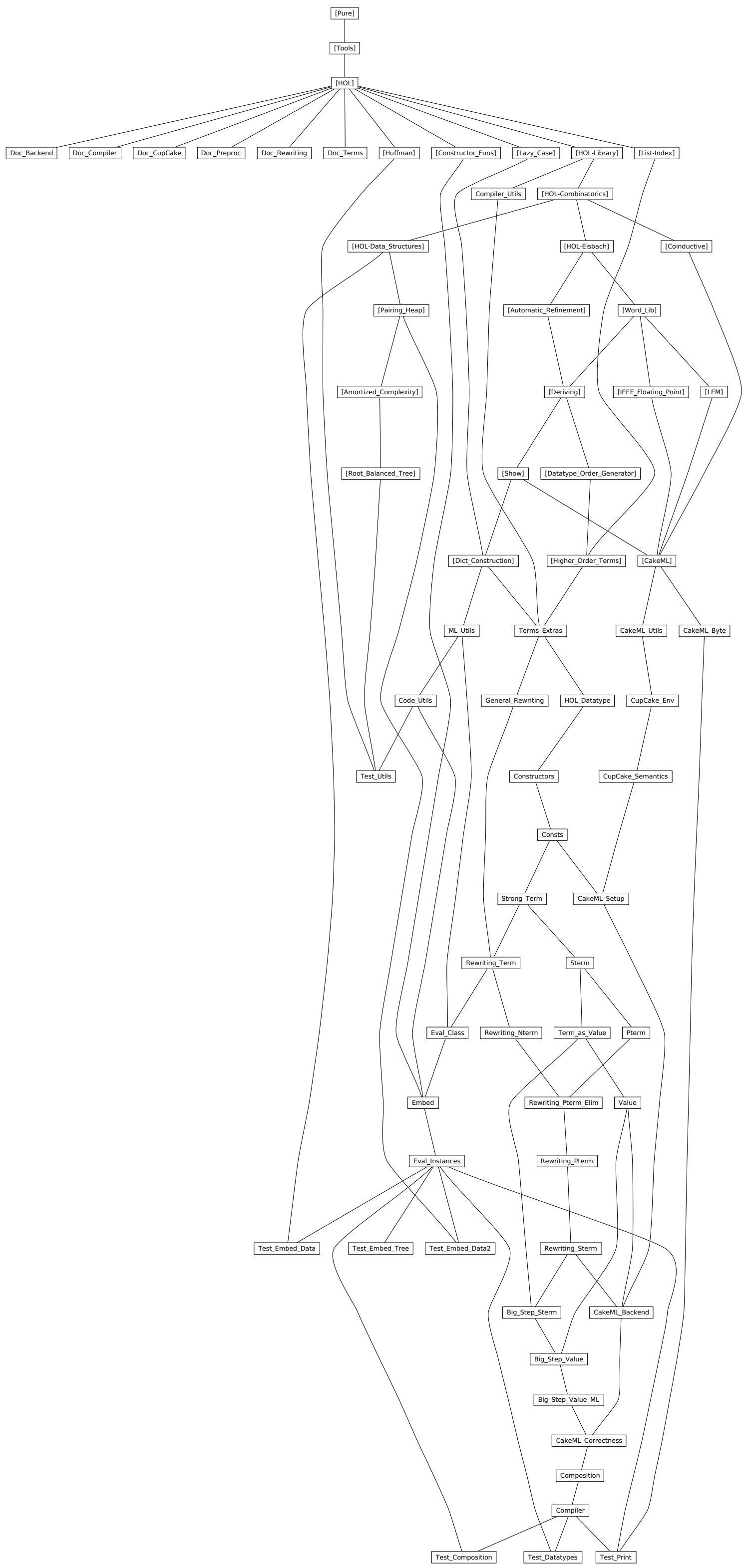
A Verified Code Generator from Isabelle/HOL to CakeML

Lars Hupel

September 1, 2025

Contents


[Pure]
[Tools]
[HOL]
Doc_Backend
Doc_Compiler
Doc_CupCake
Doc_Preproc
Doc_Rewriting
Doc_Terms
[Huffman]
[Constructor_Funs]
[Lazy_Case]
[HOL-Library]
[List-Index]
Compiler_Utils
[HOL-Combinatorics]
[HOL-Data_Structures]
[HOL-Eisbach]
[Coinductive]
[Pairing_Heap]
[Automatic_Refinement]
[Word_Lib]
[Amortized_Complexity]
[Deriving]
[IEEE_Floating_Point]
[LEM]
[Root_Balanced_Tree]
[Show]
[Datatype_Order_Generator]
[Dict_Construction]
[Higher_Order_Terms]
[CakeML]
ML_Utils
Terms_Extras
CakeML_Utils
CakeML_Byte
Code_Utils
General_Rewriting
HOL_Datatype
CupCake_Env
Test_Utils
Constructors
CupCake_Semantics
Consts
Strong_Term
CakeML_Setup
Rewriting_Term
Sterm
Eval_Class
Rewriting_Nterm
Term_as_Value
Pterm
Embed
Rewriting_Pterm_Elim
Value
Eval_Instances
Rewriting_Pterm
Test_Embed_Data
Test_Embed_Tree
Test_Embed_Data2
Rewriting_Sterm
Big_Step_Sterm
CakeML_Backend
Big_Step_Value
Big_Step_Value_ML
CakeML_Correctness
Composition
Compiler
Test_Composition
Test_Datatypes
Test_Print

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** ‹$$› *900*)

