
        by (rule list-all3-from-list-all2s) auto
    }
hence \( \ast \): match-result-related eenv

\[
(\text{Matching.fold2 } f \ (\text{Match-type-error} \ (\text{map mk-ml-pat } ps) \ \text{vs} \ (\text{Match } (\text{eenv-m } @ \text{ eenv}))))
\]

\[\text{(map-option } (\text{foldl } (++f) \ 
\Gamma) \ (\text{those } (\text{map2 } \text{vmatch } \text{ps } \text{us}))))\]

(is ?rel)

if var-env \( \Gamma \) eenv-m

for eenv-m \( \Gamma \)

using Patconstr.IH \( \langle \text{related-v } v \in \text{ml-v} \rangle \)

proof (induction us ps vs arbitrary; \( \Gamma \) eenv-m rule: list-all3-induct)

case (Cons t us p ps v vs)

have match-result-related (eenv-m @ eenv) (cupcake-pmatch as-static-eenv (mk-ml-pat p) v (eenv-m @ eenv)) (vmatch p t)

using Cons by simp

thus ?case

proof cases

  case no-match

  thus ?thesis

  unfolding f-def

  apply (cases length (\text{map mk-ml-pat } ps) = length vs)

  by (fastforce intro: match-result-related.intros simp: cup-pmatch-list-nomatch cup-pmatch-list-length-neq)+

  next

  case error

  thus ?thesis

  unfolding f-def

  apply (cases length (\text{map mk-ml-pat } ps) = length vs)

  by (fastforce intro: match-result-related.intros simp: cup-pmatch-list-typerr cup-pmatch-list-length-neq)+

  next

  case (match \( \Gamma' \) eenv-m')

have match-result-related eenv

\[
(\text{Matching.fold2 } f \ (\text{Match-type-error} \ (\text{map mk-ml-pat } ps) \ \text{vs} \ (\text{Match } ((\text{eenv-m'} \ @ \atom{eenv-m'}) \ @ \text{ eenv}))))
\]

\[\text{(map-option } (\text{foldl } (++f) \ (\Gamma' \oplus \oplus f \ \Gamma')) \ (\text{those } (\text{map2 } \text{vmatch } \text{ps } \text{us}))))\]

proof (rule Cons, rule Cons)

  show var-env (\( \Gamma' \oplus f \) \( \Gamma' \)) (eenv-m' @ eenv-m)

  using var-env \( \Gamma \) eenv-m' match

  by force

  qed (simp | fact)+

thus ?thesis

using match

unfolding f-def

by (auto simp: map-option.compositionality comp-def)

qed
moreover have var-env fmempty []
  by force

ultimately have ?rel [ ] fmempty
  by fastforce

hence ?thesis
  using conv (length ps = length us)
unfolding f-def \{name = name'\}
by (auto intro: match-result-related.intros split: option.splits elim: static-cenv-lookup)
}
moreover
{
  assume name ≠ name'
  with conv have ?thesis
  by (auto intro: match-result-related.intros split: option.splits elim: same-ctor.elims
  simp: name.expand)
}
moreover
{
  let ?fold = Matching.fold2 f Match-type-error (map mk-ml-pat ps) vs (Match eenv)

  assume *: length ps ≠ length us
  moreover have length us = length vs
    using (list-all2 related-v us vs) by (rule list-all2-lengthD)
  ultimately have length ps ≠ length vs
    by simp

  moreover have ¬ is-Match (Matching.fold2 f err xs ys init)
    if ¬ is-Match err and length xs ≠ length ys for init err xs ys
      using that f-def
    by (induct f err xs ys init rule: fold2.induct) (auto split: match-result.splits)
  ultimately have ¬ is-Match ?fold
    by simp
  hence ?fold = Match-type-error ∨ ?fold = No-match
    by (cases ?fold) auto

  with * have ?thesis
    unfolding ml-v = ¬ (v0 = ¬) f-def
    by (auto intro: match-result-related.intros split: option.splits)
}

ultimately show ?thesis
  by auto
qed (auto intro: match-result-related.intros)
qed
lemma match-all-related:
assumes list-all2 (rel-prod related-pat related-exp) cs ml-cs
assumes list-all (\lambda (pat, -). linear pat) cs
assumes related-v v ml-v
assumes cupcake-match-result as-static-cenv ml-v ml-cs Bindv = Rval (ml-rhs, ml-pat, eenv)
obtains rhs pat \Gamma where
ml-pat = mk-ml-pat (mk-pat pat)
related-exp rhs ml-rhs
vfind-match cs v = Some (\Gamma, pat, rhs)
var-env \Gamma eenv
using assms
proof (induction cs ml-cs arbitrary: thesis ml-pat ml-rhs rule: list-all2-induct)
case (Cons c cs ml-c ml-cs)
moreover obtain pat0 rhs0 where c = (pat0, rhs0) by fastforce
moreover obtain ml-pat0 ml-rhs0 where ml-c = (ml-pat0, ml-rhs0) by fastforce
ultimately have ml-pat0 = mk-ml-pat (mk-pat pat0) related-exp rhs0 ml-rhs0
by auto
have linear pat0
using Cons(5) unfolding \langle c = - \rangle by simp+
have rel: match-result-related [] (cupcake-pmatch as-static-cenv (mk-ml-pat (mk-pat pat0)) ml-v []) (vmatch (mk-pat pat0) v)
by (rule cupcake-pmatch-related) fact+
show \case
proof (cases cupcake-pmatch as-static-cenv ml-pat0 ml-v [])
case Match-type-error
hence False
using Cons(7) unfolding \langle ml-c = - \rangle
by (simp split: if-splits)
thus thesis ..
next
case No-match
show thesis
proof (rule Cons(3))
show cupcake-match-result as-static-cenv ml-v ml-cs Bindv = Rval (ml-rhs, ml-pat, eenv)
using Cons(7) No-match unfolding \langle ml-c = - \rangle
by (simp split: if-splits)
next
fix pat rhs \Gamma
assume ml-pat = mk-ml-pat (mk-pat pat)
assume related-exp rhs ml-rhs
assume vfind-match cs v = Some (\Gamma, pat, rhs)
assume var-env \Gamma eenv
from rel have match-result-related [] No-match (vmatch (mk-pat pat0) v)
  using No-match unfolding :mk-pat0 = ·
  by simp
hence vmatch (mk-pat pat0) v = None
  by (cases rule: match-result-related.cases)

show thesis
proof (rule Cons(4))
  show vfind-match (c # cs) v = Some (Γ, pat, rhs)
    unfolding (c = ·)
      using vfind-match cs v = · (vmatch (mk-pat pat0) v = ·)
      by simp
  qed fact+
  next
  show list-all (λ(pat, -), linear pat) cs
    using Cons(5) by simp
  qed fact+
next
case (Match eenv')
hence ml-rhs = ml-rhs0 ml-pat = ml-pat0 eenv = eenv'
  using Cons(7) unfolding (ml-c = ·)
  by (simp split: if-splits)+

from rel have match-result-related [] (Match eenv') (vmatch (mk-pat pat0) v)
  using Match unfolding :mk-pat0 = · by simp
then obtain Γ where vmatch (mk-pat pat0) v = Some Γ var-env Γ eenv'
  by (cases rule: match-result-related.cases) auto

show thesis
proof (rule Cons(4))
  show ml-pat = mk-ml-pat (mk-pat pat0)
    unfolding (mk-pat = ·) by fact
  next
  show related-exp rhs0 ml-rhs
    unfolding (ml-rhs = ·) by fact
  next
  show var-env Γ eenv
    unfolding (eenv = ·) by fact
  next
  show vfind-match (c # cs) v = Some (Γ, pat0, rhs0)
    unfolding (c = ·)
      using vmatch (mk-pat pat0) v = Some Γ
      by simp
  qed
  qed simp

qed simp

243
5.3.4 Correctness of compilation

theory CakeML-Correctness

imports CakeML-Backend
../Rewriting/Big-Step-Value-ML

begin

context cakeml' begin

lemma mk-rec-env-related:
  assumes fmrel (\lambda cs (n, e). related-fun cs n e) css (fmmap-of-list (map (map-prod Name (map-prod Name id)) funs))
  assumes fmrel-on-fset (fbind (fmran css) (ids \circ Sabs)) related-v \Gamma_A (fmmap-of-ns (sem-env.v env_\Lambda))
  shows fmrel related-v (mk-rec-env css \Gamma \Lambda) (cake-mk-rec-env funs env_\Lambda)

proof (rule fmrelI)
  fix name
  have rel-option (\lambda cs (n, e). related-fun cs n e) (fmlookup css name) (map-of (map (map-prod Name (map-prod Name id)) funs) name)
    using assms by (auto simp: fmmap-of-list.rep-eq)

  then have rel-option (\lambda cs (n, e). related-fun cs (Name n) e) (fmlookup css name) (map-of funs (as-string name))
    unfolding name.map-of-rekey'
      by cases auto

  have *: related-v (Vrecabs css name \Gamma_A) (Recclosure env_\Lambda funs (as-string name))
    using assms by (auto intro: related-v.recclosure)

show rel-option related-v (fmlookup (mk-rec-env css \Gamma_A) name) (fmlookup (cake-mk-rec-env funs env_\Lambda) name)
  unfolding mk-rec-env-def cake-mk-rec-env-def fmmap-of-list.rep-eq
  apply (simp add: map-of-map-keyed name.map-of-rekey option.rel-map)
  apply (rule option.rel-mono-strong)
  apply fact
  apply (rule *)
  done

qed

lemma mk-exp-correctness:
  ids t \subseteq S \Longrightarrow all-consts \subseteq S \Longrightarrow \neg shadows-consts t \Longrightarrow related-exp t (mk-exp S t)
  ids (Sabs cs) \subseteq S \Longrightarrow all-consts \subseteq S \Longrightarrow \neg shadows-consts (Sabs cs) \Longrightarrow
  list-all2 (rel-prod related-pat related-exp) cs (mk-clauses S cs)
\[ \text{ids } t \subseteq S \implies \text{all-consts } \subseteq S \implies \neg \text{shadows-consts } t \implies \text{related-exp } t (\text{mk-con } S t) \]

**proof** (induction rule: mk-exp-mk-clauses-mk-con.induct)

**case** (2 S name)

**show** ?case

**proof** (cases name \[\in\] C)

**case** True

hence \text{related-exp} (name $$[]) (\text{mk-exp } S (\text{Sconst name}))

**by** (auto intro: related-exp.intros simp del: list-comb.simps)

thus ?thesis

**by** (simp add: const-sterm-def)

**qed** (auto intro: related-exp.intros)

**next**

**case** (4 S cs)

have \text{fresh-fNext} (S \cup \text{all-consts}) \notin S \cup \text{all-consts}

**by** (rule fNext-not-member)

hence \text{fresh-fNext} S \cup \text{all-consts}

**using** (\text{all-consts } \subseteq S) by (simp add: sup-absorb1)

hence \text{fresh-fNext} S \cup \text{ids} (\text{Sabs cs}) \cup \text{all-consts}

**using** 4 by auto

**show** ?case

**apply** (simp add: Let-def)

**apply** (rule related-exp.fun)

**apply** (rule 4.IH[unfolded mk-clauses.simps])

**apply** (rule refl)

**apply** fact+

**using** (fresh-fNext S \notin S \cup \text{all-consts} by auto

**next**

**case** (5 S t)

**show** ?case

**apply** (simp add: mk-con.simps split!: prod.splits sterm.splits if-splits)

**subgoal premises** prems for args c

**proof** –

from prems have \(t = c \llstenargs\)

**apply** (fold const-sterm-def)

**by** (metis fst-conv list-strip-comb snd-conv)

**show** ?thesis

unfolding (t = \_)

**apply** (rule related-exp.constr)

**apply** fact

**apply** (simp add: list.rel-map)

**apply** (rule list.rel-refl-strong)

**apply** (rule 5(1))

**apply** (rule prems(1)[symmetric])

**apply** (rule refl)

**subgoal** by (rule prems)

**subgoal** by assumption

245
subgoal
  using ⟨ids t |⊆| S⟩ unfolding ⟨t = -⟩
apply (auto simp: ids-list-comb)
by (meson ffUnion-subset-elem fimage-eqI fset-of-list-elem fset-rev-mp)
subgoal by (rule 5)
subgoal
  using ⟨¬ shadows-consts t⟩ unfolding ⟨t = -⟩
unfolding shadows.list-comb
by (auto simp: list-ex-iff)
done
qed
using 5 by (auto split: prod.splits sterm.splits)
next
case (6 S cs)

have list-all2 (λx y. rel-prod related-pat related-exp x (case y of (pat, t) ⇒ (mk-mt-pat (mk-pat pat), mk-con (frees pat |∪| S) t))) cs cs
proof (rule list.rel-refl-strong, safe intro!: rel-prod.intros)
fix pat rhs
assume (pat, rhs) ∈ set cs

hence consts rhs |⊆| S
using ⟨ids (Sabs cs) |⊆| S⟩
unfolding ids-def
apply auto
apply (drule ffUnion-least-rev)+
apply (auto simp: fset-of-list-elem elim!: fBallE)
done

have frees rhs |⊆| frees pat |∪| S
using ⟨ids (Sabs cs) |⊆| S⟩ (case y of (pat, rhs) ∈ set cs)
unfolding ids-def
apply auto
apply (drule ffUnion-least-rev)+
apply (auto simp: fset-of-list-elem elim!: fBallE)
done

have ¬ shadows-consts rhs
using ⟨(pat, rhs) ∈ set cs⟩ 6
by (auto simp: list-ex-iff)

show related-exp rhs (mk-con (frees pat |∪| S) rhs)
apply (rule 6)
apply fact
subgoal by simp
subgoal
unfolding ids-def
using ⟨consts rhs |⊆| S⟩ (frees rhs |⊆| frees pat |∪| S)
by auto
subgoal using $6(3)$ by fast
subgoal by fact
done
qed

thus ?case
  by (simp add: list.rel-map)
qed (auto intro: related-exp.intros simp: ids-def fdisjnt-alt-def)

context begin

private lemma semantic-correctness0:
  fixes exp
  assumes cupcake-evaluate-single env exp r is-cupcake-all-env env
  assumes fmrel-on-fset (ids t) related-v Γ (fmap-of-ns (sem-env.v env))
  assumes related-exp t exp
  assumes wellformed t wellformed-venv Γ
  assumes closed-venv Γ closed-except t (fdom Γ)
  assumes fmpred (λ- vwelldefined') Γ constst t $|$ fdom Γ $|$ C
  assumes fdisjnt C (fdom Γ)
  assumes ¬ shadows-consts t not-shadows-ven consts-env Γ
  shows if-rval (λ ml-v. ∃ v. Γ ⊢ v t $\downarrow$ v $\land$ related-v v ml-v) r
  using assms proof (induction arbitrary: Γ t)
  case (con1 env cn es ress ml-vs ml-v')
  obtain name ts where cn = Some (Short (as-string name)) name $|$ C t = name $$$ ts list-all2 related-exp ts es
    using (related-exp t (Con cn es));
  by cases auto
  with con1 obtain tid where ml-v' = Conv (Some (id-to-n (Short (as-string name)), tid)) (rev ml-vs)
    by (auto split: option.splits)
  have ress = map Rval ml-vs
  using con1 by auto
define ml-vs' where ml-vs' = rev ml-vs

note IH = (list-all2-shortcircuit - - -)
  unfolded (ress = \ -> ) list-all2-shortcircuit-real list-all2-rev1 ,
  folded ml-vs'-def