ML-file *hol-term.ML*

primrec *basic-rule* :: - ⇒ *bool* **where**
basic-rule (*lhs*, *rhs*) ⟷
  linear lhs ∧
  is-const (*fst* (*strip-comb lhs*)) ∧
  ¬ *is-const lhs* ∧
  frees rhs |⊆| *frees lhs*

lemma *basic-ruleI*[*intro*]:
  assumes *linear lhs*
  assumes *is-const* (*fst* (*strip-comb lhs*))
  assumes ¬ *is-const lhs*
  assumes *frees rhs* |⊆| *frees lhs*
  shows *basic-rule* (*lhs*, *rhs*)

using *assms* **by** *simp*

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

fun *unsplit-rule* :: $(name \times (term\ list \times 'a)) \Rightarrow (term \times 'a)$ **where**
unsplit-rule (*name*, (*args*, *rhs*)) = (*name* $$ *args*, *rhs*)

lemma *split-unsplit*: *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* (*split-rule r*) = *r*
using *assms*
by (*cases r*) (*simp add*: *split-beta*)

datatype *pat* = *Patvar name* | *Patconstr name pat list*

fun *mk-pat* :: *term* ⇒ *pat* **where**
mk-pat pat = (*case strip-comb pat of* (*Const s*, *args*) ⇒ *Patconstr s* (*map mk-pat args*) | (*Free s*, []) ⇒ *Patvar s*)

declare *mk-pat.simps*[*simp del*]

lemma *mk-pat-simps*[*simp*]:
  mk-pat (*name* $$ *args*) = *Patconstr name* (*map mk-pat args*)
  mk-pat (*Free name*) = *Patvar name*
apply (*auto simp*: *mk-pat.simps strip-list-comb-const*)
apply (*simp add*: *const-term-def*)
done

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

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

  have *map* (*patvars* ∘ *mk-pat*) *args* = *map frees args*
    using *comb* **by** *force*
  hence *fset-of-list* (*map* (*patvars* ∘ *mk-pat*) *args*) = *fset-of-list* (*map frees args*)
    by *metis*
  thus *?case*
    by (*simp add*: *freess-def*)

qed *simp*

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 $(t_1\ \$\ t_2)\ (u_1\ \$\ u_2) \longleftrightarrow$ *pattern-compatible* $t_1\ u_1 \wedge (t_1 = u_1 \longrightarrow$ *pattern-compatible* $t_2\ u_2)$ |
pattern-compatible $t\ u \longleftrightarrow t = u \vee$ *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-compatile-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* t_1 t_2 u_1 u_2)

  show *?case*
    proof (*cases* $t_1 = u_1 \wedge t_2 = u_2$)
      case *True*
      with *app-app* **show** *thesis*
        by *simp*
    next
      case *False*
      from *app-app* **have** *pattern-compatible* $t_1\ u_1\ t_1 = u_1 \Longrightarrow$ *pattern-compatible* $t_2\ u_2$
        by *auto*
      with *False* **have** *non-overlapping* $(t_1\ \$\ t_2)\ (u_1\ \$\ u_2)$
        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($pats_1$, -) ($pats_2$, -). *patterns-compatible* $pats_1$ $pats_2$)

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* (*list-comb f xs*) (*list-comb g ys*)
using *assms*
proof (*induction rule*: *rev-accum-rel.induct*)
case (*snoc xs ys x y*)

then have *pattern-compatible* (*list-comb f xs*) (*list-comb g ys*)
by *auto*

moreover have *pattern-compatible x y* **if** *list-comb f xs* = *list-comb g ys*
proof (*rule snoc*, *rule list-comb-semi-inj*)
show *length xs* = *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* ⟷ *t* = *u* ∨ *non-overlapping t u*
apply *rule*
apply (*erule pattern-compatible-cases*; *simp*)
apply (*erule disjE*)
apply (*metis pattern-compatible-refl*)
oops

— The counterexample:

definition *pats1* = [*Free* (*Name* ''*f*''), *Const* (*Name* ''*nil*'')]
definition *pats2* = [*Free* (*Name* ''*g*''), *Const* (*Name* ''*cons*'') $ *Free* (*Name* ''*x*'') $ *Free* (*Name* ''*xs*'')]

proposition *non-overlapping* (*list-comb c pats1*) (*list-comb c pats2*)
unfolding *pats1-def pats2-def*
apply (*simp add*: *app-term-def*)
apply (*rule non-overlapping-appI2*)
apply (*rule non-overlapping-const-appI*)
done

proposition ¬ *patterns-compatible pats1 pats2*
unfolding *pats1-def 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(*lhs*$_1$, -) (*lhs*$_2$, -). *pattern-compatible lhs*$_1$ *lhs*$_2$)

corollary *match-compatible-pat-eq*:
assumes *pattern-compatible* t_1 t_2 *linear* t_1 *linear* t_2
assumes *match* t_1 u = *Some env*$_1$ *match* t_2 u = *Some env*$_2$
shows $t_1 = t_2$
using *assms* **by** (*metis pattern-compatible-cases match-overlapping*)

corollary *match-compatible-env-eq*:
assumes *pattern-compatible* t_1 t_2 *linear* t_1 *linear* t_2
assumes *match* t_1 u = *Some env*$_1$ *match* t_2 u = *Some env*$_2$
shows *env*$_1$ = *env*$_2$
using *assms* **by** (*metis match-compatible-pat-eq option.inject*)

corollary *matchs-compatible-eq*:
assumes *patterns-compatible ts*$_1$ *ts*$_2$ *linears ts*$_1$ *linears ts*$_2$
assumes *matchs ts*$_1$ *us* = *Some env*$_1$ *matchs ts*$_2$ *us* = *Some env*$_2$
shows *ts*$_1$ = *ts*$_2$ *env*$_1$ = *env*$_2$
proof −
fix *name*
have *match* (*name* $$ *ts*$_1$) (*name* $$ *us*) = *Some env*$_1$ *match* (*name* $$ *ts*$_2$) (*name* $$ *us*) = *Some env*$_2$
using *assms* **by** *auto*
moreover have *length ts*$_1$ = *length ts*$_2$
using *assms* **by** (*metis matchs-some-eq-length*)
ultimately have *pattern-compatible* (*name* $$ *ts*$_1$) (*name* $$ *ts*$_2$)
using *assms* **by** (*auto simp*: *const-term-def*)
moreover have *linear* (*name* $$ *ts*$_1$) *linear* (*name* $$ *ts*$_2$)
using *assms* **by** (*auto intro*: *linear-list-comb'*)
note ∗ = *calculation*