moreover have
  list-all wellformed ts list-all (λt. ¬ shadows-consts t) ts
  list-all (λt. constst t $|$ fdom Γ $|$ C) ts list-all (λt. closed-except t (fdom Γ)) ts
  subgoal
    using wellformed t; unfolding (t = \ -> ) wellformed.list-comb by simp
  subgoal
    using (¬ shadows-consts t); unfolding (t = \ -> ) shadows.list-comb
  by (simp add: list-all-iff list-ex-iff)
subgoal
using ⟨consts t \subseteq fmdom \Gamma \cup C⟩ unfolding list-all-iff
by (metis Ball-set \( t = \text{name \$$ ts} \) con1.prems(9) special-constants.sconsts-list-comb)

subgoal
using ⟨closed-except t (fmdom \Gamma)⟩ unfolding ⟨t = \_⟩ closed.list-comb by simp
done

moreover have
list-all (\lambda t'. fmrel-on-fset (\ids t') \related-v \Gamma (fmap-of-ns \sem-env.v env)) \ts
proof (standard, rule fmrel-on-fsubset)
fix t'
assume t' \in set ts
thus \ids t' \subseteq \ids t
unfolding \( t = \_ \)
apply (simp add: ids-list-comb)
apply (subst (2) ids-def)
apply simp
apply (rule fsubset-finsertI2)
apply (auto simp: fset-of-list-element intro: ffUnion-subset-elem)
done

show fmrel-on-fset (\ids t) \related-v \Gamma (fmap-of-ns \sem-env.v env))
by fact

qed

ultimately obtain \us where list-all2 (\veval' \Gamma) ts \us list-all2 related-v \us ml-vs'
using ⟨list-all2 related-exp \ts es⟩
proof (induction es ml-vs' arbitrary: \ts thesis rule: list.rel-induct)
case (Cons e es ml-v ml-vs ts0)
then obtain \ts where ts0 = t \# \ts related-exp t e by (cases ts0) auto
with Cons have list-all2 related-exp \ts es by simp
with Cons obtain \us where list-all2 (\veval' \Gamma) ts \us list-all2 related-v \us ml-vs
unfolding \( ts0 = \_ \)
by auto

from Cons.hyps[simplified, THEN conjunct2, rule-format, of t \Gamma]
obtain \u where \Gamma \vdash_v t \downarrow \u related-v \u ml-v
proof
show
  is-cupcake-all-env env related-exp t e wellformed-env \Gamma closed-env \Gamma
  fmnpred (\lambda_. uwelldefined) \Gamma \disjnt C (fmdom \Gamma)
  not-shadows-vconsts-env \Gamma
  by fact+
next
show
  wellformed t \neg shadows-consts t closed-except t (fmdom \Gamma)
 consts t \subseteq fmdom \Gamma \cup C fmrel-on-fset (\ids t) related-v \Gamma (fmap-of-ns \sem-env.v env))
using Cons unfolding \( ts0 = \_ \)
by auto
qed blast

show ?case
  apply (rule Cons(3)[of u # us])
  unfolding ts0 = -
  apply auto
  apply fact+
done
qed auto

show ?case
  apply simp
  apply (intro exI conjI)
  unfolding (t = -)
  apply (rule veval!.constr)
  apply fact+
  unfolding (ml-v' = -)
  apply (subst ml-vs'-def[symmetric])
  apply simp
  apply (rule related-v.conv)
  apply fact
done

next
case (var1 env id ml-v)

from (related-exp t (Var id)) obtain name where id = Short (as-string name)
  by cases auto
with var1 have cupcake-nsLookup (sem-env v env) (as-string name) = Some
  ml-v
  by auto

from (related-exp t (Var id)) consider
  (var) t = Svar name |
  (const) t = Sconst name name |\notin| C
  unfolding (id = -)
  apply (cases t)
  using name.expand by blast+
thus ?case
proof cases
  case var
  hence name |\in| ids t
  unfolding ids-def by simp

  have rel-option related-v (fmlookup Γ name) (cupcake-nsLookup (sem-env v
  env) (as-string name))
    using fmrel-on-fset (ids t) - - -
    apply
    apply (erule fmrel-on-fsetD[OF name |\in| ids t])
apply simp
done
then obtain v where related-v v ml-v fmlookup Γ name = Some v
  using ⟨cupcake-nsLookup (sem-env.v env) - = -⟩
by cases auto

show ?thesis
  unfolding ⟨t = ⟩
  apply simp
  apply (rule exI)
  apply (rule conjI)
  apply (rule veval',var)
  apply fact+
done
next
case const
hence name |∈| ids t
  unfolding ids-def by simp

have rel-option related-v (fmlookup Γ name) (cupcake-nsLookup (sem-env.v env) (as-string name))
  using ⟨fmrel-on-fset (ids t) - - ⟩
apply --
apply (drule fmrel-on-fsetD[OF ⟨name |∈| ids t⟩])
apply simp
done
then obtain v where related-v v ml-v fmlookup Γ name = Some v
  using ⟨cupcake-nsLookup (sem-env.v env) - = -⟩
by cases auto

show ?thesis
  unfolding ⟨t = ⟩
  apply simp
  apply (rule exI)
  apply (rule conjI)
  apply (rule veval',const)
  apply fact+
done
qed
next
case (fn env n u)
obtain n′ where n = as-string n'
  by (metis name.sel)
obtain cs ml-cs
  where t = Sabs cs u = Mat (Var (Short (as-string n′))) ml-cs n′ |∈| ids (Sabs cs) n′ |∈| all-consts
  and list-all2 (rel-prod related-pat related-exp) cs ml-cs
using ⟨related-exp t (Fun n u)⟩ unfolding ⟨n = ⟩
by cases (auto dest: name.expand)
obtain ns where fmap-of-ns (sem-env.v env) = fmap-of-list ns
apply (cases env)
apply simp
subgoal for v by (cases v) simp
done

show ?case
apply simp
apply (rule exI)
apply (rule conjI)
unfolding ⟨t = t⟩
apply (rule veval′.abs)
unfolding ⟨n = n⟩
apply (rule related-v.closure)
unfolding ⟨u = u⟩
apply (rule subst related-fun-alt-def; rule conjI)
apply fact
using (frrel-on-fset (ids t) - - -)
unfolding ⟨fmap-of-ns (sem-env.v env) = t⟩
by simp

next
case (app1 env exps ress ml-vs env′ exp′ bv)
from ⟨related-exp t -⟩ obtain exp1 exp2 t1 t2
  where rev exps = [exp2, exp1] exps = [exp1, exp2] t = t1 $s, t2
  and related-exp t1 exp1 related-exp t2 exp2
by cases auto
moreover from app1 have ress = map Real ml-vs
by auto
ultimately obtain ml-v1 ml-v2 where ml-vs = [ml-v2, ml-v1]
using app1(1)
by (smt list-all2-shortcircuit-real list-all2-Cons1 list-all2-Nil)

have is-cupcake-exp exp1 is-cupcake-exp exp2
using app1 unfolding (exps = -) by (auto dest: related-exp-is-cupcake)
moreover have fnrel-on-fset (ids t1) related-v Γ (fmap-of-ns (sem-env.v env))
using app1 unfolding ids-def (t = -)
by (auto intro: fnrel-on-fsubset)
moreover have fnrel-on-fset (ids t2) related-v Γ (fmap-of-ns (sem-env.v env))
using app1 unfolding ids-def (t = -)
by (auto intro: fnrel-on-fsubset)
ultimately have
cupcake-evaluate-single env exp1 (Real ml-v1) cupcake-evaluate-single env exp2
(Real ml-v2) and
∃ t1′, Γ ‖ v t1 ‖ t1′ ∧ related-v t1′ ml-v1 t2′, Γ ‖ v t2 ‖ t2′ ∧ related-v t2′ ml-v2
using app1 (related-exp t1 exp1) (related-exp t2 exp2)
unfolding ⟨ress = -⟩ ⟨exps = -⟩ ⟨ml-vs = -⟩ ⟨t = -⟩
by (auto simp: closed-except-def)
then obtain $v_1 v_2$
where $\Gamma \vdash v_1 \downarrow v_1$ \text{related-$v$ $v_1$ $ml$-$v_1$}
and $\Gamma \vdash v_2 \downarrow v_2$ \text{related-$v$ $v_2$ $ml$-$v_2$}
by blast

have \text{is-cupcake-value $ml$-$v_1$}
by (rule \text{cupcake-single-preserve}) \text{ fact+}
moreover have \text{is-cupcake-value $ml$-$v_2$}
by (rule \text{cupcake-single-preserve}) \text{ fact+}
ultimately have \text{list-all is-cupcake-value (rev $ml$-$vs$)}
unfolding \langle ml$-$vs$ = $\cdot$ \rangle by simp

hence \text{is-cupcake-exp exp$'$ is-cupcake-all-env env$'$}
using \langle do-opapp $=$ $\cdot$ \rangle by \text{(metis \text{cupcake-opapp-preserve})+}

have vclosed $v_1$
proof (rule \text{veval$'$-closed})
show \text{wellformed $t_1$ \langle \text{fmdom $\Gamma$} \rangle}
using \langle \text{wellformed $t$ \cdot \text{unfolding} \langle \langle \rangle \rangle} \\text{by simp}
next
show \text{wellformed $t_1$}
using \langle \text{wellformed $t$} \rangle unfolding \langle \langle \rangle \rangle by simp
qed fact+

have vclosed $v_2$
apply (rule \text{veval$'$-closed})
apply fact
using \langle do-opapp $=$ $\cdot$ \rangle by (auto simp: \text{closed-except-def})

have vvwellformed $v_1$
apply (rule \text{veval$'$-wellformed})
apply fact
using \langle do-opapp $=$ $\cdot$ \rangle by auto
have vvwellformed $v_2$
apply (rule \text{veval$'$-wellformed})
apply fact
using \langle do-opapp $=$ $\cdot$ \rangle by auto

have vvwelldefined$'$ $v_1$
apply (rule \text{veval$'$-welldefined$'$})
apply fact
using \langle do-opapp $=$ $\cdot$ \rangle by auto
have vvwelldefined$'$ $v_2$
apply (rule \text{veval$'$-welldefined$'$})
apply fact
using \langle do-opapp $=$ $\cdot$ \rangle by auto
have not-shadows-vconsts \( v_1 \)
apply (rule veval'-shadows)
apply fact
using app1 unfolding \( t = \cdot \) by auto
have not-shadows-vconsts \( v_2 \)
apply (rule veval'-shadows)
apply fact
using app1 unfolding \( t = \cdot \) by auto

show ?case
proof (rule if-rvalI)
fix \( ml-v \)
assume \( bv = Rval ml-v \)
show \( \exists v. \Gamma \vdash v \downarrow v \land related-v v ml-v \)
using (do-opapp - = \cdot)
proof (cases rule: do-opapp-cases)
case (closure env \( \Lambda \) \( n \))
then have closure':
\( ml-v_1 = Closure env_{\Lambda} (as-string (Name n)) exp' \)
\( env' = update-v (\lambda \cdot nsBind (as-string (Name n)) ml-v_2 (sem-env.v env_{\Lambda})) \)
unfolding (ml-vs = \cdot) by auto
obtain \( \Gamma_{\Lambda} cs \) where \( v_1 = Vabs cs \Gamma_{\Lambda} related-fun cs (Name n) exp' \)
and fnrel-on-fset (ids (Sabs cs)) related-v \( \Gamma_{\Lambda} (fnmap-of-ns (sem-env.v env_{\Lambda})) \)
using (related-v v_1 ml-v_1) unfolding (ml-v_1 = \cdot) by cases auto
then have ml-cs
where exp' = Mat (Var (Short (as-string (Name n)))) ml-cs Name n
\( |\xi| ids (Sabs cs) Name n |\xi| all-consts \)
and list-all2 (rel-prod related-pat related-exp) cs ml-cs
by (auto elim: related-funE)

hence cupcake-evaluate-single env' (Mat (Var (Short (as-string (Name n)))) ml-cs (Real ml-v))
using (cupcake-evaluate-single env' exp' bv)
unfolding (bv = \cdot) by simp
then obtain m-env v' ml-rhs ml-pat
where cupcake-evaluate-single env' (Var (Short (as-string (Name n)))) (Real v')
and cupcake-match-result (sem-env.c env') v' ml-cs Bindv = Real (ml-rhs, ml-pat, m-env)
and cupcake-evaluate-single (env' [] sem-env.v := nsAppend (alist-to-ns m-env) (sem-env.v env' [])) ml-rhs (Real ml-v)
by cases auto

have
  closed-venv (fmupd (Name n) v₂ Γₐ) wellformed-venv (fmupd (Name n) v₂ Γₐ)
  not-shadows-vconsts-env (fmupd (Name n) v₂ Γₐ) fmupd (λ _. vwelldefined') (fmupd (Name n) v₂ Γₐ)
  using (vclosed v₁) (vwellformed v₂)
  using (vwellformed v₁) (vwellformed v₂)
  using (not-shadows-vconsts v₁) (not-shadows-vconsts v₂)
  using (vwelldefined' v₁) (vwelldefined' v₂)
  unfolding ⟨v₁ = -⟩
  by auto

have closed-except (Sabs cs) (fmdom (fmupd (Name n) v₂ Γₐ))
  using (vclosed v₁) unfolding (v₁ = -)
  apply (auto simp: Stem.closed-except-simps list-all-iff)
  apply (auto simp: closed-except-def)
  done

have consts (Sabs cs) |⊆| fmdom (fmupd (Name n) v₂ Γₐ) |∪| C
  using (vwelldefined' v₁) unfolding (v₁ = -)
  unfolding sconsts-sabs
  by (auto simp: list-all-iff)

have ¬ shadows-consts (Sabs cs)
  using (not-shadows-vconsts v₁) unfolding (v₁ = -)
  by (auto simp: list-all-iff list-ex-iff)

have fdisjnt C (fmdom Γₐ)
  using (vwelldefined' v₁) unfolding (v₁ = -)
  by simp

have if-rval (λml-v. ∃ v. fmupd (Name n) v₂ Γₐ →₀ Sabs cs $ₕ Svar (Name n) ↓ v ∧ related-v v ml-v) bv
  proof (rule app1(2))
    show fnrel-on-fset (ids (Sabs cs $ₕ Svar (Name n))) related-v (fmupd (Name n) v₂ Γₐ) (fnmap-of-ns (sem-env.v env'))
    unfolding closure'
    unfolding fnrel-on-fset-updateI
    apply (simp del: frees-sterm.simps(3) constsimp.simps(3) name.sel)
    add: ids-def split!: sem-env.splits
    apply (rule fnrel-on-fset-updateI)
    apply (fold ids-def)
    using fnrel-on-fset (ids (Sabs cs)) related-v Γₐ →₀ apply simp
    done
  next
    show wellformed (Sabs cs $ₕ Svar (Name n))
    using vwellformed v₁ unfolding (v₁ = -)
by simp

next

show related-exp (Sabs cs $s$ Svar (Name n)) exp'
  unfolding ⟨exp' = ∅⟩
  using ⟨list-all2 (rel-prod related-pat related-exp) cs ml-cs⟩
  by (auto intro:related-exp.intros simp del: name.sel)

next

show closed-except (Sabs cs $s$ Svar (Name n)) (fmdom (fmupd (Name n) $v_2$ $\Gamma_\Lambda$))
  using ⟨closed-except (Sabs cs) (fmdom (fmupd (Name n) $v_2$ $\Gamma_\Lambda$))⟩
  by (simp add: closed-except-def)

next

show ¬ shadows-consts (Sabs cs $s$ Svar (Name n))
  using ⟨¬ shadows-consts (Sabs cs)⟩ ⟨Name n |\notin| all-consts⟩
  by simp

next

show consts (Sabs cs $s$ Svar (Name n)) |⊆| fmdom (fmupd (Name n) $v_2$ $\Gamma_\Lambda$) |∪| C
  using ⟨consts (Sabs cs) |⊆| fmdom (fmupd (Name n) $v_2$ $\Gamma_\Lambda$) |∪| C⟩
  by simp

next

show fdisjnt C (fmdom (fmupd (Name n) $v_2$ $\Gamma_\Lambda$))
  using ⟨Name n |\notin| all-consts⟩ ⟨fdisjnt C (fmdom $\Gamma_\Lambda$)⟩
  unfolding fdisjnt-alt-def all-consts-def by auto

qed fact +

then obtain $v$ where fmupd (Name n) $v_2$ $\Gamma_\Lambda$ |-v Sabs cs $s$ Svar (Name n) |-v related-v v ml-v
  unfolding ⟨bv = ∅⟩
  by auto

then obtain env pat rhs
  where vfind-match cs v2 = Some (env, pat, rhs)
  and fmupd (Name n) $v_2$ $\Gamma_\Lambda$ ++f env |-v rhs |-v
  by (auto elim: veval'-sabs-svarE)

hence (pat, rhs) ∈ set cs vmatch (mk-pat pat) v2 = Some env
  by (metis vfind-match-elem)+

hence linear pat wellformed rhs
  using ⟨wellformed v1⟩ unfolding ⟨v1 = ∅⟩
  by (auto simp: list-all-iff)

hence frees pat = patvars (mk-pat pat)
  by (simp add: mk-pat-frees)

hence fmdom env -= frees pat
  apply simp
  apply (rule vmatch-dom)
  apply (rule vmatch (mk-pat pat) v2 = Some env)
  done

obtain $v'$ where $\Gamma_\Lambda$ ++f env |-v rhs |-v $v'$ $v'$ ≈_e $v$
  proof (rule veval'-agree-eq)
    show fmupd (Name n) $v_2$ $\Gamma_\Lambda$ ++f env |-v, rhs |-v $v$ by fact