from ∗ **have** *name* $$ *ts*$_1$ = *name* $$ *ts*$_2$
by (*rule match-compatible-pat-eq*) *fact*+
thus *ts*$_1$ = *ts*$_2$
by (*meson list-comb-inj-second injD*)

from ∗ **show** *env*$_1$ = *env*$_2$
by (*rule match-compatible-env-eq*) *fact*+
qed

lemma *compatible-find-match*:
  assumes *pattern-compatibles* (*fset-of-list cs*) *list-all* (*linear* ∘ *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 *pat′ rhs′* **where** [*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* ∘ *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 = -› **.**
qed *auto*

context *term* **begin**

definition *arity-compatible* :: $'a \Rightarrow 'a \Rightarrow$ *bool* **where**
arity-compatible t_1 t_2 = (
  let
    ($head_1$, $pats_1$) = *strip-comb* t_1;
    ($head_2$, $pats_2$) = *strip-comb* t_2
  in $head_1 = head_2 \longrightarrow$ *length* $pats_1$ = *length* $pats_2$
)

abbreviation *arity-compatibles* :: $('a \times 'b)$ *fset* $\Rightarrow$ *bool* **where**
arity-compatibles $\equiv$ *fpairwise* (λ(lhs_1, -) (lhs_2, -). *arity-compatible* lhs_1 lhs_2)

definition *head* :: $'a \Rightarrow$ *name* **where**
head $t \equiv$ *const-name* (*fst* (*strip-comb* t))

abbreviation *heads-of* :: (*term* $\times$ $'a$) *fset* $\Rightarrow$ *name fset* **where**
heads-of rs $\equiv$ (*head* $\circ$ *fst*) |‘| *rs*

end

definition *arity* :: $('a$ *list* $\times$ $'b)$ *fset* $\Rightarrow$ *nat* **where**
arity rs = *fthe-elem'* ((*length* $\circ$ *fst*) |‘| *rs*)

lemma *arityI*:
  assumes *fBall rs* (λ(*pats*, -). *length pats* = n) *rs* $\neq$ {||}
  shows *arity rs* = n
proof −
  have (*length* $\circ$ *fst*) |‘| *rs* = {| n |}
    using *assms* **by** *force*
  thus *?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

theory *Constructors*
imports
  Terms-Extras
  HOL-Datatype
begin

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 = *fmdom C-info*

definition *C* :: *name fset* **where**
C = *ffUnion* (*fmdom* |'| *constructors* |'| *fmran 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* |∪| *C*

abbreviation *welldefined* :: $'a$::*term* $\Rightarrow$ *bool* **where**
welldefined t $\equiv$ *consts t* $|\subseteq|$ *all-consts*

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* :: $'a \Rightarrow$ *bool*
fixes *all-frees* :: $'a \Rightarrow$ *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) $\longleftrightarrow$ *wellformed* u_1 $\wedge$ *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 $|\cup|$ *all-frees* u_2
begin

abbreviation *wellformed-env* :: (*name*, $'a$) *fmap* $\Rightarrow$ *bool* **where**
wellformed-env $\equiv$ *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* (λ*t*. *frees t* |⊆| *all-frees t*) *t*
 assumes *raw-subst-wellformed*: *abs-pred* (λ*t*. *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 (t_1 $ t_2) = *all-frees-term* t_1 |∪| *all-frees-term* t_2 |

all-frees-term (Λ *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* ⇒ *bool* **where**
[*simp*]: *wellformed-term t* ⟷ *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* ⇒ *bool* **where**
[*simp*]: *wellformed-nterm t* ⟷ *True*

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

instance proof (*standard*, *goal-cases*)
  case (*7 t*)
  show *?case*
    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 ¬ *shadows-consts t* ⟹ *fdisjnt all-consts* (*frees t*)
unfolding *fdisjnt-alt-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* (‹Λ_s -. -› [*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 $\bigwedge x.\ P\ (Sconst\ x)$
  assumes $\bigwedge x.\ P\ (Svar\ x)$
  assumes $\bigwedge cs.\ (\bigwedge pat\ t.\ (pat,\ t) \in set\ cs \Longrightarrow P\ t) \Longrightarrow P\ (Sabs\ cs)$
  assumes $\bigwedge t\ u.\ P\ t \Longrightarrow P\ u \Longrightarrow P\ (t\ \$_s\ u)$
  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*) |

unapp-sterm - = *None*

definition *const-sterm* **where**
const-sterm = *Sconst*

fun *unconst-sterm* **where**
unconst-sterm (*Sconst name*) = *Some name* |
unconst-sterm - = *None*

fun *unfree-sterm* **where**
unfree-sterm (*Svar name*) = *Some name* |
unfree-sterm - = *None*

definition *free-sterm* **where**
free-sterm = *Svar*

fun *frees-sterm* **where**
frees-sterm (*Svar name*) = {|*name*|} |
frees-sterm (*Sconst* -) = {||} |
frees-sterm (*Sabs cs*) = *ffUnion* (*fset-of-list* (*map* (λ(*pat*, *rhs*). *frees-sterm rhs* − *frees pat*) *cs*)) |
frees-sterm (t $\$_s$ u) = *frees-sterm* t |∪| *frees-sterm* u

fun *subst-sterm* **where**
subst-sterm (*Svar s*) *env* = (*case fmlookup env s of Some t* ⇒ *t* | *None* ⇒ *Svar s*) |
subst-sterm (t_1 $\$_s$ t_2) *env* = *subst-sterm* t_1 *env* $\$_s$ *subst-sterm* t_2 *env* |
subst-sterm (*Sabs cs*) *env* = *Sabs* (*map* (λ(*pat*, *rhs*). (*pat*, *subst-sterm rhs* (*fmdrop-fset* (*frees pat*) *env*))) *cs*) |
subst-sterm t env = *t*

fun *consts-sterm* :: *sterm* ⇒ *name fset* **where**
consts-sterm (*Svar* -) = {||} |
consts-sterm (*Sconst name*) = {|*name*|} |
consts-sterm (*Sabs cs*) = *ffUnion* (*fset-of-list* (*map* (λ(-, *rhs*). *consts-sterm rhs*) *cs*)) |
consts-sterm (t $\$_s$ u) = *consts-sterm* t |∪| *consts-sterm* u

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

end

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

definition *abs-pred-sterm* :: (*sterm* ⇒ *bool*) ⇒ *sterm* ⇒ *bool* **where**
[*code del*]: *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*)
  case (*1 P t*)
  then show *?case*
   by (*induction t*) (*auto simp*: *const-sterm-def free-sterm-def app-sterm-def abs-pred-sterm-def*)
next
  case (*2 t*)
  show *?case*
    apply *rule*
    apply *auto*
    apply (*subst* (*3*) *list.map-id*[*symmetric*])
    apply (*rule list.map-cong0*)
    apply *auto*
    by *blast*
next
  case (*3 x t*)
  show *?case*
    apply *rule*
    apply *clarsimp*
    subgoal for *cs env pat rhs*
      apply (*cases x* |∈| *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 *fact*
        subgoal
          using *prems*
          unfolding *ffUnion-alt-def*
          by (*auto simp add*: *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*
    apply *rule*

```
    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-stermI 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 *name* |∈| *frees rhs* |−| *fmdom* (*fmdrop-fset* (*frees pat*) *env*) |−| *frees pat*
using ‹*name* |∉| *fmdom env*› **by** *fastforce*

ultimately have *name* |∈| *frees* (*subst rhs* (*fmdrop-fset* (*frees pat*) *env*)) |−| *frees pat*
using *1* **by** (*simp add*: *fmpred-drop-fset*)

show *name* |∈| *frees* (*subst* (*Sabs cs*) *env*)
apply (*simp add*: *case-prod-twice comp-def*)
unfolding *ffUnion-alt-def*
apply (*rule fBexI*)
apply (*fact* ‹*name* |∈| *frees* (*subst rhs* (*fmdrop-fset* (*frees pat*) *env*)) |−| *frees pat*›)
apply (*subst fimage-iff*)
apply (*rule fBexI*[**where** *x* = (*pat*, *rhs*)])
apply *simp*
using ‹(*pat*, *rhs*) ∈ *set cs*›
by (*auto simp*: *fset-of-list-elem*)
qed
qed
next
case *6*
show *?case*
proof (*intro abs-pred-stermI allI impI*, *goal-cases*)
case (*1 cs env*)

— some property on various operations that is only useful in here
have ∗: *fbind* (*fmimage m* (*fbind A g*)) *f* = *fbind A* (λ*x*. *fbind* (*fmimage m* (*g x*)) *f*)
for *m A f g*
including *fset.lifting* **and** *fmap.lifting*
by *transfer′ force*

have *consts* (*subst* (*Sabs cs*) *env*) = *fbind* (*fset-of-list cs*) (λ(*pat*, *rhs*). *consts rhs* |∪| *ffUnion* (*consts* |‘| *fmimage* (*fmdrop-fset* (*frees pat*) *env*) (*frees rhs*)))
apply (*simp add*: *funion-image-bind-eq*)
apply (*rule fbind-cong*[*OF refl*])
apply (*clarsimp split*: *prod.splits*)
apply (*subst 1*)
apply (*subst* (*asm*) *fset-of-list-elem*, *assumption*)
apply *simp*
by (*simp add*: *funion-image-bind-eq*)
also have . . . = *fbind* (*fset-of-list cs*) (*consts* ∘ *snd*) |∪| *fbind* (*fset-of-list cs*) (λ(*pat*, *rhs*). *ffUnion* (*consts* |‘| *fmimage* (*fmdrop-fset* (*frees pat*) *env*) (*frees rhs*)))
apply (*subst fbind-fun-funion*[*symmetric*])
apply (*rule fbind-cong*[*OF refl*])
by *auto*
also have . . . = *consts* (*Sabs cs*) |∪| *fbind* (*fset-of-list cs*) (λ(*pat*, *rhs*). *ffUnion*

(*consts* |'| *fmimage* (*fmdrop-fset* (*frees pat*) *env*) (*frees rhs*)))
  apply (*rule cong*[*OF cong*, *OF refl* - *refl*, **where** *f1* = *funion*])
  apply (*subst funion-image-bind-eq*[*symmetric*])
  unfolding *consts-sterm.simps*
  apply (*rule arg-cong*[**where** *f* = *ffUnion*])
  apply (*subst fset-of-list-map*)
  apply (*rule fset.map-cong*[*OF refl*])
  by *auto*
 also have ... = *consts* (*Sabs cs*) |∪| *fbind* (*fmimage env* (*fbind* (*fset-of-list cs*) (λ(*pat*, *rhs*). *frees rhs* |−| *frees pat*))) *consts*
  apply (*subst funion-image-bind-eq*)
  apply (*subst fmimage-drop-fset*)
  apply (*rule cong*[*OF cong*, *OF refl refl*, **where** *f1* = *funion*])
  apply (*subst* ∗)
  apply (*rule fbind-cong*[*OF refl*])
  by *auto*
 also have ... = *consts* (*Sabs cs*) |∪| *ffUnion* (*consts* |'| *fmimage env* (*frees* (*Sabs cs*)))
  by (*simp only*: *frees-sterm.simps fset-of-list-map fmimage-Union funion-image-bind-eq*)

 finally show *?case* **.**
 qed
qed (*auto simp*: *abs-pred-sterm-def*)

end

lemma *no-abs-abs*[*simp*]: ¬ *no-abs* (*Sabs cs*)
by (*subst no-abs.simps*) (*auto simp*: *term-cases-def*)