255
next
have $*$: Name $n |\notin| ids rhs if Name $n |\notin| fmdom env
proof
  assume Name $n |\in| ids rhs
  thus False
  unfolding ids-def
proof (cases rule: funion-strictE)
case $A$
moreover have Name $n |\notin| frees pat
  using that unfolding $fmdom env = frees pat$ .
ultimately have Name $n |\in| frees (Sabs cs)
  apply auto
  unfolding ffUnion-alt-def
  apply simp
  apply (rule fBexI[\text{where} x = (pat, rhs)])
  apply (auto simp: fset-of-list-elem)
  apply (rule \langle(pat, rhs) \in set cs\rangle)
  done
thus $?thesis$
  using \langleName $n |\notin| ids (Sabs cs)\rangle unfolding ids-def
  by blast
next
case $B$
hence Name $n |\in| consts (Sabs cs)
  apply auto
  unfolding ffUnion-alt-def
  apply simp
  apply (rule fBexI[\text{where} x = (pat, rhs)])
  apply (auto simp: fset-of-list-elem)
  apply (rule \langle(pat, rhs) \in set cs\rangle)
  done
thus $?thesis$
  using \langleName $n |\notin| ids (Sabs cs)\rangle unfolding ids-def
  by blast
qed
qed

show fnrel-on-fset (ids rhs) erelated ($\Gamma \Lambda ++f env$) (fmupd (Name $n$) $v_2 \Gamma \Lambda ++f env$)
apply rule
apply auto
  apply (rule option.rel-refl; rule erelated-refl)
using $*$ apply auto[]
  apply (rule option.rel-refl; rule erelated-refl)+
done
next
show closed-venv (fmupd (Name $n$) $v_2 \Gamma \Lambda ++f env$)
apply rule
apply fact
apply (rule vclosed.vmatch-env)
apply fact
apply fact
done
next
show wellformed-venv (fmupd (Name n) v_2 \Gamma_\Lambda ++f env)
apply rule
apply fact
apply (rule vwellformed.vmatch-env)
apply fact
apply fact
done
next
show closed-except rhs (fmupd (Name n) v_2 \Gamma_\Lambda ++f env)
using (fmupd env = frees pat) ((pat, rhs) \in set cs)
using :closed-except (Sabs cs) (fmupd (Name n) v_2 \Gamma_\Lambda))
by (auto simp: Sterm.closed-except-simps list-all-iff)
next
show wellformed rhs by fact
next
show consts rhs \subseteq fmdom (fmupd (Name n) v_2 \Gamma_\Lambda ++f env) \cup C
using :consts (Sabs cs) \subseteq fmdom (fmupd (Name n) v_2 \Gamma_\Lambda) \cup C
i((pat, rhs) \in set cs)
unfolding sconsts-sabs
by (auto simp: list-all-iff)
next
have fdisjnt C (fmdom env)
using ((pat, rhs) \in set cs) \n.shadowed-consts (Sabs cs):
unfolding (fmdom env = frees pat):
by (auto simp: list-ex-iff fdisjnt-alt-def all-consts-def)
thus fdisjnt C (fmdom (fmupd (Name n) v_2 \Gamma_\Lambda ++f env))
using :Name n \notin all-consts) fdisjnt C (fmdom \Gamma_\Lambda):
unfolding fdisjnt-alt-def
by (auto simp: all-consts-def)
next
show \n.shadowed-consts rhs
using ((pat, rhs) \in set cs) \n.shadowed-consts (Sabs cs):
by (auto simp: list-ex-iff)
next
have not-shadows-consts-env env
by (rule not-shadows-consts.vmatch-env) fact+
thus not-shadows-consts-env (fmupd (Name n) v_2 \Gamma_\Lambda ++f env)
using :not-shadows-consts-env (fmupd (Name n) v_2 \Gamma_\Lambda) by blast
next
have fmpred (\lambda-. vwelldefined') env
by (rule vmatch-welldefined') fact+
thus fmpred (\lambda-. vwelldefined') (fmupd (Name n) v_2 \Gamma_\Lambda ++f env)
using :fmpred (\lambda-. vwelldefined') (fmupd (Name n) v_2 \Gamma_\Lambda) by blast
qed blast
show thesis
apply (intro exI conjI)
unfolding (t = \_)
apply (rule veval'.comb)
using \( \Gamma \vdash t_1 \downarrow v_1 \) unfolding (\( v_1 = \_ \))
apply blast
apply fact
apply fact+
apply (rule related-v-ext)
apply fact+
done

next

case (reclosure \( \text{env} \Lambda \) \( \text{funs} \) \( \text{name} \) \( n \))
with reclosure have reclosure':
\( ml-v_1 = \text{Reclosure} \ \text{env} \Lambda \ \text{funs} \) \( \text{name} \)
\( \text{env}' = \text{update-v} \ (\lambda-. \ nsBind \ (\text{as-string} \ (\text{Name} \ n)) \ \text{ml-v}_2 \ (\text{build-rec-env} \ \text{env} \Lambda)) \) \( \text{env} \Lambda \)
unfolding (\( ml-vs = \_ \)) by auto
obtain \( \Gamma_A \ \text{css} \)
where \( v_1 = \text{Vrecabs} \ \text{css} \ (\text{Name} \ n) \) \( \Gamma_A \)
and \( \text{fmrel-on-fset} \ (\text{fbind} \ (\text{fmran} \ \text{css}) \ (\text{ids} \ o \ Sabs)) \ \text{related-v} \ \Gamma_A \)
(\( \text{fmap-of-ns} \ (\text{sem-env} \cdot \ \text{v} \ \text{env} \Lambda) \))
and \( \text{fmrel} \ (\lambda cs \ (n, e). \ \text{related-fun} \ cs \ n \ e) \ \text{css} \ (\text{fmap-of-list} \ (\text{map} \ (\text{map-prod} \ \text{Name} \ (\text{map-prod} \ \text{Name} \ \text{id}))) \ \text{funs})) \)
using (\( \text{related-v} \ v_1 \ ml-v_1 \) unfolding (\( ml-v_1 = \_ \))
by cases auto
from (\( \text{fmrel} - - \) ) have rel-option (\( \lambda cs \ (n, e). \ \text{related-fun} \ cs \ (\text{Name} \ n) \ e \))
(\( \text{fmlookup} \ \text{css} \ (\text{Name} \ n) \)) (\( \text{find-recfun} \ \text{name} \ \text{funs} \))
apply –
apply (\( \text{subst} \ \text{option}.\text{rel-set} \))
apply auto
apply (\( \text{drule} \ \text{fmrel-fmdom-eq} \))
apply (\( \text{drule} \ \text{fmdom-notI} \))
using (\( v_1 = \text{Vrecabs} \ \text{css} \ (\text{Name} \ n) \) \( \Gamma_A; \) \( \text{vwellformed} \ v_1 \)) apply auto[1]

using reclosure(\( \_ \)) apply auto[1]
apply (rule \( \text{fmrel-cases}[\text{where} \ x = \text{Name} \ n] \))
apply simp
apply (\( \text{subst} \ (\text{asm}) \ \text{fmlookup-of-list} \))
apply (\( \text{simp} \ \text{add: name.map-of-rekey'} \))
by blast

then obtain \( cs \) where \( \text{fmlookup} \ \text{css} \ (\text{Name} \ n) = \text{Some} \ cs \ \text{related-fun} \ cs \ (\text{Name} \ n) \) \( \exp' \)
using (\( \text{find-recfun} - - = \_ \))
by cases auto

then obtain \( ml-cs \)

258
where \( exp' = Mat (Var \ (Short \ (as-string \ (Name \ n)))) \) ml-cs Name n

\[
\begin{align*}
\text{|\$|} \ & \text{ids} \ (Sabs \ cs) \ \text{Name} \ n \ \text{|\$|} \ \text{all-consts} \\
\text{and} \ & \text{list-all2} \ (\text{rel-prod} \ \text{related-pat} \ \text{related-exp}) \ cs \ ml-cs \\
\text{by} \ & \text{(auto elim: related-funE)}
\end{align*}
\]

hence \( \text{cupcake-evaluate-single env'} \ (Mat \ (Var \ (Short \ n))) \) ml-cs

\[
\begin{align*}
\text{by} \ & \text{(auto elim: related-funE)} \\
\text{hence} \ & \text{cupcake-evaluate-single env'} \ (Mat \ (Var \ (Short \ n))) \ ml-cs \ \text{(Real ml-v)} \\
\text{using} \ & \langle \text{cupcake-evaluate-single env'} \ exp' \ bv \rangle \\
\text{unfolding} \ & \langle bv = \_ \rangle \\
\text{by} \ & \text{simp}
\end{align*}
\]

then obtain \( m\text{-env} \ v' \ ml\text{-rhs} \ ml\text{-pat} \)

\[
\begin{align*}
\text{where} \ & \text{cupcake-evaluate-single env'} \ (Var \ (Short \ n)) \ \text{(Real v')} \\
\text{and} \ & \text{cupcake-match-result} \ (\text{sem-env} \ . \ v \ (\text{env'}) \ ml-cs \ \text{Bind} v = \text{Real} \ (\text{ml-rhs} \ ml-pat \ ml-env)) \\
\text{by} \ & \text{cases auto}
\end{align*}
\]

have \( \text{closed-venv} \ (\text{fmupd} \ (\text{Name} \ n) \ v_2 \ (\Gamma_\Lambda \ ++_f \ \text{mk-rec-env} \ css \ \Gamma_\Lambda)) \)

\[
\begin{align*}
\text{using} \ & \langle \text{velosed} \ v_1 \rangle \ \langle \text{velosed} \ v_2 \rangle \\
\text{using} \ & \text{fmlookup css (Name name) = Some cs} \\
\text{unfolding} \ & \langle v_1 = \_ \rangle \ \text{mk-rec-env-def} \\
\text{apply} \ & \text{auto} \\
\text{apply} \ & \text{rule} \\
\text{apply} \ & \text{rule} \\
\text{apply} \ & \langle \text{auto intro: fmdomI} \rangle
\end{align*}
\]

done

have \( \text{wellformed-venv} \ (\text{fmupd} \ (\text{Name} \ n) \ v_2 \ (\Gamma_\Lambda \ ++_f \ \text{mk-rec-env} \ css \ \Gamma_\Lambda)) \)

\[
\begin{align*}
\text{using} \ & \langle \text{wellformed} \ v_1 \rangle \ \langle \text{wellformed} \ v_2 \rangle \\
\text{using} \ & \text{fmlookup css (Name name) = Some cs} \\
\text{unfolding} \ & \langle v_1 = \_ \rangle \ \text{mk-rec-env-def} \\
\text{apply} \ & \text{auto} \\
\text{apply} \ & \text{rule} \\
\text{apply} \ & \text{rule} \\
\text{apply} \ & \langle \text{auto intro: fmdomI} \rangle
\end{align*}
\]

done

have \( \text{not-shadows-vconsts-env} \ (\text{fmupd} \ (\text{Name} \ n) \ v_2 \ (\Gamma_\Lambda \ ++_f \ \text{mk-rec-env} \ css \ \Gamma_\Lambda)) \)

\[
\begin{align*}
\text{using} \ & \langle \text{not-shadows-vconsts} \ v_1 \rangle \ \langle \text{not-shadows-vconsts} \ v_2 \rangle \\
\text{using} \ & \text{fmlookup css (Name name) = Some cs} \\
\text{unfolding} \ & \langle v_1 = \_ \rangle \ \text{mk-rec-env-def} \\
\text{apply} \ & \text{auto} \\
\text{apply} \ & \text{rule} \\
\text{apply} \ & \text{rule} \\
\text{apply} \ & \langle \text{auto intro: fmdomI} \rangle
\end{align*}
\]

done

have \( \text{fmpred} \ (\lambda-. \ \text{welldefined'}) \ (\text{fmupd} \ (\text{Name} \ n) \ v_2 \ (\Gamma_\Lambda \ ++_f \ \text{mk-rec-env} \ css \ \Gamma_\Lambda)) \)