lemma *closed-except-simps*:
 closed-except (*Svar x*) *S* ⟷ *x* |∈| *S*
 closed-except (t_1 $\$_s$ t_2) *S* ⟷ *closed-except* t_1 *S* ∧ *closed-except* t_2 *S*
 closed-except (*Sabs cs*) *S* ⟷ *list-all* (λ(*pat*, *t*). *closed-except t* (*S* |∪| *frees pat*)) *cs*
 closed-except (*Sconst name*) *S* ⟷ *True*
proof *goal-cases*
 case *3*
 show *?case*
  proof (*standard*, *goal-cases*)
   case *1*
   then show *?case*
   apply (*auto simp*: *list-all-iff ffUnion-alt-def fset-of-list-elem closed-except-def*)
    apply (*drule ffUnion-least-rev*)
    apply *auto*
    by (*smt case-prod-conv fbspec fimageI fminusI fset-of-list-elem fset-rev-mp*)
  next
   case *2*
   then show *?case*

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*) $\in$ *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* $\Rightarrow$ *bool* **where**
wellformed-sterm (t_1 $\$_s$ t_2) $\longleftrightarrow$ *wellformed-sterm* t_1 $\wedge$ *wellformed-sterm* t_2 |
wellformed-sterm (*Sabs cs*) $\longleftrightarrow$ *list-all* (λ(*pat*, *t*). *linear pat* $\wedge$ *wellformed-sterm t*) *cs* $\wedge$ *distinct* (*map fst cs*) $\wedge$ *cs* $\neq$ [] |
wellformed-sterm - $\longleftrightarrow$ *True*

primrec *all-frees-sterm* :: *sterm* $\Rightarrow$ *name fset* **where**
all-frees-sterm (*Svar x*) = {|*x*|} |
all-frees-sterm (t_1 $\$_s$ t_2) = *all-frees-sterm* t_1 |$\cup$| *all-frees-sterm* t_2 |
all-frees-sterm (*Sabs cs*) = *ffUnion* (*fset-of-list* (*map* (λ(*P*, *T*). *P* |$\cup$| *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 impI*)
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 impI*)
apply (*simp add*: *list.pred-map comp-def case-prod-twice*, *safe*)
subgoal
apply (*subst list-all-iff*)
apply (*rule ballI*)
apply *safe*[*1*]
apply (*fastforce simp*: *list-all-iff*)

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* $\Longrightarrow$ *match t* (*Sabs cs*) = *None*
by (*cases t*) *auto*

context *pre-constants* **begin**

lemma *welldefined-sabs*: *welldefined* (*Sabs cs*) $\longleftrightarrow$ *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 (t_1 $\$_s$ t_2) $\longleftrightarrow$ *shadows-consts* t_1 $\vee$ *shadows-consts* t_2
shadows-consts (*Svar name*) $\longleftrightarrow$ *name* |∈| *all-consts*
shadows-consts (*Sabs cs*) $\longleftrightarrow$ *list-ex* (λ(*pat*, *t*). ¬ *fdisjnt all-consts* (*frees pat*) $\vee$ *shadows-consts t*) *cs*
shadows-consts (*Sconst name*) $\longleftrightarrow$ *False*
proof (*goal-cases*)
case *3*
show *?case*
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-consts* (*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*)
      apply (*rule Sabs*(*1*))
        apply *assumption*
      subgoal using *Sabs*(*2*) **by** (*fastforce simp*: *list-ex-iff*)
      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-pterm* :: *sterm* ⇒ *pterm* **where**
sterm-to-pterm (*Sconst name*) = *Pconst name* |
sterm-to-pterm (*Svar name*) = *Pvar name* |
sterm-to-pterm (*t* $\$_s$ *u*) = *sterm-to-pterm t* $\$_p$ *sterm-to-pterm u* |
sterm-to-pterm (*Sabs cs*) = *Pabs* (*fset-of-list* (*map* (*map-prod id sterm-to-pterm*) *cs*))

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

lemma *sterm-to-pterm-total*:
  obtains t' **where** t = *sterm-to-pterm* t'
proof (*induction t arbitrary*: *thesis*)
  case (*Pconst x*)
  then show *?case*
    by (*metis sterm-to-pterm.simps*)
next
  case (*Pvar x*)
  then show *?case*
    by (*metis sterm-to-pterm.simps*)
next
  case (*Pabs cs*)
  from *Pabs.IH* **obtain** cs' **where** *cs* = *fset-of-list* (*map* (*map-prod id sterm-to-pterm*) cs')
    apply *atomize-elim*
    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 ∃ cs'. *cs* = *fset-of-list* (*map* (*map-prod id sterm-to-pterm*) cs')
        apply (*rule insert*)
        using *insert.prems* **unfolding** *finsert.rep-eq*
        by *blast*
      then obtain cs' **where** *cs* = *fset-of-list* (*map* (*map-prod id sterm-to-pterm*) cs')
        by *blast*

      obtain rhs' **where** *rhs* = *sterm-to-pterm* rhs'
        apply (*rule insert.prems*[*of* (*pat*, *rhs*) *rhs*])
        unfolding ‹*c* = -› **by** *simp*+

      show *?case*
        apply (*rule exI*[**where** *x* = (*pat*, rhs') # cs'])
        unfolding ‹*c* = -› ‹*cs* = -› ‹*rhs* = -›
        by (*simp add*: *id-def*)
    qed
  hence *Pabs cs* = *sterm-to-pterm* (*Sabs* cs')
    by *simp*
  then show *?case*
    using *Pabs* **by** *metis*
next
  case (*Papp t1 t2*)
  then obtain $t1'$ $t2'$ **where** *t1* = *sterm-to-pterm* $t1'$ *t2* = *sterm-to-pterm* $t2'$
    by *metis*

then have $t1\ \$_p\ t2 = sterm\text{-}to\text{-}pterm\ (t1'\ \$_s\ t2')$
by *simp*
with *Papp* **show** *?case*
by *metis*
qed

lemma *pterm-induct*[*case-names Pconst Pvar Pabs Papp*]:
assumes $\bigwedge x.\ P\ (Pconst\ x)$
assumes $\bigwedge x.\ P\ (Pvar\ x)$
assumes $\bigwedge cs.\ (\bigwedge pat\ t.\ (pat,\ t)\ |\in|\ cs \Longrightarrow P\ t) \Longrightarrow P\ (Pabs\ cs)$
assumes $\bigwedge t\ u.\ P\ t \Longrightarrow P\ u \Longrightarrow P\ (t\ \$_p\ u)$
shows $P\ t$
proof (*rule pterm.induct*, *goal-cases*)
case (*3 cs*)
show *?case*
apply (*rule assms*)
using *3*
by *auto*
qed *fact+*

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

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

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

definition *const-pterm* **where**
const-pterm = *Pconst*

fun *unconst-pterm* **where**
unconst-pterm (*Pconst name*) = *Some name* |
unconst-pterm - = *None*

definition *free-pterm* **where**
free-pterm = *Pvar*

fun *unfree-pterm* **where**
unfree-pterm (*Pvar name*) = *Some name* |
unfree-pterm - = *None*

function (*sequential*) *subst-pterm* **where**
subst-pterm (*Pvar s*) *env* = (*case fmlookup env s of Some t* ⇒ *t* | *None* ⇒ *Pvar s*) |
$subst\text{-}pterm\ (t_1\ \$_p\ t_2)\ env = subst\text{-}pterm\ t_1\ env\ \$_p\ subst\text{-}pterm\ t_2\ env\ |$
subst-pterm (*Pabs cs*) *env* = *Pabs* ((λ(*pat*, *rhs*). (*pat*, *subst-pterm rhs* (*fmdrop-fset* (*frees pat*) *env*))) |\`| *cs*) |

subst-pterm t - = t
by *pat-completeness auto*

termination
proof (*relation measure* (*size* ∘ *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* ⇒ *name fset* **where**
consts-pterm (*Pconst x*) = {|*x*|} |
consts-pterm (t_1 $\$_p$ t_2) = *consts-pterm* t_1 |∪| *consts-pterm* t_2 |
consts-pterm (*Pabs cs*) = *ffUnion* (*snd* |‘| *map-prod id consts-pterm* |‘| *cs*) |
consts-pterm (*Pvar* -) = {||}

primrec *frees-pterm* :: *pterm* ⇒ *name fset* **where**
frees-pterm (*Pvar x*) = {|*x*|} |
frees-pterm (t_1 $\$_p$ t_2) = *frees-pterm* t_1 |∪| *frees-pterm* t_2 |
frees-pterm (*Pabs cs*) = *ffUnion* ((λ(*pv*, *tv*). *tv* − *frees pv*) |‘| *map-prod id frees-pterm* |‘| *cs*) |
frees-pterm (*Pconst* -) = {||}

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

end

corollary *subst-pabs-id*:
  assumes ⋀*pat rhs*. (*pat*, *rhs*) |∈| *cs* ⟹ *subst rhs* (*fmdrop-fset* (*frees pat*) *env*) = *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 *assumption*
done

corollary *frees-pabs-alt-def*:
  frees (*Pabs cs*) = *ffUnion* ((λ(*pat*, *rhs*). *frees rhs* − *frees pat*) |‘| *cs*)

apply *simp*
apply (*rule arg-cong*[**where** *f* = *ffUnion*])
apply (*rule fset.map-cong*[*OF 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* = *ffUnion*])
    apply (*rule fimage-cong*[*OF 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* = *ffUnion*])
    apply (*rule fimage-cong*[*OF refl*])
    apply *clarsimp*
    apply (*subst Sabs*)
    by (*auto simp*: *fset-of-list-elem snds.simps*)
qed *auto*

lemma *subst-sterm-to-pterm*:
  subst (*sterm-to-pterm t*) (*fmmap sterm-to-pterm env*) = *sterm-to-pterm* (*subst t env*)
proof (*induction t arbitrary*: *env rule*: *sterm-induct*)
  case (*Sabs cs*)
  show *?case*
    apply *simp*
    apply (*rule fset.map-cong0*)
    apply (*auto split*: *prod.splits*)
    apply (*rule Sabs*)
    by (*auto simp*: *fset-of-list.rep-eq*)
qed (*auto split*: *option.splits*)

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

definition *abs-pred-pterm* :: (*pterm* ⇒ *bool*) ⇒ *pterm* ⇒ *bool* **where**
[*code del*]: *abs-pred P t* ⟷ (∀ *cs*. *t* = *Pabs cs* ⟶ (∀ *pat t*. (*pat*, *t*) |∈| *cs* ⟶ *P t*) ⟶ *P t*)

context begin