259
using (\texttt{vwelldefined\,' }v_1) (\texttt{vwelldefined\,' }v_2)
using \texttt{fmlookup css (Name name) = Some cs}
unfolding (v_1 = \triangleright mk-rec-env-def)
apply auto
apply rule
apply rule
apply (auto intro: fmdomI)
done

have closed-except (Sabs cs) (fmdom (fmupd (Name n) v_2 \Gamma_\Lambda))
using \texttt{veclosed v_1\ unfolding} (v_1 = \triangleright)
apply (auto simp: Stem\_closed-except-simps list-all-iff)
using \texttt{fmlookup css (Name name) = Some cs}
apply (auto simp: closed-except-def dest: \texttt{fmpredD[where m = css]})
done

have constrs (Sabs cs) \subseteq fmdom (fmupd (Name n) v_2 \Gamma_\Lambda) \cup (C \cup fmdom css)
using \texttt{vwelldefined\,' v_1} unfolding (v_1 = \triangleright)
unfolding sconssts-sabs
using \texttt{fmlookup css (Name name) = Some cs}
by (auto simp: list-all-iff list-ex-iff)

have \neg shadows-constrs (Sabs cs)
using \texttt{not-shadows-vconsts v_1} unfolding (v_1 = \triangleright)
using \texttt{fmlookup css (Name name) = Some cs}
by (auto simp: list-all-iff list-ex-iff)

have fdisjnt C (fmdom \Gamma_\Lambda)
using \texttt{vwelldefined\,' v_1} unfolding (v_1 = \triangleright)
using \texttt{fmlookup css (Name name) = Some cs}
by auto

have if-rval \((\lambda ml-v. \exists v. fmupd (Name n) v_2 (\Gamma_\Lambda ++ f mk-rec-env css \Gamma_\Lambda))\)
\vdash Sabs cs \texttt{Svar (Name n) \downarrow v \wedge related-v v ml-v} bv
proof (rule app1(2))
have fmrel-on-fset (ids (Sabs cs)) related-v \Gamma_\Lambda (fmap-of-ns (sem-env.v env_\Lambda))
apply (rule fmrel-on-fsubset)
apply fact
apply (subst funion-image-bind-eq[symmetric])
apply (rule \texttt{ffUnion-subset-elem})
apply (subst \texttt{fimage-iff})
apply (rule \texttt{fBexI})
apply simp
apply (rule \texttt{fmranI})
apply fact
done
have fmrel-on-fset (ids (Sabs cs)) related-v (mk-rec-env css Γ A)
cake-mk-rec-env funs envΛ
apply rule
apply (rule mk-rec-env-related[THEN fmrelD])
apply (rule fmrel - css -)
apply (rule fmrel-on-fset (fbind - -) related-v Γ A -)
done

show fmrel-on-fset (ids (Sabs cs $ Svar (Name n))) related-v (fmupd (Name n) v2 (Γ A ++f mk-rec-env css Γ A)) (fmap-of-ns (sem-env v env'))
unfolding reclosure'
apply (simp del: frees-sterm.simps(3) consts-sterm.simps(3) name.sel add: ids-def split!: sem-env.splits)
apply (rule fmrel-on-fset-updateI)
unfolding build-rec-env-fmap
apply (rule fmrel-on-fset-addI)
apply (fold ids-def)
subgoal
using (fmrel-on-fset (ids (Sabs cs)) related-v Γ A - by simp
subgoal
using (fmrel-on-fset (ids (Sabs cs)) related-v (mk-rec-env css Γ A) -
by simp
apply (rule (related-v v 2 ml-v2))
done
next
show wellformed (Sabs cs $ Svar (Name n))
using :wellformed v1 unfolding (v1 = -)
using (fmlookup css (Name name) = Some cs)
by auto
next
show related-exp (Sabs cs $ Svar (Name n)) exp'
unfolding (exp' = -)
apply (rule related-exp.intros)
apply fact
apply (rule related-exp.intros)
done
next
show closed-except (Sabs cs $ Svar (Name n)) (fdom (fmupd (Name n) v2 (Γ A ++f mk-rec-env css Γ A)))
using :closed-except (Sabs cs) (fdom (fmupd (Name n) v2 Γ A))
by (auto simp: list-all-iff closed-except-def)
next
show ¬ shadows-consts (Sabs cs $ Svar (Name n))
using ¬ shadows-consts (Sabs cs): (Name n | $| all-consts) by simp
next
show const (Sabs cs $ Svar (Name n)) |⊆| fdom (fmupd (Name n) v2 (Γ A ++f mk-rec-env css Γ A)) |∪| C
using :consts (Sabs cs) |⊆| - unfolding mk-rec-env-def
by auto
next

\[\text{show } \text{fdisjnt } C \left( (\text{fmdom } (\text{fmupd } (\text{Name } n) \ v_2 \ (\Gamma_\Lambda \ !+f \ \text{mk-rec-env } css \ \Gamma_\Lambda)) \right) \text{ using } \langle \text{Name } n \mid \notin \rangle \ \text{all-consts} \ (\text{fdisjnt } C \ (\text{fmdom } \Gamma_\Lambda)) \langle \text{vwelldefined} \rangle\]

\[\text{unfolding } \text{mk-rec-env-def } \langle v_1 = - \rangle \]

\[\text{by } (\text{auto simp: fdisjnt-alt-def all-consts-def})\]

\text{qed fact+}

then obtain \(v\)

where \(\text{fmupd } (\text{Name } n) \ v_2 \ (\Gamma_\Lambda \ !+f \ \text{mk-rec-env } css \ \Gamma_\Lambda) \vdash_v \text{Sabs cs } S\var\) \(\langle \text{vrelated-v } v \ \text{ml-v} \rangle \)

\[\text{unfolding } \langle \text{bv } = - \rangle\]

by \(\text{auto}\)

then obtain \(\text{env } \text{pat } \text{rhs}\)

where \(\text{vfind-match } cs \ v_2 = \text{Some } (\text{env, pat, rhs})\)

and \(\text{fmupd } (\text{Name } n) \ v_2 \ (\Gamma_\Lambda \ !+f \ \text{mk-rec-env } css \ \Gamma_\Lambda) \ !+f \ \text{env} \vdash_v \text{rhs}\)

\[\downarrow v\]

by \(\text{(auto elim: veval'-sabs-svarE)}\)

hence \((\text{pat, rhs}) \in \text{set cs vmatch } (\text{mk-pat pat}) \ v_2 = \text{Some env}\)

by \(\text{(metis vfind-match-elem)}\)

hence \(\text{linear pat wellformed rhs}\)

\[\text{using } \langle \text{vwellformed } v_1 \rangle \text{ unfolding } \langle v_1 = - \rangle\]

\[\text{by } (\text{auto simp: list-all-iff})\]

hence \(\text{frees pat } = \text{patvars } (\text{mk-pat pat})\)

by \(\text{(simp add: mk-pat-frees)}\)

hence \(\text{fmdom env } = \text{frees pat}\)

apply \text{simp}

apply \(\text{(rule vmatch-dom)}\)

apply \(\text{(rule vmatch } (\text{mk-pat pat}) \ v_2 = \text{Some env})\)

done

obtain \(v'\) where \(\Gamma_\Lambda \ !+f \ \text{mk-rec-env } css \ \Gamma_\Lambda \ !+f \ \text{env} \vdash_v \text{rhs } \downarrow v' \ v' \approx_e v\)

proof \(\text{(rule veval'-agree-eq)}\)

show \(\text{fmupd } (\text{Name } n) \ v_2 \ (\Gamma_\Lambda \ !+f \ \text{mk-rec-env } css \ \Gamma_\Lambda) \ !+f \ \text{env} \vdash_v \text{rhs}\)

\(\downarrow v\) by \text{fact}

next

have \(\ast\): \(\text{Name } n \mid \notin \mid \text{ids rhs if Name } n \mid \notin \mid \text{fdom env}\)

proof

assume \(\text{Name } n \mid \in \mid \text{ids rhs}\)

thus \(\text{False}\)

unfolding \text{ids-def}

proof \(\text{(cases rule: funion-strictE)}\)

case \(A\)

moreover have \(\text{Name } n \mid \notin \mid \text{frees pat}\)

using that \text{unfolding } \langle \text{fmdom env } = \text{frees pat} \rangle .\)

ultimately have \(\text{Name } n \mid \in \mid \text{frees } (\text{Sabs cs})\)

262
apply auto
unfolding ffUnion-alt-def
apply simp
apply (rule fBexI[where \( x = (\text{pat, rhs}) \)])
apply (auto simp: fset-of-list-elem)
apply (rule (\( \text{pat, rhs} \) \in \text{set cs} ) )
done
thus \(?thesis\)
using (\( \text{Name n} \mid \text{\notin} \) \text{ids} (\text{Sabs cs})\)
by blast

next
case B
hence \( \text{Name n} \mid \text{\in} \) \text{consts} (\text{Sabs cs})
apply auto
unfolding ffUnion-alt-def
apply simp
apply (rule fBexI[where \( x = (\text{pat, rhs}) \)])
apply (auto simp: fset-of-list-elem)
apply (rule (\( \text{pat, rhs} \) \in \text{set cs} ) )
done
thus \(?thesis\)
using (\( \text{Name n} \mid \text{\notin} \) \text{ids} (\text{Sabs cs})\)
by blast

qed

show \( \text{fmrel-on-fset (ids rhs) erelated (} \Sigma \text{mk-rec-env css} \Sigma \text{)} \)
\( ++f \text{ env} )\)
(\( \text{fmupd (Name n) v2 (} \Sigma \text{mk-rec-env css} \Sigma \text{)} ++f \text{ env} )\)
apply rule
apply auto
apply (rule option.rel-refl; rule erelated-refl)
using * apply auto]
apply (rule option.rel-refl; rule erelated-refl) +
using * apply auto]
apply (rule option.rel-refl; rule erelated-refl) +
done
next
show \( \text{closed-venv (fmupd (Name n) v2 (} \Sigma \text{mk-rec-env css} \Sigma \text{)} \)
\( ++f \text{ env} )\)
apply rule
apply fact
apply (rule vclosed.vmatch-env)
apply fact
apply fact
done
next
show \( \text{wellformed-venv (fmupd (Name n) v2 (} \Sigma \text{mk-rec-env css} \Sigma \text{)} \)
\( ++f \text{ env} )\)
apply rule

...
apply fact
apply (rule vwellformed.ematch-env)
apply fact
apply fact
done
next
  show closed-except rhs (fmdom (fmupd (Name n) v2 (ΓΛ ++f mk-rec-env css ΓΛ)) ++f env)
  using (fmdom env = \text{frees pat}) (pat, rhs) ∈ set cs
  using (closed-except (Sabs cs) (fmupd (Name n) v2 ΓΛ))
  apply (auto simp: Stem.closed-except-simps list-all-iff)
  apply (erule ballE [where x = (pat, rhs)])
  apply (auto simp: closed-except-def)
done
next
  show wellformed rhs by fact
next
  show consts rhs |⊆| fmdom (fmupd (Name n) v2 (ΓΛ ++f mk-rec-env css ΓΛ)) ++f env |∪| C
  using :consts (Sabs cs) |⊆| (pat, rhs) ∈ set cs
  unfolding sconsts-sabs mk-rec-env-def
  by (auto simp: list-all-iff)
next
  have fdisjnt C (fmdom env)
  using (\text{pat}, \text{rhs}) ∈ set cs (¬ \text{shadows-consts} (Sabs cs))
  unfolding \{\text{fmdom env = \text{frees pat}}\}
  by (auto simp: list-ex-iff all-consts-def fdisjnt-alt-def)
moreover have fdisjnt C (fmdom css)
  using :\text{vwelldefined}′ v1 unfolding (v1 = -)
  by simp
ultimately show fdisjnt C (fmupd (Name n) v2 (ΓΛ ++f mk-rec-env css ΓΛ) ++f env)
  using :\text{Name n} \notin all-consts (fdisjnt C (fmdom ΓΛ))
  unfolding fdisjnt-alt-def mk-rec-env-def
  by (auto simp: all-consts-def)
next
  show ¬ \text{shadows-consts} rhs
  using (\text{pat}, \text{rhs}) ∈ set cs (¬ \text{shadows-consts} (Sabs cs))
  by (auto simp: list-ex-iff)
next
  have not-shadows-vconsts-env env
  by (rule not-shadows-vconsts.ematch-env) fact+
thus not-shadows-vconsts-env (fmupd (Name n) v2 (ΓΛ ++f mk-rec-env css ΓΛ) ++f env)
  using (not-shadows-vconsts-env (fmupd (Name n) v2 (ΓΛ ++f mk-rec-env css ΓΛ))) by blast
next
  have fmpred (\lambda-. \text{vwelldefined}′) env
  by (rule ematch-welldefined′) fact+
thus \( \text{fmpred} (\lambda- \text{vwelldefined'}) (\text{fmupd} (\text{Name} n) v_2 (\Gamma_A + +f \text{mk-rec-env css} \Gamma_A) + +f \text{env}) \)

using \( \text{fmpred} (\lambda- \text{vwelldefined'}) (\text{fmupd} (\text{Name} n) v_2 (\Gamma_A + +f \text{mk-rec-env css} \Gamma_A)) \); by blast

qed simp

show \( \theta \text{thesis} \)
apply (intro exI conjI)
unfolding \( t = - \)
apply (rule \text{veval',rec-comb})
using \( \Gamma \vdash_v t \downarrow v_1 \)
unfolding \( v_1 = - \)
apply blast
apply fact+
apply (rule \text{related-v-ext})
apply fact+
done

qed

next

\( \text{case} \ (\text{mat1 env ml-scr ml-scr'} ml-cs ml-rhs ml-pat env' ml-res) \)

obtain \( \text{scr cs} \)
where \( t = \text{Subs cs \$}_t \) \( \text{scr related-exp scr ml-scr} \)
and \( \text{list-all2 (rel-prod related-pat related-exp) cs ml-cs} \)
using \( \text{(related-exp t (Mat ml-scr ml-cs))} \)
by cases

have \( \text{sem-env.c env = as-static-env} \)
using \( \text{(is-cupcake-all-env env)} \)
by (auto elim: is-cupcake-all-envE)

obtain \( \text{scr' where} \ \Gamma \vdash_v \text{scr} \downarrow \text{scr'} \text{ related-v scr' ml-scr'} \)
using \( \text{mat1(4) unfolding if-real.simps} \)
proof

show \( \text{is-cupcake-all-env env related-exp scr ml-scr wellformed-venv} \Gamma \)
closed-venv \( \Gamma \) \( \text{fmpred} (\lambda- \text{vwelldefined'}) \Gamma \) \( \text{fdisjnt C (fmdom } \Gamma) \)
not-shadows-consts-env \( \Gamma \)
by fact+

next

show \( \text{fmrel-on-fset (ids scr) related-v} \Gamma \) \( \text{(fmap-of-ns (sem-env.v env))} \)
apply (rule \text{fmrel-on-fsubset})
apply fact
unfolding \( \langle t = - \rangle \) ids-def
apply auto
done

next

show \( \text{wellformed scr consts scr} \subseteq \text{fmdom } \Gamma \cup C \)
closed-except scr \( \text{(fmdom } \Gamma) \) \( \neg \text{shadows-consts scr} \)
using mat1 unfolding \((t = \cdot)\) by (auto simp: closed-except-def)

qed blast

have list-all \((\lambda(pat, \cdot). \text{linear}\ pat)\) cs
  using mat1 unfolding \((t = \cdot)\) by (auto simp: list-all-iff)

obtain rhs pat \(\Gamma'\)
  where ml-pat = mk-ml-pat (mk-pat pat) related-exp rhs ml-rhs
    and vfind-match cs scr' = Some \((\Gamma',\ pat, rhs)\)
  and var-env \(\Gamma'\) env'
  using (list-all2 - cs ml/cs) (list-all - cs) \(\text{related}-v\ scr'\) ml-ser' mat1(2)
unfolding \(\text{sem-env}.c\ env = \text{as-static-cenv}\)
by (auto elim: match-all-related)

due vmatch (mk-pat pat) scr' = Some \(\Gamma'\)
by (auto dest: vfind-match-elem)

hence patvars (mk-pat pat) = fmdom \(\Gamma'\)
by (auto simp: vmatch-dom)

have \((pat, rhs)\) \(\in\) set cs
  by (rule vfind-match-elem) fact

have linear pat
  using \((pat, rhs)\) \(\in\) set cs \(\text{wellformed}\) unfolding \((t = \cdot)\)
  by (auto simp: list-all)

hence fmdom \(\Gamma'\) = frees pat
  using \(\text{patvars}\) (mk-pat pat) = fmdom \(\Gamma'\)
  by (simp add: mk-pat-frees)

show ?case
proof (rule if-rvalI)
  fix ml-rhs'
  assume ml-res = Rval ml-rhs'

obtain rhs' where \(\Gamma \vdash_{f} \Gamma' \vdash_{v} rhs' \vdash_{\text{related-v}} rhs'\) ml-rhs'
  using mat1(5) unfolding \(\langle ml-res = \cdot \rangle\) if-rval.simps
proof
  show is-cupcake-all-env \((env \| \text{sem-env}.v := \text{nsAppend} (\text{alist-to-ns} env')\)
  \((\text{sem-env}.v env) \|)\)
    proof (rule cupcake-v-update-preserve)
    have list-all (is-cupcake-value \circ snd) env'
      proof (rule match-all-preserve)
        show cupcake-c-ns (sem-env.c env)
        unfolding \(\text{sem-env}.c\ env = \cdot\) by (rule static-cenv)
      next
      have is-cupcake-exp (Mat ml-scr ml-cs)
        apply (rule related-exp-is-cupcake)
        using mat1 by auto
      thus cupcake-clauses ml-cs

266
by simp

show is-cupcake-value ml-scr'
  apply (rule cupcake-single-preserve)
  apply (rule mat1)
  apply (rule mat1)
  using (is-cupcake-exp (Mat ml-scr ml-cs)) by simp
qed fact+

hence is-cupcake-ns (alist-to-ns env')
  by (rule cupcake-alist-to-ns-preserve)
moreover have is-cupcake-ns (sem-env.v env)
  by (rule is-cupcake-all-envD) fact
ultimately show is-cupcake-ns (nsAppend (alist-to-ns env') (sem-env.v env))
  by (rule cupcake-nsAppend-preserve)
qed fact

next

show related-exp rhs ml-rhs
  by fact
next

have *: fmdom (fmap-of-list (map (map-prod Name id) env')) = fmdom Γ'
  using (var-env Γ' env')
  by (metis fnrel-fmdom-eq)

have **: id |∈| ids t if id |∈| ids rhs id |∉| fmdom Γ' for id
  using (id |∈| ids rhs) unfolding ids-def
proof (cases rule: funion-strictE)
case A
  from that have id |∉| frees pat
    unfolding (fmdom Γ' = frees pat) by simp
  hence id |∈| frees t
    using (pat, rhs) ∈ set cs
    unfolding (t = -)
    apply auto
    apply (subst ffUnion-alt-def)
    apply simp
    apply (rule fBexI [where x = (pat, rhs)])
    using A apply (auto simp: fset-of-list-elem)
  done
thus id |∈| frees t |∪| consts t by simp
next
case B
  hence id |∈| consts t
    using (pat, rhs) ∈ set cs
    unfolding (t = -)
    apply auto
    apply (subst ffUnion-alt-def)
    apply simp

267
apply (rule fBext[where \( x = (\text{pat}, \text{rhs}) \)])
apply (auto simp: fset-of-list-elem)
done
thus \( \text{id} \in \text{frees t} \cup \text{consts t} \) by simp
qed

have \( \text{fmrel-on-fset} (\text{ids rhs}) \text{ related-\( v \)} (\Gamma \mapsto f \Gamma') (\text{fmap-of-ns (sem-env.v env)} \mapsto f \text{fmap-of-list} (\text{map (map-prod Name id) env'})) \)
apply rule
apply simp
apply safe
subgoal
apply (rule fmrelD)
apply (rule \( \langle \text{var-env \Gamma', env'} \rangle \))
done
subgoal using \( * \) by simp
subgoal using \( * \)
by (metis \( \text{no-types, hide-lams, comp-def fimageI fdom-fmap-of-list} \) fset-of-list-map fst-comp-map-prod)
subgoal using \( ** \)
by (metis \( \text{fmlookup-ns fmrel-on-fsetD mat1.prems(2)} \))
done

thus \( \text{fmrel-on-fset} (\text{ids rhs}) \text{ related-\( v \)} (\Gamma \mapsto f \Gamma') (\text{fmap-of-ns (sem-env.v env)} \mapsto f \text{fmap-of-list} (\text{map (map-prod Name id) env'})) \)
by (auto split: sem-env.splits)

next
show \( \text{wellformed-venv} (\Gamma \mapsto f \Gamma') \)
apply rule
apply fact
apply (rule \( \text{wellformed.\( vmatch-env \)} \))
apply fact
apply (rule \( \text{veval'\text{-wellformed}} \))
apply fact
using mat1 unfolding \( t = \_ \) by auto

next
show \( \text{closed-venv} (\Gamma \mapsto f \Gamma') \)
apply rule
apply fact
apply (rule \( \text{closed.\( vmatch-env \)} \))
apply fact
apply (rule \( \text{veval'\text{-closed}} \))
apply fact
using mat1 unfolding \( t = \_ \) by (auto simp: closed-except-def)

next
show \( \text{fmpred (\lambda . \text{vwelldefined'}) (\Gamma \mapsto f \Gamma') \)} \)
apply rule
apply fact
apply (rule \( \text{vmatch-welldefined'} \))
apply fact
apply (rule veval’-welldefined’)
apply fact
using mat1 unfolding \( t = \cdot \) by auto
next
show not-shadows-vconsts-env \( (\Gamma ++ f \Gamma’) \)
apply rule
apply fact
apply (rule not-shadows-vconsts.vmatch-env)
apply fact
apply (rule veval’-shadows)
apply fact
using mat1 unfolding \( t = \cdot \) by auto
next
show wellformed rhs
using \((\text{pat}, \text{rhs}) \in \text{set cs} \) (wellformed t) unfolding \( t = \cdot \)
by (auto simp: list-all-iff)
next
show closed-except rhs \((\text{fdom} (\Gamma ++ f \Gamma’))\)
apply simp
unfolding \( \text{fdom} \Gamma’ = \text{frees pat} \)
using \((\text{pat}, \text{rhs}) \in \text{set cs} \) (closed-except t \((\text{fdom} \Gamma)\)) unfolding \( t = \cdot \)
by (auto simp: Sterm.closed-except-simps list-all-iff)
next
have consts \((\text{Sabs cs} | \subseteq | \text{fdom} \Gamma \uplus C)\)
using mat1 unfolding \( t = \cdot \) by auto
show consts rhs \( | \subseteq | \text{fdom} (\Gamma ++ f \Gamma’) \uplus C \)
apply simp
unfolding \( \text{fdom} \Gamma’ = \text{frees pat} \)
using \((\text{pat}, \text{rhs}) \in \text{set cs} \) (consts \((\text{Sabs cs} | \subseteq) \)
unfolding sc consts-sabs
by (auto simp: list-all-iff)
next
have fdisjnt C \( (\text{fdom} \Gamma’)\)
unfolding \( \text{fdom} \Gamma’ = \text{frees pat} \)
using \( \neg \text{shadows-consts t} \) \((\text{pat}, \text{rhs}) \in \text{set cs} \)
unfolding \( t = \cdot \)
by (auto simp: list-ex-iff fdisjnt-alt-def all-consts-def)
thus fdisjnt C \( (\text{fdom} (\Gamma ++ f \Gamma’))\)
using fdisjnt C \( (\text{fdom} \Gamma)\)
unfolding fdisjnt-alt-def by auto
next
show \( \neg \text{shadows-consts rhs} \)
using \((\text{pat}, \text{rhs}) \in \text{set cs} \) (\( \neg \text{shadows-consts t} \)) unfolding \( t = \cdot \)
by (auto simp: list-ex-iff)
qed blast

show \( \exists t’. \Gamma \vdash _v t \downarrow t’ \wedge \text{related-v} t’ \text{ ml-rhs’} \)
unfolding (t = -)
apply {intro exf conjI seval.intros}
apply {rule seval'.intros}
apply {rule seval'.intros}
apply fact+
done
qed
qed auto

theorem semantic-correctness:
fixes exp
assumes cupcake-evaluate-single env exp (Rval ml-v) is-cupcake-all-env env
assumes fmrel-on-fset (ids t) related-v Γ (fmap-of-ns (sem-env.v env))
assumes related-exp t exp
assumes wellformed t wellformed-env Γ
assumes closed-env Γ closed-except t (fmdom Γ)
assumes fnpred (λ- vwelldefined) Γ consts t |⊆| fmdom Γ |∪| C
assumes fdisjnt C (fmdom Γ)
assumes ¬ shadows consts t not-shadows-vconsts-env Γ
obtains v where Γ ⊢ v t ↓ v related-v ml-v
using semantic-correctness0[OF assms]
by auto
end end

5.4 Converting bytes to integers

theory CakeML-Byte
imports
  CakeML.Evaluate-Single
  Show.Show-Instances
begin

definition pat :: Ast.pat where
pat = Ast.Pcon (Some (Short "String-char-Char")) (map (λn. Ast.Pvar ("b" @ show n)) [0..<8])

definition cake-plus :: exp ⇒ exp ⇒ exp where
cake-plus t u = Ast.App (Ast.Opn Ast.Plus) [t, u]

lemma cake-plus-correct:
  assumes evaluate env s u = (s', Rval (Litv (IntLit y)))
  assumes evaluate env s' t = (s'', Rval (Litv (IntLit x)))
  shows evaluate env s (cake-plus t u) = (s'', Rval (Litv (IntLit (x + y))))
unfolding cake-plus-def using assms by simp

definition cake-times :: exp ⇒ exp ⇒ exp where
cake-times \( t \ u = \text{Ast.App} (\text{Ast.Opn} \text{Ast.Times}) [t, u] \)

**Lemma cake-times-correct:**

**Assumes** evaluate env \( s \ u = (s', \text{Real} (\text{IntLit} \ y)) \)

**Assumes** evaluate env \( s' \ t = (s'', \text{Real} (\text{IntLit} \ x)) \)

**Shows** evaluate env \( s \) (cake-times \( t \ u) = (s'', \text{Real} (\text{IntLit} \ (x \ast y))) \)

**Unfolding cake-times-def using assms by simp**

**Definition cake-int-of-bool :: exp ⇒ exp where**

cake-int-of-bool \( e = \text{Ast.Mat} \ e \)

\[
\begin{align*}
&[(\text{Ast.Pcon} (\text{Some} (\text{Short} "HOL-False")) [], \text{Lit} (\text{IntLit} 0)), \\
&(\text{Ast.Pcon} (\text{Some} (\text{Short} "HOL-True")) [], \text{Lit} (\text{IntLit} 1))] \\
\end{align*}
\]

**Definition summands :: exp list where**

summands = map (λn. cake-times (Lit (IntLit (2 ^ n))) (cake-int-of-bool (Ast.Var (Short ("b" @ show n)))))) [0..<8]

**Definition cake-int-of-byte :: exp where**

cake-int-of-byte =

\[
\text{Ast.Fun} "x" (\text{Ast.Mat} (\text{Ast.Var} (\text{Short} "x")) [(\text{pat}, \text{foldl} \text{cake-plus} (\text{Lit} (\text{IntLit} 0)) \text{summands})])
\]

end
Chapter 6

Composition of phases and full compilation pipeline

theory Doc-Compiler
imports Main
begin
end

6.1 Composition of correctness results

theory Composition
imports
../Backend/CakeML-Correctness
CakeML.Semantics
begin

hide-const (open) sem-env.v

Term-Class.term → nterm → pterm → sterm

6.1.1 Reflexive-transitive closure of irules.compile-correct.

lemma (in prules) prewrite-closed:
  assumes rs ⊢ p t → t' closed t
  shows closed t'
using assms proof induction
  case (step name rhs)
  thus ?case
    using all-rules by force
next
  case (beta c)
  obtain pat rhs where c = (pat, rhs) by (cases c) auto
  with beta have closed-except rhs (frees pat)
by (auto simp: closed-except-simps)
show ?case
apply (rule rewrite-step-closed [OF - beta (2) [unfolded c : =]])
using (closed-except rhs (frees pat)) betas by (auto simp: closed-except-def)
qed (auto simp: closed-except-def)
corollary (in prules) prewrite-rt-closed:
assumes rs ⊢ p t →* t' closed t
shows closed t'
using assms
by induction (auto intro: prewrite-closed)
corollary (in irules) compile-correct-rt:
assumes Rewriting-Pterm.compile rs ⊢ p t →* t' finished rs
shows rs ⊢ i t →* t'
using assms proof (induction rule: rtranclp-induct)
case step
thus ?case by (meson compile-correct rtranclp simps)
qed auto

6.1.2 Reflexive-transitive closure of prules.compile-correct.

lemma (in prules) compile-correct-rt:
assumes Rewriting-Sterm.compile rs ⊢ s t →* t' wellformed t
shows rs ⊢ p stem-to-pterm u →* stem-to-pterm u'
using assms proof induction
  case step
  thus ?case
  by (meson compile-correct rtranclp.simps)
q ed auto

lemma srewrite-stepD:
assumes srewrite-step rs name t
shows (name, t) ∈ set rs
using assms by induct auto

lemma (in srules) srewrite-wellformed:
assumes rs ⊢ s t →* t' wellformed t
shows wellformed t'
using assms proof induction
  case (step name rhs)
  hence (name, rhs) ∈ set rs
  by (auto dest: srewrite-stepD)
  thus ?case using all-rules by (auto simp: list-all-iff)
next
case (beta cs t t')
then obtain pat rhs env where (pat, rhs) ∈ set cs match pat t = Some env t'

= subst rhs env
  by (elim rewrite-firstE)
show ?case
  unfolding (t' = ∅)
proof (rule subst-wellformed)
  show wellformed rhs
    using ⟨(pat, rhs) ∈ set cs; beta by (auto simp: list-all-iff)⟩
next
  show wellformed-env env
    using ⟨match pat t = Some env; beta⟩
    by (auto intro: wellformed.match)
qed
qed auto

lemma (in srules) srewrite-wellformed-rt:
  assumes rs ⊢ s t −→∗ t' wellformed t
  shows wellformed t'
  using assms
  by induction (auto intro: srewrite-wellformed)

lemma vno-abs-value-to-sterm: no-abs (value-to-sterm v) −→ vno-abs v for v
  by (induction v) (auto simp: no-abs.list-comb list-all-iff)

6.1.3 Reflexive-transitive closure of rules.compile-correct.

corollary (in rules) compile-correct-rt:
  assumes compile ⊢ n u −→∗ u' closed u
  shows rs ⊢ nterm-to-term' u −→∗ nterm-to-term' u'
  using assms
  proof induction (auto intro: srewrite-wellformed)
    case (step u' u'')
    hence rs ⊢ nterm-to-term' u −→∗ nterm-to-term' u'
      by auto
    also have rs ⊢ nterm-to-term' u' −→ nterm-to-term' u''
      using step by (auto dest: rewrite-rt-closed intro!: compile-correct simp: closed-except-def)
    finally show ?case .
  qed auto

6.1.4 Reflexive-transitive closure of irules.transform-correct.

corollary (in irules) transform-correct-rt:
  assumes transform-irule-set rs ⊢ i u −→* u'' t ≈ₚ u closed t
  obtains t'' where rs ⊢ i t −→* t'' t ≈ₚ u''
  using assms proof (induction arbitrary: thesis t)
    case (step u' u'')
    obtain t' where rs ⊢ i t −→* t' t' ≈ₚ u'
      using step by blast
    obtain t'' where rs ⊢ i t −→* t'' t'' ≈ₚ u''
apply (rule transform-correct)
  apply (rule (transform-irule-set rs ⊢ i t † u †+ u′))
  apply (rule (t′ ≈p u′))
  apply (rule irerule-rt-closed)
  apply (rule rs ⊢ i t →* t′)
  apply (rule (closed t))
  apply blast
  done

show ?case
  apply (rule step.prems)
  apply (rule rtranclp-trans)
  apply fact+
  done
qed blast

corollary (in irules) transform-correct-rt-no-abs:
  assumes transform-irule-set rs ⊢ i t →* u closed t no-abs u
  shows rs ⊢ i t →* u
proof -
  have t ≈p t by (rule prelated-refl)
  obtain t' where rs ⊢ i t →* t' t' ≈p u
    apply (rule transform-correct-rt)
    apply (rule assms)
    apply (rule t ≈p t')
    apply (rule assms)
    apply blast
    done
thus ?thesis
  using assms by (metis prelated-no-abs-right)
qed

corollary transform-correct-rt-n-no-abs0:
  assumes irules C rs (transform-irule-set ^ n) rs ⊢ i t →* u closed t no-abs u
  shows rs ⊢ i t →* u
using assms(1,2) proof (induction n arbitrary: rs)
  case (Suc n)
  interpret irules C rs by fact
  show ?case
    apply (rule transform-correct-rt-no-abs)
    apply (rule Suc.IH)
    apply (rule rules-transform)
    using Suc(3) apply (simp add: funpow-swap1)
    apply fact+
    done
qed auto

corollary (in irules) transform-correct-rt-n-no-abs:
  assumes (transform-irule-set ^ n) rs ⊢ i t →* u closed t no-abs u
shows \( rs \vdash t \rightarrow^* u \)
by (rule transform-correct-rt-n-no-abs0) (rule irules-axioms assms)+

hide-fact transform-correct-rt-n-no-abs0

6.1.5 Iterated application of \( \text{transform-irule-set} \).

**definition** max-arity :: \( \text{irule-set} \Rightarrow \text{nat} \)
where
max-arity \( rs \) = \( f\text{Max} \ ((\text{arity} \circ \text{snd}) |' |\ rs) \)