```
private lemma abs-pred-trivI0: P t ⟹ abs-pred P (t::pterm)
unfolding abs-pred-pterm-def by auto

instance proof (standard, goal-cases)
  case (1 P t)
  then show ?case
    by (induction t rule: pterm-induct)
      (auto simp: const-pterm-def free-pterm-def app-pterm-def abs-pred-pterm-def)
next
  case (2 t)
  show ?case
    unfolding abs-pred-pterm-def
    apply clarify
    apply (rule subst-pabs-id)
    by blast
next
  case (3 x t)
  show ?case
    unfolding abs-pred-pterm-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 |∈| 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 fact
        subgoal
          using prems unfolding ffUnion-alt-def
          by (auto simp: 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)[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 (subst subst-sterm-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 = -› ‹env = -›
        apply simp
```

apply (*subst comp-def*)
apply *simp*
apply (*subst subst-sterm-to-pterm*)
apply *simp*
apply (*rule subst-consts'*)
done
qed
qed (*rule abs-pred-trivI0*)

end

end

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

lemma *sterm-to-pterm*:
assumes *no-abs t*
shows *sterm-to-pterm 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* (‹Λ_p -. -› [*0*, *50*] *50*) **where**
Pabs-single x rhs ≡ *Pabs* {| (*Free x*, *rhs*) |}

lemma *closed-except-simps*:
closed-except (*Pvar x*) *S* ⟷ *x* |∈| *S*
closed-except (t_1 $\$_p$ t_2) *S* ⟷ *closed-except* t_1 *S* ∧ *closed-except* t_2 *S*
closed-except (*Pabs cs*) *S* ⟷ *fBall cs* (λ(*pat*, *t*). *closed-except t* (*S* |∪| *frees pat*))
closed-except (*Pconst name*) *S* ⟷ *True*
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-pterm* :: *pterm* ⇒ *bool* **where**
wellformed-pterm (t_1 $\$_p$ t_2) ⟷ *wellformed-pterm* t_1 ∧ *wellformed-pterm* t_2 |
wellformed-pterm (*Pabs cs*) ⟷ *fBall cs* (λ(*pat*, *t*). *linear pat* ∧ *wellformed-pterm*
t) ∧ *is-fmap cs* ∧ *pattern-compatibles cs* ∧ *cs* ≠ {||} |
wellformed-pterm - ⟷ *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-pterm* :: *pterm* ⇒ *name fset* **where**
all-frees-pterm (*Pvar x*) = {|*x*|} |
all-frees-pterm (t_1 $\$_p$ t_2) = *all-frees-pterm* t_1 |∪| *all-frees-pterm* t_2 |
all-frees-pterm (*Pabs cs*) = *ffUnion* ((λ(*P*, *T*). *P* |∪| *T*) |'| *map-prod frees all-frees-pterm*
|'| *cs*) |
all-frees-pterm (*Pconst* -) = {||}

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

end

lemma *sterm-to-pterm-all-frees*[*simp*]: *all-frees* (*sterm-to-pterm t*) = *all-frees t*
proof (*induction t*)

case (*Sabs cs*)
show *?case*
apply *simp*
apply (*rule arg-cong*[**where** *f* = *ffUnion*])
apply (*rule fimage-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 = *sterm-to-pterm* t'
by (*metis sterm-to-pterm-total*)
show *?case*
apply (*rule abs-pred-trivI*)
unfolding ‹*t* = -› *sterm-to-pterm-all-frees sterm-to-pterm-frees*
by (*rule frees-all-frees*)
next
case (*2 t*)
show *?case*
unfolding *abs-pred-pterm-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*. (*pat*, *t*) |∈| *cs* $\Longrightarrow$ *linear pat*
assumes $\bigwedge$*pat t*. (*pat*, *t*) |∈| *cs* $\Longrightarrow$ *wellformed t*
shows *wellformed* (*Pabs cs*)
using *assms* **by** *auto*

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

lemma (**in** *constants*) *shadows-consts-pterm-simps*[*simp*]:
shadows-consts (t_1 $\$_p$ t_2) $\longleftrightarrow$ *shadows-consts* t_1 $\vee$ *shadows-consts* t_2
shadows-consts (*Pvar name*) $\longleftrightarrow$ *name* |∈| *all-consts*
shadows-consts (*Pabs cs*) $\longleftrightarrow$ *fBex cs* (λ(*pat*, *t*). *shadows-consts pat* $\vee$ *shadows-consts t*)

shadows-consts (*Pconst name*) $\longleftrightarrow$ *False*
proof *goal-cases*
case *3*

show *?case*
unfolding *shadows-consts-def*
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*)

end

1.8 Irreducible terms (values)

theory *Term-as-Value*
imports *Sterm*
begin

1.9 Viewing *sterm* as values

declare *list.pred-mono*[*mono*]

context *constructors* **begin**

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

lemma *value-distinct*:
Sabs cs $\neq$ *name* $$ *ts* (**is** *?P*)
name $$ *ts* $\neq$ *Sabs cs* (**is** *?Q*)
proof −
show *?P*
apply (*rule list-comb-cases'*[**where** *f* = *const name* **and** *xs* = *ts*])
apply (*auto simp*: *const-sterm-def is-app-def elim*: *unapp-sterm.elims*)
done
thus *?Q*
by *simp*
qed

abbreviation *value-env* :: (*name*, *sterm*) *fmap* $\Rightarrow$ *bool* **where**

```
value-env ≡ fmpred (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 |∈| 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 |∈| C
        unfolding Sapp
        by cases auto
      thus thesis using comb by simp
    qed
qed