**lemma** rules-transform-iter0:
- assumes \( \text{irules C-info rs} \)
- shows \( \text{irules C-info ((transform-irule-set "\ n\) \ rs)} \)
- using \( \text{assms} \)
- by (induction \( n \)) (auto intro: \( \text{irules.rules-transform del: \ irulesI} \))

**lemma** (in \( \text{irules} \)) rules-transform-iter: \( \text{irules C-info ((transform-irule-set "\ n\) \ rs)} \)
by (rule rules-transform-iter0) (rule irules-axioms)

**lemma** transform-irule-set-n-heads: \( \text{fst} |' | (\text{(transform-irule-set "\ n\}) \ rs) = \text{fst} |' | \ rs \)
by (induction \( n \)) (auto simp: transform-irule-set-heads)

hide-fact rules-transform-iter0

**definition** transform-irule-set-iter :: \( \text{irule-set} \Rightarrow \text{irule-set} \)
where
transform-irule-set-iter \( rs \) = \( (\text{transform-irule-set "\ max-arity rs\} \ rs} \)

**lemma** transform-irule-set-iter-heads: \( \text{fst} |' | (\text{(transform-irule-set-iter rs)} = \text{fst} |' | \ rs \)
unfolding transform-irule-set-iter-def by (simp add: transform-irule-set-n-heads)

**lemma** (in \( \text{irules} \)) finished-alt-def: \( \text{finished rs} \leftrightarrow \text{max-arity rs} = 0 \)
proof
- assume \( \text{max-arity rs} = 0 \)
- hence \( \neg \text{fBex (arity o snd) |' | rs} ((\lambda x. 0 < x) \)
  using nonempty
  unfolding max-arity-def
  by (metis fBex-fempty fmax-ex-gr not-less0)
- thus \( \text{finished rs} \)
- unfolding finished-def
- by force
next
- assume \( \text{finished rs} \)
  have \( \text{fMax (arity o snd) |' | rs} \leq 0 \)
  proof (rule fMax-le)
    show \( \text{fBall (arity o snd) |' | rs} ((\lambda x. x \leq 0) \)
      using (finished rs) unfolding finished-def by force
  next

276
show (arity o snd) |\*| rs \neq \{||\}
using nonempty by force
qed
thus max-arity rs = 0
unfolding max-arity-def by simp
qed

lemma (in irules) transform-finished-id: finished rs \implies transform-irule-set rs = rs
unfolding transform-irule-set-def finished-def transform-irules-def map-prod-def id-apply
by (rule fset-map-snd-id) (auto simp: fmember.rep-eq elim: fBallE)

lemma (in irules) max-arity-decr: max-arity (transform-irule-set rs) = max-arity rs - 1
proof (cases finished rs)
  case True
  thus \(?thesis
  by (auto simp: transform-finished-id finished-alt-def)
next
  case False
  have (arity o snd) |\*| transform-irule-set rs = (\lambda x. x - 1) |\*| (arity o snd) |\*|
  rs
  unfolding transform-irule-set-def fset.map-comp
proof (rule fset.map-cong0, safe, unfold o-apply map-prod-simp id-apply snd-conv)
  fix name irs
  assume (name, irs) \in fset rs
  hence (name, irs) |\in| rs
    by (simp add: fmember.rep-eq)
  hence arity-compatibles irs irs \neq \{||\}
  using nonempty inner by (blast dest: fpairwiseD)+
  thus arity (transform-irules irs) = arity irs - 1
  by (simp add: arity-transform-irules)
qed
hence max-arity (transform-irule-set rs) = fMax ((\lambda x. x - 1) |\*| (arity o snd) |\*|
rs)
unfolding max-arity-def by simp
also have \ldots = fMax ((arity o snd) |\*| rs) - 1
proof (rule fmax-decr)
  show fBex (arity o snd) |\*| rs (\leq 1)
    using False unfolding finished-def by force
qed
finally show \(?thesis
  unfolding max-arity-def
  by simp
qed

lemma max-arity-decr':0:
  assumes irules C rs
  shows max-arity ((transform-irule-set ^' n) rs) = max-arity rs - n
proof (induction \( n \))

  case (Suc \( n \))

  show \(?case\)

  apply simp
  apply (subst irules.max-arity-decr)
  using Suc assms by (auto intro: irules.rules-transform-iter del: irulesI)

qed auto

lemma (in irules) max-arity-decr':

  max-arity ((transform-irule-set \^\ n) \( rs \)) =
  max-arity \( rs \) \( - \) \( n \)

  by (rule max-arity-decr'0) (rule irules-axioms)

hide-fact max-arity-decr'0

lemma (in irules) transform-finished:

  finished (transform-irule-set-iter \( rs \))

  unfolding transform-irule-set-iter-def

  by (auto simp: max-arity-decr' intro: rules-transform-iter del: Rewriting-Pterm-Elim.irulesI)

Trick as described in §7.1 in the locale manual.

locale irules' =

sublocale irules' \subseteq irules' -as-irules: irules C-info transform-irule-set-iter \( rs \)

unfolding transform-irule-set-iter-def by (rule rules-transform-iter)

sublocale crules \subseteq crules-as-irules': irules' C-info Rewriting-Pterm-Elim.compile \( rs \)

unfolding irules'-def by (fact compile-rules)

sublocale irules' \subseteq irules'-as-prules: prules C-info Rewriting-Pterm.compile (transform-irule-set-iter \( rs \))

by (rule irules'-as-irules.compile-rules) (rule transform-finished)

6.1.6 Big-step semantics

context srules begin

definition global-css :: 
  \( \langle \text{name}, \text{sclauses}\rangle \) fmap where

  global-css = fmap-of-list (map (map-prod id clauses) \( rs \))

lemma fmdom-global-css: fmdom global-css = \( \text{fst} |\ | fset-of-list \( rs \)\)

  unfolding global-css-def by simp

definition as-vrules :: vrule list where

  as-vrules = map (λ(name, -). (name, Vrecabs global-css name fnempty)) \( rs \)

lemma as-vrules-fst[simp]:

  \( \text{fst} |\ | fset-of-list \text{as-vrules} = fst |\ | fset-of-list \( rs \)\)

  unfolding as-vrules-def

  apply simp

278
apply (rule fset.map cong)
apply (rule refl)
by auto

lemma as-vrules-fst[simp]: map fst as-vrules = map fst rs
unfolding as-vrules-def
by auto

lemma list-all-as-vrulesI:
assumes list-all (λ(-, t). P fmempty (clauses t)) rs
assumes R (fst |- fset-of-list rs)
shows list-all (λ(-, t). value-pred.pred P Q R t) as-vrules
proof (rule list-allI, safe)
fix name rhs
assume (name, rhs) ∈ set as-vrules
hence rhs = Vrcabs global-css name fmempty
unfolding as-vrules-def by auto
moreover have fmpred (λ-. P fmempty) global-css
unfolding global-css-def list.pred-map
using assms by (auto simp: list-all-iff intro!: fmpred-of-list)
moreover have name |∈| fmdom global-css
unfolding global-css-def
apply auto
moreover have list-all (λ(-, t). vclosed t) as-vrules
unfolding vclosed-def
by (simp add: value-pred-alt-def)
ultimately show value-pred.pred P Q R rhs
by (simp add: value-pred-alt-def)
qed

lemma srules-as-vrules: vrules C-info as-vrules
proof (standard, unfold as-vrules-fst)
have list-all (λ(-, t). vwellformed t) as-vrules
unfolding vwellformed-def
apply (rule list-all-as-vrulesI)
apply (rule list.pred-mono-strong)
apply (rule all-rules)
apply (auto elim: clausesE)
done
moreover have list-all (λ(-, t). vclosed t) as-vrules
unfolding vclosed-def

279
apply (rule list-all-as-vrulesI)
apply auto
apply (rule list.pred-mono-strong)
apply (rule all-rules)
apply (auto elim: clausesE simp: Sterm.closed-except-simps)
done

moreover have list-all (\(\lambda (\cdot, t). \neg \text{is-Vconstr } t\)) as-vrules
unfolding as-vrules-def
by (auto simp: list-all-iff)

ultimately show list-all vrule as-vrules
unfolding list-all-iff by fastforce
next
show distinct (map fst as-vrules)
using distinct by auto
next
show fdisjnt (fst \(\mid\) fset-of-list rs) C
using disjnt by simp
next
show list-all (\(\lambda (\cdot, rhs). \text{not-shadows-vconsts } rhs\)) as-vrules
unfolding not-shadows-vconsts-def
apply (rule list-all-as-vrulesI)
apply auto
apply (rule list.pred-mono-strong)
apply (rule not-shadows)
by (auto simp: list-all-iff list-ex-iff all-consts-def elim!: clausesE)
next
show vconstructor-value-rs as-vrules
unfolding vconstructor-value-rs-def
apply (rule conjI)
unfolding vconstructor-value-def
apply (rule list-all-as-vrulesI)
apply (simp add: list-all-iff)
apply simp
apply simp
using disjnt by simp
next
show list-all (\(\lambda (\cdot, rhs). \text{vwelldefined } rhs\)) as-vrules
unfolding vwelldefined-def
apply (rule list-all-as-vrulesI)
apply auto
apply (rule list.pred-mono-strong)
apply (rule swelldefined-rs)
apply (rule not-shadows)
apply (erule clausesE)
apply hypsubst-thin
apply (subst (asm) welldefined-sabs)
by simp
next
  show distinct all-constructors
  by (fact distinct-ctr)
qed

sublocale srules-as-vrules: vrules C-info as-vrules
by (fact srules-as-vrules)

lemma rs'-rs-eq: srules-as-vrules.rs' = rs
unfolding srules-as-vrules.rs'-def
unfolding as-vrules-def
apply (subst map-prod-def)
apply simp
unfolding comp-def
apply (subst case-prod-twice)
apply (rule list-map-snd-id)
unfolding global-css-def
using all-rules map
apply (auto simp: list-all-iff map-of-is-map map-of-map map-prod-def fmap-of-list.rep-eq)
subgoal for a b
  by (erule ballE[where x = (a, b)], cases b, auto simp: is-abs-def term-cases-def)
done

lemma veval-correct:
  fixes v
  assumes as-vrules, fnempty ⊢ v t ↓ v wellformed t closed t
  shows rs, fnempty ⊢ s t ↓ value-to-sterm v
using assms
by (rule srules-as-vrules.veval-correct[unfolded rs'-rs-eq])
end

6.1.7 ML-style semantics

context srules begin

lemma as-vrules-mk-rec-env: fmap-of-list as-vrules = mk-rec-env global-css fnempty
apply (subst global-css-def)
apply (subst as-vrules-def)
apply (subst mk-rec-env-def)
apply (rule fmap-ext)
apply (subst fmlookup-fmmap-keys)
apply (subst fmap-of-list.rep-eq)
apply (subst fmap-of-list.rep-eq)
apply (subst map-of-map-keyed)
apply (subst (2) map-prod-def)
apply (subst id-apply)
apply (subst map-of-map)
apply simp
apply (subst option.map-comp)
apply (rule option.map-cong)
apply (rule refl)
apply simp
apply (subst global-css-def)
apply (rule refl)
done

abbreviation (input) vrelated ≡ srules-as-vrules.vrelated
notation srules-as-vrules.vrelated (⊢v / - [0, 50] 50)

lemma vrecabs-global-css-refl:
  assumes name |∈| fmdom global-css
  shows ⊢v Vrecabs global-css name fmempty ≈ Vrecabs global-css name fmempty
using assms
proof (coinduction arbitrary: name)
case vrelated
  have rel-option (λv1 v2. (∃ name. v1 = Vrecabs global-css name fmempty ∧ v2 = Vrecabs global-css name fmempty ∧ name |∈| fmdom global-css) ∨ ⊢v v1 ≈ v2)
  (fmlookup (fmap-of-list as-vrules) y) (fmlookup (mk-rec-env global-css fmempty) y)
  for y
  apply (subst as-vrules-mk-rec-env)
  apply (rule option.rel-refl-strong)
  apply (rule disjI1)
  apply (simp add: mk-rec-env-def)
  apply (elim conjE exE)
  apply (intro exI conjI)
  by (auto intro: fmdomI)
with vrelated show ?case
  by fastforce
qed

lemma as-vrules-refl-rs: fmrel-on-fset (fst |'| fset-of-list as-vrules) vrelated (fmap-of-list as-vrules) (fmap-of-list as-vrules)
apply rule
apply (subst (2) as-vrules-def)
apply (subst (2) as-vrules-def)
apply (simp add: fmap-of-list.rep-eq)
apply (rule rel-option-reflI)
apply simp
apply (erule map-of-SomeD)
apply auto
apply (rule vrecabs-global-css-refl)
unfolding global-css-def
by (auto simp: fset-of-list-elem intro: rev-fimage-eqI)

lemma as-vrules-refl-C: fmrel-on-fset C vrelated (fmap-of-list as-vrules) (fmap-of-list as-vrules)
proof
fix \( c \)
assume \( c \in C \)
hence \( c \notin \text{fset-of-list (map fst as-vrules)} \)
    using srules-as-vrules.vconstructor-value-rs
    unfolding vconstructor-value-rs-def fdisjnt-alt-def
    by auto
hence \( c \notin \text{fset-of-list (map fst as-vrules)} \)
    by simp
hence \( \text{fmlookup (fmap-of-list as-vrules)} = \text{None} \)
    by (metis fmdom-notD)
thus rel-option vrelated (fmlookup (fmap-of-list as-vrules)) \( c \)
    by simp
qed

lemma veval'\-correct'\:.
fixes \( t \) \( v \)
assumes \( \text{fmap-of-list as-vrules} \vdash v \downarrow t \)
assumes \( \text{wellformed} \ t \)
assumes \( \neg \text{shadows-consts} \ t \)
assumes \( \text{welldefined} \ t \)
assumes \( \text{closed} \ t \)
assumes \( \text{vno-abs} \ v \)
shows \( \text{as-vrules, fnempty} \vdash v \downarrow t \)
proof –
obtain \( v_1 \) where \( \text{as-vrules, fnempty} \vdash v \downarrow t \vdash v_1 \vdash v_1 \approx v \)
    using \( \text{fmap-of-list as-vrules} \vdash v \downarrow t \vdash v \)
    proof (rule srules-as-vrules.veval'\-correct', unfold as-vrules-fst)
    show \( \text{wellformed} \ t \vdash \text{shadows-consts} \ t \text{ closed} \ t \text{ consts} \ t \subseteq \text{all-consts} \)
        by fact+
next
    show \( \text{wellformed-venv} (\text{fmap-of-list as-vrules}) \)
        apply rule
        using srules-as-vrules.all-rules
        apply (auto simp: list-all-iff)
        done
next
    show \( \text{not-shadows-vconsts-env} (\text{fmap-of-list as-vrules}) \)
        apply rule
        using srules-as-vrules.not-shadows
        apply (auto simp: list-all-iff)
        done
next
    have \( \text{fmrel-on-fset (fst }|\vdash| \text{fset-of-list as-vrules }|\cup| C \text{ vrelated (fmap-of-list as-vrules) (fmap-of-list as-vrules)} \)
        apply (rule fmrel-on-fset-unionI)
        apply (rule as-vrules-refl-rs)
        apply (rule as-vrules-refl-C)
        done

283
show \( fmrel-on-fset \ (\text{consts } t) \ vrelated \ (fmap-of-list \ as-vrules) \ (fmap-of-list \ as-vrules) \)

apply (rule \( fmrel-on-fsubset \))

apply fact+

using assms by (auto simp: all-consts-def)

qed

thus \(?thesis\)

using assms by (metis srules-as-vrules.vrelated.eq-right)

qed

end

6.1.8 CakeML

context srules begin

definition as-sem-env :: \( v \ \text{sem-env} \Rightarrow v \ \text{sem-env} \) where

\[ as-sem-env \ \text{env} = (| \ \text{sem-env}.v = \text{build-rec-env} (\text{mk-letrec-body} \ (\text{all-consts} \ rs) \ \text{env} \ \text{nsEmpty}, \ \text{sem-env}.c = \text{nsEmpty} |) \]

lemma compile-sem-env:

evaluate-dec ck mn \text{env state} \ (\text{compile-group} \ \text{all-consts} \ rs) \ (\text{state}, \ Rval \ (\text{as-sem-env} \ \text{env} ))

unfolding compile-group-def as-sem-env-def

apply (rule evaluate-dec.dletrec1)

unfolding mk-letrec-body-def Let-def

apply (simp add; comp-def case-prod-twice)

using name-as-string.fst-distinct[OF distinct]

by auto

lemma compile-sem-env':

fun-evaluate-decs mn \text{state} \text{env} \ [(\text{compile-group} \ \text{all-consts} \ rs)]= \ (\text{state}, \ Rval \ (\text{as-sem-env} \ \text{env} ))

unfolding compile-group-def as-sem-env-def mk-letrec-body-def Let-def

apply (simp add; comp-def case-prod-twice)

using name-as-string.fst-distinct[OF distinct]

by auto

lemma compile-prog[unfolded combine-dec-result.simps, simplified]:

evaluate-prog ck env state \ (\text{compile} \ rs) \ (\text{state}, \ \text{combine-dec-result} \ (\text{as-sem-env} \ \text{env}) \ (\text{Rval} \ (| \ \text{sem-env}.v = \text{nsEmpty}, \ \text{sem-env}.c = \text{nsEmpty} |)))

unfolding compile-def

apply (rule evaluate-prog.cons1)

apply rule

apply (rule evaluate-top.tdec1)

apply (rule compile-sem-env)

apply (rule evaluate-prog.empty)
done

lemma compile-prog"[unfolded combine-dec-result.simps, simplified]:
  fun-evaluate-prog state env (compile rs) = (state, combine-dec-result (as-sem-env env) (Rval (| sem-env.v = nsEmpty, sem-env.c = nsEmpty |))
unfolding compile-def fun-evaluate-prog-def no-dup-mods-def no-dup-top-types-def
prog-to-mods-def prog-to-top-types-def decs-to-types-def
using compile-sem-env’ compile-group-def by simp

lemma semantics-prog: semantics-prog empty-state env (compile rs) (Terminate Success [])
unfolding semantics-prog-def evaluate-prog-with-clock-def
by (auto split: prod.split option.split simp: compile-prog’ empty-state-def empty-ffi-state-def
initial-ffi-state-def)
definition sem-env :: v sem-env where
sem-env ≡ extend-dec-env (as-sem-env empty-sem-env) empty-sem-env

lemma cupcake-sem-env: is-cupcake-all-env sem-env
unfolding as-sem-env-def sem-env-def
apply (rule is-cupcake-all-envI)
apply (simp add: extend-dec-env-def empty-sem-env-def nsEmpty-def)
apply (rule cupcake-nsAppend-preserve)
apply (simp add: empty-sem-env-def)
apply (simp add: nsEmpty-def)
apply (rule mk-letrec-cupcake)
apply simp
apply (simp add: empty-sem-env-def)
done

lemma sem-env-refl: fmrel related-v (fmap-of-list as-vrules) (fmap-of-ns (sem-env.v sem-env))
proof
  fix name
  show rel-option related-v (fmlookup (fmap-of-list as-vrules) name) (fmlookup (fmap-of-ns (sem-env.v sem-env)) name)
  apply (simp add: as-sem-env-def build-rec-env-fmap cake-mk-rec-env-def sem-env-def
    fmap-of-list.rvp-eq map-of-map-keyed option.rel-map
    as-vrules-def mk-letrec-body-def comp-def case-prod-twice)
  apply (rule option.rel-refl-strong)
  apply (rule related-v.rel-refl
    apply auto
    apply (simp add: fmapmap-of-list[ symmetric, unfolded apsnd-def map-prod-def id-def]
    fmap.rel-map

285
apply (thin-tac map-of rs name = -)
apply (rule fmap.rel-refl-strong)
apply simp
subgoal premises prems for rhs
proof –
obtain name where (name, rhs) ∈ set rs
using prems
including fmap.lifting
by transfer' (auto dest: map-of-SomeD)
hence is-abs rhs closed rhs welldefined rhs
using all-rules swelldefined-rs by (auto simp add: list-all-iff)
then obtain cs where clauses rhs = cs rhs = Sabs cs wellformed-clauses
using (⟨(name, rhs) ∈ set rs⟩ all-rules)
by (cases rhs) (auto simp: list-all-iff is-abs-def term-cases-def)
show ?thesis
unfolding related-fun-alt-def (clauses rhs = cs)
proof (intro conjI)
show list-all2 (rel-prod related-pat related-exp) cs (map (λ(pat, t). (mk-ml-pat (mk-pat pat), mk-con (frees pat |∪| all-consts) t)) cs)
unfolding list.rel-map
apply (rule list.rel-refl-strong)
apply (rename-tac z, case-tac z, hypsubst-thin)
apply simp
subgoal premises prems for pat t
proof (rule mk-exp-correctness)
have ¬ shadows-consts rhs
using (⟨(name, rhs) ∈ set rs⟩ not-shadows)
by (auto simp: list-all-iff all-consts-def)
thus ¬ shadows-consts t
unfolding :rhs = Sabs cs using prems
by (auto simp: list-all-iff list-ex-iff)
done
next
have frees t |⊆| frees pat
using (closed rhs) prems unfolding (rhs = -)
apply (auto simp: list-all-iff Sterm.closed-except-simps)
apply (erule ballE[where x = (pat, t)])
apply (auto simp: closed-except-def)
done
moreover have consts t |⊆| all-consts
using (welldefined rhs) prems unfolding (rhs = -) welldefined-sabs
by (auto simp: list-all-iff all-consts-def)
ultimately show ids t |⊆| frees pat |∪| all-consts
unfolding ids-def by auto
qed (auto simp: all-consts-def)
done
next
have 1: frees (Sabs cs) = {||}
using ⟨closed rhs⟩ unfolding ⟨rhs = Sabs cs⟩
by (auto simp: closed-except-def)

have 2: welldefined rhs
  using swelldefined-rs ⟨(name, rhs) ∈ set rs⟩
  by (auto simp: list-all-iff)

show fresh-fNext all-consts |∅| ids (Sabs cs)
  apply (rule fNext-not-member-subset)
  unfolding ids-def I
  using 2 ⟨rhs = -⟩ by (simp add: all-consts-def del:consts-sterm, simps)
next
show fresh-fNext all-consts |∅| all-consts
  by (rule fNext-not-member)
qed

qed

lemma semantic-correctness':
assumes cupcake-evaluate-single sem-env (mk-con all-consts t) (Rval ml-v)
assumes welldefined t closed t ∼ shadows-consts t wellformed t
obtains v where fmap-of-list as-vrules ⊢ v ⊥ v related-v v ml-v
using assms(1) proof (rule semantic-correctness)
show is-cupcake-all-env sem-env
  by (fact cupcake-sem-env)
next
show related-exp t (mk-con all-consts t)
  apply (rule mk-exp-correctness)
  using assms
  unfolding ids-def closed-except-def by (auto simp: all-consts-def)
next
show wellformed t ∼ shadows-consts t by fact+
next
show closed-except t (fmdom (fmap-of-list as-vrules))
  using ⟨closed t⟩ by (auto simp: closed-except-def)
next
show closed-venv (fmap-of-list as-vrules)
  apply (rule fmpred-of-list)
  using srules-as-vrules.all-rules
  by (auto simp: list-all-iff)

show wellformed-venv (fmap-of-list as-vrules)
  apply (rule fmpred-of-list)
  using srules-as-vrules.all-rules
  by (auto simp: list-all-iff)
next
have 1: fmpred (λ-. list-all (λ(pat, t). consts t |⊆| C |∪| fmdom global-css))
global-css

287
apply (subst (2) global-css-def)
apply (rule fnmpred-of-list)
apply (auto simp: map-prod-def)
subgoal premises prems for pat t
proof -
  from prems obtain cs where t = Sabs cs
  by (elim clausesE)
  have welldefined t
    using swelldefined-rs prems
    by (auto simp: list-all-iff fmdom-global-css)

show ?thesis
  using (welldefined t)
  unfolding (t = -) welldefined-sabs
  by (auto simp: all-consts-def list-all-iff fmdom-global-css)
qed
done

show fnmpred (λ-. welldefined′) (fmap-of-list as-vrules)
apply (rule fnmpred-of-list)
unfolding as-vrules-def
apply simp
apply (erule imageE)
apply (auto simp: prod.splits)
apply (subst fdisjnt-alt-def)
apply simp
apply (rule 1)
apply (subst global-css-def)
apply simp
subgoal for x1 x2
  apply (rule fimage-eqI[where x = (x1, x2)])
  by (auto simp: fset-of-list-elem)
subgoal
  using disjnt by (auto simp: fdisjnt-alt-def fmdom-global-css)
done

next
show not-shadows-vconsts-env (fmap-of-list as-vrules)
apply (rule fnmpred-of-list)
using srules-as-vrules.not-shadows
unfolding list-all-iff
by auto

next
show fdisjnt C (fmdom (fmap-of-list as-vrules))
using disjnt by (auto simp: fdisjnt-alt-def)

next
show fmrel-on-fset (ids t) related-v (fmap-of-list as-vrules) (fmap-of-ns (sem-env.v sem-env))
  unfolding fmrel-on-fset-fmrel-restrict
apply (rule fmrel-restrict-fset)

288
apply (rule sem-env-refl)
done
next
show consts t \subseteq\ fmdom (fmap-of-list as-erules) \cup C
  apply (subst fmdom-fmap-of-list)
  apply (subst as-erules-fst')
  apply simp
  using assms by (auto simp: all-consts-def)
qed blast
end

fun cake-to-value :: v ⇒ value where
cake-to-value (Conv (Some (name, -)) vs) = Vconstr (Name name) (map cake-to-value vs)

context cakeml' begin

lemma cake-to-value-abs-free:
  assumes is-cupcake-value v cake-no-abs v
  shows vno-abs (cake-to-value v)
  using assms by (induction v) (auto elim: is-cupcake-value.elims simp: list-all-iff)

lemma cake-to-value-related:
  assumes cake-no-abs v is-cupcake-value v
  shows related-v (cake-to-value v) v
  using assms proof (induction v)
    case (Conv c vs)
      then obtain name tid where c = Some ((as-string name), TypeId (Short tid))
      apply (elim is-cupcake-value.elims)
      subgoal
        by (metis name.sel v.simps(2))
      by auto
    show ?case
      unfolding (c = -)
      apply simp
      apply (rule related-v.conv)
      apply (simp add: list.rel-map)
      apply (rule list.rel-refl-strong)
      apply (rule Conv)
      using Conv unfolding (c = -)
      by (auto simp: list-all-iff)
  qed auto

lemma related-v-abs-free-uniq:
  assumes related-v v1 ml-v related-v v2 ml-v cake-no-abs ml-v
  shows v1 = v2
  using assms proof (induction arbitrary: v2)
    case (conv vs1 ml-vs name)
  qed

end
then obtain $v_2$ where $v_2 = Vconstr\ name\ vs_2\ list-all2\ related-v\ vs_2\ ml-vs$
    by (auto elim: related-v.cases simp: name.expand)
moreover have list-all cake-no-\abs\ ml-vs
    using conv by simp
have list-all2 ($=$) vs_1 vs_2
    using (list-all2 - vs_1 -$\vdash$) (list-all2 - vs_2 -$\vdash$) (list-all cake-no-\abs\ ml-vs)
    by (induction arbitrary: vs_2 rule: list.rel-induct) (auto simp: list-all2-Cons2)
thus ?case
    unfolding ($v_2 = -$)
    by (simp add: list.rel-eq)
qed auto

corollary related-v-abs-free-cake-to-value:
  assumes related-v v ml-v cake-no-\abs\ ml-v\ is-cupcake-value ml-v
  shows $v = cake-to-value ml-v$
  using assms by (metis cake-to-value-related related-v-abs-free-uniq)
end

context srules begin

lemma cupcake-sem-env-preserve:
  assumes cupcake-evaluate-single sem-env (mk-con S t) (Rval ml-v) wellformed t
  shows is-cupcake-value ml-v
apply (rule cupcake-single-preserve[OF assms(1)])
apply (rule cupcake-sem-env)
apply (rule mk-exp-cupcake)
apply fact
done

lemma semantic-correctness\':
  assumes cupcake-evaluate-single sem-env (mk-con all-consts t) (Rval ml-v)
  assumes welldefined t closed t $\neg$ shadows-consts t wellformed t
  assumes cake-no-\abs\ ml-v
  shows fmap-of-list as-\vrules $\vdash_v\ t \downarrow$ cake-to-value ml-v
using assms
by (metis cupcake-sem-env-preserve semantic-correctness\' related-v-abs-free-cake-to-value)
end

6.1.9 Composition

context rules begin

abbreviation term-to-nterm where
term-to-nterm t $\equiv$ fresh-frun (Term-to-Nterm.\term\to\nterm [] t) all-consts

abbreviation sterm-to-cake where
sterm-to-cake $\equiv$ rules-as-\irules.\crules-as-\irules'.\irules'-\as-\prules.\prules-as-\srules.mk-con
allconsts

abbreviation term-to-cake \( t \equiv\) stem-to-cake (pterm-to-sterm (nterm-to-pterm (term-to-nterm \( t \))))

abbreviation cake-to-term \( t \equiv\) (convert-term (value-to-sterm (cake-to-value \( t \))) :: term)

abbreviation cake-sem-env \( \equiv\) rules-as-nrules.crules-as-irules'.irules'-.as-prules.prules-as-srules.sem-env

definition compiled \( \equiv\) rules-as-nrules.crules-as-irules'.irules'-.as-prules.prules-as-srules.as-vrules

lemma fmdom-compiled: fmdom (fmap-of-list compiled) = heads-of \( rs \)

unfolding compiled-def

by (simp add:
   rules-as-nrules.crules-as-irules'.irules'-.as-prules.compile-heads
   Rewriting-Pterm.compile-heads transform-irule-set-iter-heads
   Rewriting-Pterm-Elim.compile-heads
   compile-heads consts-of-heads)

lemma cake-semantic-correctness:
  assumes cupcake-evaluate-single cake-sem-env (term-to-cake \( t \)) (Real ml-v)
  assumes welldefined \( t \) closed \( t \) ~ shadows-consts \( t \) wellformed \( t \)
  assumes cake-no-abs ml-v
  shows fmap-of-list compiled \( \vdash \) \( t \) ↓ cake-to-value ml-v

unfolding compiled-def

apply (rule rules-as-nrules.crules-as-irules'.irules'-.as-prules.prules-as-srules.semantic-correctness’’)

using assms

by (simp-all add:
   rules-as-nrules.crules-as-irules'.irules'-.as-prules.compile-heads
   Rewriting-Pterm.compile-heads transform-irule-set-iter-heads
   Rewriting-Pterm-Elim.compile-heads
   compile-heads consts-of-heads all-consts-def)

Lo and behold, this is the final correctness theorem!

theorem compiled-correct:
  — If CakeML evaluation of a term succeeds ...
    assumes \( \exists \) \( k \). Evaluate-Single.evaluate cake-sem-env \( s \{ \) clock := \( k \} \) (term-to-cake \( t \)) = \( (s', \) Real ml-v)
  — ... producing a constructor term without closures ...
    assumes cake-no-abs ml-v
  — ... and some syntactic properties of the involved terms hold ...
    assumes closed \( t \) ~ shadows-consts \( t \) welldefined \( t \) wellformed \( t \)
  — ... then this evaluation can be reproduced in the term-rewriting semantics
    shows \( rs \vdash t \) \( \rightarrow^* \) cake-to-term ml-v

proof
  let \( \textit{heads} = \textit{fst} \{\textit{fst-of-list rules-as-nrules.crules-as-irules'.irules'-.as-prules.prules-as-srules.as-vrules} \}
  have \( \textit{heads} = \textit{heads-of rs} \)
  using fmdom-compiled unfolding compiled-def by simp
have wellformed \( \text{nterm-to-pterms} \ (\text{term-to-nterm} \ t) \)
  by auto

hence wellformed \( \text{pterms-to-terms} \ (\text{nterm-to-pterms} \ (\text{term-to-nterm} \ t)) \)
  by (auto intro: pterm-to-sterm-wellformed)

have is-cupcake-all-env cake-sem-env
  by (rule rules-as-nrules.nrules-as-crules.crules-as-irules'.irules'-as-prules.prules-as-srules.cupcake-sem-env)

have is-cupcake-exp \( \text{term-to-cake} \ t \)

fact obtain \( k \) where Evaluate-Single.evaluate cake-sem-env \( (s \ | \ \text{clock} := k \ | \ t) = (s', \ \text{Rval} \ ml-v) \)
  using assms by blast

then have Big-Step-Unclocked-Single.evaluate cake-sem-env \( (s \ | \ \text{clock} := (\text{clock} \ s') \ | \ t) = (s', \ \text{Rval} \ ml-v) \)
  using unclocked-single-fun-eq by fastforce

have cupcake-evaluate-single cake-sem-env \( (\text{stern-to-cake} \ (\text{pterms-to-sterm} \ (\text{nterm-to-pterms} \ (\text{term-to-nterm} \ t)))) = (\text{Rval} \ ml-v) \)
  apply (rule cupcake-single-complete)
  apply fact+
  done

hence is-cupcake-value ml-v
  apply (rule rules-as-nrules.nrules-as-crules.crules-as-irules'.irules'-as-prules.prules-as-srules.cupcake-sem-env-preserve)
  by (auto intro: pterm-to-sterm-wellformed)

hence no-abs \( (\text{value-to-sterm} \ (\text{cake-to-value} \ ml-v)) \)
  by (metis no-abs-value-to-sterm)

hence no-abs \( (\text{stern-to-pterms} \ (\text{value-to-sterm} \ (\text{cake-to-value} \ ml-v))) \)
  by (metis stern-to-pterms convert-term-no-abs)

have welldefined \( \text{term-to-nterms} \ (\text{nterm-to-nterm} \ t) \)
  unfolding term-to-nterm'-def
  apply (subst fresh-frun-def)
  apply (subst pred-stateD[OF term-to-nterm-consts])
  apply (subst surjective-pairing)
  apply (rule refl)
  apply fact
  done

have welldefined \( \text{pterms-to-sterms} \ (\text{nterm-to-pterms} \ (\text{term-to-nterm} \ t)) \)
  apply (subst pterm-to-sterm-consts)
  apply fact
  apply (subst consts-nterm-to-pterms)
  apply fact+
  done


have \( \neg \text{shadows-consts} \ t \)
using assms unfolding shadows-consts-def fdisjnt-alt-def
by auto

hence \( \neg \text{shadows-consts} \ (\text{term-to-nterm} \ t) \)
unfolding shadows-consts-def shadows-consts-def
apply auto
using term-to-nterm-all-vars[folded wellformed-term-def]
by (metis assms(6) fdisjnt-swap sup-idem)

have \( \neg \text{shadows-consts} \ (\text{pterm-to-sterm} \ (\text{nterm-to-pterm} \ (\text{term-to-nterm} \ t))) \)
apply (subst pterm-to-sterm-shadows[ symmetric])
apply fact
apply (subst shadows-nterm-to-pterm)
unfolding shadows-consts-def
apply simp
apply (rule term-to-nterm-all-vars[ where \( T = \text{fempty} \), simplified, THEN fdisjnt-swap])
apply (fold wellformed-term-def)
apply fact
using \( \langle \text{closed} \ t \rangle \) unfolding closed-except-def
by (auto simp: fdisjnt-alt-def)

have \( \text{closed} \ (\text{term-to-nterm} \ t) \)
using assms unfolding closed-except-def
using term-to-nterm-vars unfolding wellformed-term-def by blast
hence \( \text{closed} \ (\text{nterm-to-pterm} \ (\text{term-to-nterm} \ t)) \)
using closed-nterm-to-pterm unfolding closed-except-def
by auto

have \( \text{closed} \ (\text{pterm-to-sterm} \ (\text{nterm-to-pterm} \ (\text{term-to-nterm} \ t))) \)
unfolding closed-except-def
apply (subst pterm-to-sterm-frees)
apply fact
using \( \langle \text{closed} \ (\text{nterm-to-pterm} \ (\text{term-to-nterm} \ t)) \rangle \) closed-nterm-to-pterm
unfolding closed-except-def
by auto

have fmap-of-list compiled \( \vdash _v \) pterm-to-sterm (nterm-to-pterm (term-to-nterm \( t \))) \| cake-to-value ml-v
by (rule cake-semantic-correctness) fact+
hence fmap-of-list rules-as-nrules.crules-as-irules'.irules'~as-prules.prules-as-srules.as-errules
\( \vdash _v \) pterm-to-sterm (nterm-to-pterm (term-to-nterm \( t \))) \| cake-to-value ml-v
using assms unfolding compiled-def by simp
hence rules-as-nrules.crules-as-irules'.irules'~as-prules.prules-as-srules.as-errules, fnempty \( \vdash _v \) pterm-to-sterm (nterm-to-pterm (term-to-nterm \( t \))) \| cake-to-value ml-v
proof (rule rules-as-nrules.crules-as-irules'.irules'~as-prules.prules-as-srules.veval'~correct'')
show \( \neg \) rules-as-nrules.crules-as-irules'.irules'~as-prules.prules-as-srules.shadows-consts
( pterm-to-sterm (nterm-to-pterm (term-to-nterm \( t \))))
using \( \langle \neg \text{shadows-consts} \ (:: \text{sterm}) \rangle \) (?heads = heads-of rs) by auto
next
show consts (pterm-to-sterm (nterm-to-pterm (term-to-nterm \( t \)))) \| \subseteq rules-as-nrules.crules-as-irules'.irules'~as-prules.prules-as-srules.as-errules

using (welldefined (pterm-to-stern -)) (?heads = -) by auto

qed fact+

hence Rewriting-Stern.compile (Rewriting-Pterm.compile (transform-irule-set-iter
(Rewriting-Pterm-Elim.compile (consts-of compile))))) \to_n stencil-to-stern
(stern-to-stern (term-to-stern t)) \to_n stencil-to-stern (value-to-stern (cake-to-value ml-v)

by (rule rules-as-nrules,crules-as-irules',irules'-as-prules,prules-as-srules,veval-correct)

fact+

hence Rewriting-Stern.compile (Rewriting-Pterm.compile (transform-irule-set-iter
(Rewriting-Pterm-Elim.compile (consts-of compile))))) \to_n stencil-to-stern (nterm-to-stern
term-to-nterm t) \to_n stencil-to-stern (value-to-stern (cake-to-value ml-v)

by (rule rules-as-nrules,crules-as-irules',irules'-as-prules,prules-as-srules,seval-correct)

fact

hence Rewriting-Pterm.compile (transform-irule-set-iter (Rewriting-Pterm-Elim.compile
(consts-of compile))))) \to_p stencil-to-stern (pterm-to-stern (term-to-stern
term-to-nterm t)) \to_p stencil-to-stern (value-to-stern (cake-to-value ml-v)

by (rule rules-as-nrules,crules-as-irules',irules'-as-prules,compile-correct-rt)

hence Rewriting-Pterm.compile (transform-irule-set-iter (Rewriting-Pterm-Elim.compile
(consts-of compile))))) \to_p nterm-to-stern (term-to-nterm t) \to_n stencil-to-stern

(value-to-stern (cake-to-value ml-v)

by (subst (asm) pterm-to-stern (stern-to-stern pterm-to-stern)

hence transform-irule-set-iter (Rewriting-Pterm-Elim.compile (consts-of compile))))) \to_i nterm-to-stern

(term-to-nterm t) \to_n stencil-to-stern (value-to-stern

(cake-to-value ml-v)

by (rule rules-as-nrules,crules-as-irules',irules'-as-prules,compile-correct-rt)

(rule rules-as-nrules,crules-as-irules.transform-finished)

have Rewriting-Pterm-Elim.compile (consts-of compile) \to_i nterm-to-stern (term-to-nterm
term-to-nterm t) \to_n stencil-to-stern (value-to-stern

(cake-to-value ml-v)

by (rule rules-as-nrules,crules-as-irules,transform-correct-rt-n-no-abs)

apply (rule rules-as-nrules,crules-as-irules,transform-correct-rt-def)

using (transform-irule-set-iter \to_i - \to_n unfolding transform-irule-set-iter-def

apply simp

apply fact+

done

then obtain t' where compile \to_n term-to-nterm t \to_n stencil-to-stern (term-to-nterm
term-to-nterm t) \to_n t' t' \to_i stencil-to-stern

(value-to-stern (cake-to-value ml-v)

using (closed (term-to-nterm t))

by (metis rules-as-nrules,compile-correct-rt)

hence no-abs t'

using (no-abs (stern-to-stern -))

by (metis irelated-no-abs)

have rs \to t nterm-to-nterm' (term-to-nterm t) \to_n nterm-to-nterm' t'

by (rule compile-correct-rt) fact+

hence rs \to t nterm-to-nterm' t'

apply (subst (asm) fresh-frun-def)

apply (subst (asm) nterm-to-nterm-nterm-to-term[where S = fempty and t =
t', simplified])

apply (fold wellformed-term-def)

apply fact
using `assms unfolding closed-except-def` by `auto`

have `nterm-to-pterm t' = stern-to-pterm (value-to-sterm (cake-to-value ml-v))`
using `t' ≈_t` by `auto`

hence `(convert-term t' :: pterm) = convert-term (value-to-sterm (cake-to-value ml-v))`

apply `(subst (asm) nterm-to-pterm)`
apply `fact`
apply `(subst (asm) stern-to-pterm)`
apply `fact`
apply `assumption`
done

hence `nterm-to-term' t' = convert-term (value-to-sterm (cake-to-value ml-v))`

apply `(subst nterm-to-term')`
apply `(rule no-abs t')`
apply `(rule convert-term-inj)`

subgoal premises
apply `(rule convert-term-no-abs)`
apply `fact`
done

subgoal premises
apply `(rule convert-term-no-abs)`
apply `fact`
done

apply `(subst convert-term-idem)`
apply `(rule no-abs t')`
apply `(rule convert-term-idem)`
apply `(rule no-abs (value-to-sterm (cake-to-value ml-v)))`
apply `assumption`
done

thus `?thesis`

using `rs ⊢ t →∗ nterm-to-term' t'` by `simp`

qed

end

end

6.2 Executable compilation chain

theory `Compiler`
imports `Composition`
begin

definition `term-to-exp :: C-info ⇒ rule fset ⇒ term ⇒ exp` where
`term-to-exp C-info rs t = cakeml.mk-con C-info (heads-of rs |\cup| constructors.C C-info)`
(pterm-to-sterm (nterm-to-pterms (fresh-frun (term-to-nterm [] t) (heads-of rs |∪| constructors.C C-info)))))

lemma (in rules) Compiler.term-to-exp C-info rs = term-to-cake
  unfolding term-to-exp-def by (simp add: allconsts-def)

primrec compress-pterms :: pterm ⇒ pterm where
  compress-pterms (Pabs cs) = Pabs (fcompress (map-prod id compress-pterms |'| cs))
  | compress-pterms (Pconst name) = Pconst name |
  compress-pterms (Pvar name) = Pvar name |
  compress-pterms (t $ p u) = compress-pterms t $ p compress-pterms u

lemma compress-pterms-eq[simp]: compress-pterms t = t
  by (induction t) (auto simp: subst-pabs-map_snd_id map-prod-id fnmem-ber.rep-eq)

definition compress-cruleset :: crule-set ⇒ crule-set where
  compress-cruleset = fcompress ◦ fimage (map-prod id fcompress)

definition compress-iruleset :: irule-set ⇒ irule-set where
  compress-iruleset = fcompress ◦ fimage (map-prod id (fcompress ◦ fimage (map-prod id compress-pterms)))

definition compress-pruleset :: prule fset ⇒ prule fset where
  compress-pruleset = fcompress ◦ fimage (map-prod id compress-pterms)

lemma compress-cruleset-eq[simp]: compress-cruleset rs = rs
  unfolding compress-cruleset-def by force

lemma compress-iruleset-eq[simp]: compress-iruleset rs = rs
  unfolding compress-iruleset-def map-prod-def by simp

lemma compress-pruleset-eq[simp]: compress-pruleset rs = rs
  unfolding compress-pruleset-def by force

definition transform-iruleset-iter :: irule-set ⇒ irule-set where
  transform-iruleset-iter rs = ((transform-iruleset ◦ compress-iruleset) ^^ max-arity rs) rs

definition as-sem-env :: C-info ⇒ srule list ⇒ v sem-env ⇒ v sem-env where
  as-sem-env C-info rs env =
    (| v = build-rec-env (cakeml.mk-letrec-body C-info (fcompress (map fst rs) |∪| constructors.C C-info) rs) env nsEmpty,
      nsEmpty = nsEmpty |
  )
definition empty-sem-env :: C-info ⇒ v sem-env where
empty-sem-env C-info = ([| sem-env.v = nsEmpty, sem-env.c = constructors, as-static-env C-info |])

definition sem-env :: C-info ⇒ srule list ⇒ v sem-env where
sem-env C-info rs = extend-dec-env (as-sem-env C-info rs (empty-sem-env C-info))

definition compile :: C-info ⇒ rule fset ⇒ Ast.prog where
compile C-info =
  CakeML-Backend.compile C-info ◦
  Rewriting-Sterm.compile ◦
  compress-prule-set ◦
  Rewriting-Pterm.compile ◦
  transform-irule-set-iter ◦
  compress-irule-set ◦
  Rewriting-Pterm-Elim.compile ◦
  compress-crule-set ◦
  Rewriting-Nterm.consts-of ◦
  fcompress ◦
  Rewriting-Nterm.compile′ C-info ◦
fcompress

definition compile-to-env :: C-info ⇒ rule fset ⇒ v sem-env where
compile-to-env C-info =
  sem-env C-info ◦
  Rewriting-Sterm.compile ◦
  compress-prule-set ◦
  Rewriting-Pterm.compile ◦
  transform-irule-set-iter ◦
  compress-irule-set ◦
  Rewriting-Pterm-Elim.compile ◦
  compress-crule-set ◦
  Rewriting-Nterm.consts-of ◦
  fcompress ◦
  Rewriting-Nterm.compile′ C-info ◦
fcompress

lemma (in rules) Compiler.compile-to-env C-info rs = rules.cake-sem-env C-info rs
unfolding Compiler.compile-to-env-def Compiler.sem-env-def Compiler.as-sem-env-def Compiler.empty-sem-env-def
unfolding rules-as-irules.crules-as-irules′.irules′-as-prules.prules-as-srules.sem-env-def
unfolding rules-as-irules.crules-as-irules′.irules′-as-prules.prules-as-srules.as-sem-env-def
unfolding empty-sem-env-def
by (auto simp:
  Compiler.compress-irule-set-eq[abs-def]
  Composition.transform-irule-set-iter-def[abs-def]
  Compiler.transform-irule-set-iter-def[abs-def] comp-def pre-constants.all-consts-def)
export-code

term-to-exp compile compile-to-env
checking Scala

end