end

fun smatch' :: pat ⇒ sterm ⇒ (name, sterm) fmap option where
smatch' (Patvar name) t = Some (fmap-of-list [(name, t)]) |
smatch' (Patconstr name ps) t =
```

(*case strip-comb t of*
  (*Sconst name'*, *vs*) ⇒
    (*if name* = *name'* ∧ *length ps* = *length vs then*
      map-option (*foldl* (++$_f$) *fmempty*) (*those* (*map2 smatch' ps vs*))
    else
      None)
| - ⇒ *None*)

lemmas *smatch'-induct* = *smatch'.induct*[*case-names var constr*]

context *constructors* **begin**

context begin

private lemma *smatch-list-comb-is-value*:
  assumes *is-value t*
  shows *match* (*name* $$ *ps*) *t* = (*case strip-comb t of*
  (*Sconst name'*, *vs*) ⇒
    (*if name* = *name'* ∧ *length ps* = *length vs then*
      map-option (*foldl* (++$_f$) *fmempty*) (*those* (*map2 match ps vs*))
    else
      None)
| - ⇒ *None*)
using *assms*
apply *cases*
apply (*auto simp*: *strip-list-comb split*: *option.splits*)
apply (*subst* (*2*) *const-sterm-def*)
apply (*auto simp*: *matchs-alt-def*)
done

lemma *smatch-smatch'-eq*:
  assumes *linear pat is-value t*
  shows *match pat t* = *smatch'* (*mk-pat pat*) *t*
using *assms*
proof (*induction pat arbitrary*: *t rule*: *linear-pat-induct*)
  case (*comb name args*)

  show *?case*
    using ‹*is-value t*›
    proof (*cases rule*: *is-value.cases*)
      case (*abs cs*)
      thus *?thesis*
        by (*force simp*: *strip-list-comb-const*)
    next
      case (*constr args' name'*)

      have *map2 match args args'* = *map2 smatch'* (*map mk-pat args*) *args'* **if** *length args* = *length args'*
        using *that constr*(*2*) *comb*(*2*)

by (*induct args 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*: *matchs-alt-def*)
done
qed
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* × *value*

setup ‹*Sign.mandatory-path quickcheck*›

datatype *value* =
Vconstr name value list |
Vabs sclauses (*name* × *value*) *list* |
Vrecabs (*name* × *sclauses*) *list name* (*name* × *value*) *list*

primrec *Value* :: *quickcheck.value* ⇒ *value* **where**
Value (*quickcheck.Vconstr s vs*) = *Vconstr s* (*map 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-of-list* [(*name*, *v*)]) |
vmatch (*Patconstr name ps*) (*Vconstr name′ vs*) =
  (*if name* = *name′* ∧ *length ps* = *length vs then*
    map-option (*foldl* ($++_f$) *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 del*]

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 *fmpred* (λ-. *pred*) *env*
using *assms* **proof** (*induction pat v arbitrary*: *env rule*: *vmatch-induct*)
  case (*constr name ps name′ vs*)

hence
map-option (*foldl* $(++_f)$ *fmempty*) (*those* (*map2 vmatch ps vs*)) = *Some env*
name = *name′ length ps* = *length vs*
by (*auto split*: *if-splits*)
then obtain *envs* **where** *env* = *foldl* $(++_f)$ *fmempty envs map2 vmatch ps vs* = *map Some envs*
by (*blast dest*: *those-someD*)

moreover have *fmpred* (λ-. *pred*) *env* **if** *env* ∈ *set envs* **for** *env*
proof −
from *that* **have** *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*
apply (*rule map2-elemE*)
by *auto*
hence *pred v*
using *constr* **by** (*simp add*: *list-all-iff*)
show *?thesis*
by (*rule constr*; *safe?*) *fact+*
qed

ultimately show *?case*
by *auto*
qed *auto*

end

primrec *value-to-sterm* :: *value* ⇒ *sterm* **where**
value-to-sterm (*Vconstr name vs*) = *name* $$ *map value-to-sterm vs* |
value-to-sterm (*Vabs cs* Γ) = *Sabs* (*map* (λ(*pat*, *t*). (*pat*, *subst t* (*fmdrop-fset* (*frees pat*) (*fmmap value-to-sterm* Γ)))) *cs*) |
value-to-sterm (*Vrecabs css name* Γ) =
Sabs (*map* (λ(*pat*, *t*). (*pat*, *subst t* (*fmdrop-fset* (*frees pat*) (*fmmap value-to-sterm* Γ)))) (*the* (*fmlookup css name*)))

This locale establishes a connection between a predicate on *value*s with the corresponding predicate on *sterm*s, by means of *value-to-sterm*.

locale *pre-value-sterm-pred* = *value-pred* +
fixes *S*
assumes *value-to-sterm*: *pred v* ⟹ *S* (*value-to-sterm v*)
begin

corollary *value-to-sterm-env*:
assumes *fmpred* (λ-. *pred*) Γ
shows *fmpred* (λ-. *S*) (*fmmap value-to-sterm* Γ)
unfolding *fmpred-map* **proof**
fix *name v*
assume *fmlookup* 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*: $\bigwedge$*name*. Q *name* $\Longrightarrow$ S (*const name*)
assumes *abs*: $\bigwedge\Gamma$ *cs*.
($\bigwedge$*n v*. *fmlookup* Γ *n* = *Some v* $\Longrightarrow$ *pred v* $\Longrightarrow$ S (*value-to-sterm v*)) $\Longrightarrow$
fmpred (λ-. *pred*) Γ $\Longrightarrow$
P Γ *cs* $\Longrightarrow$
S (*Sabs* (*map* (λ(*pat*, *t*). (*pat*, *subst t* (*fmmap value-to-sterm* (*fmdrop-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'I*)
next
case (*Vrecabs x1 x2 x3*)
show *?case*
apply *auto*
apply (*rule abs*)
using *Vrecabs*
by (*auto simp*: *fmlookup-dom-iff fmpred-iff intro*: *fmran'I*)
qed
qed

end

global-interpretation *vwellformed*:
value-sterm-pred

λ-. *wellformed-clauses*
λ-. *True*
λ-. *True*
wellformed
defines *vwellformed* = *vwellformed.pred*
proof (*standard*, *goal-cases*)
case (*2* Γ *cs*)
hence *cs* $\neq$ []
by *simp*

moreover have *wellformed* (*subst rhs* (*fmdrop-fset* (*frees pat*) (*fmmap value-to-sterm* Γ)))
if (*pat*, *rhs*) $\in$ *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* $\circ$ (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* $\equiv$ *fmpred* (λ-. *vwellformed*)

global-interpretation *vclosed*:
value-sterm-pred
$\lambda\Gamma$ *cs*. *list-all* (λ(*pat*, *t*). *closed-except t* (*fmdom* Γ |$\cup$| *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-twice Sterm.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*
λ*name*. *name* |∈| *C*
λ*dom*. *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 subst-consts*)
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
qed (*simp add*: *all-consts-def*)

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* ≡ *fmpred* (*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*). *fdisjnt 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 fmpred-drop-fset)
      apply (rule fmpredI)
      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 ≡ fmpred (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 ∘ 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-sterm (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* ($++_f$) *fmempty*) (*those* (*map2 vmatch ps vs*)) = *Some env*
name = *name′ length ps* = *length vs*
by (*auto split*: *if-splits*)
then obtain *envs* **where** *env* = *foldl* ($++_f$) *fmempty 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* ∈ *set vs vmatch p v* ∈ *set* (*map2 vmatch ps vs*)
using ‹*length ps* = *length vs*› **by** (*auto elim*!: *map2-elemE1*)
then obtain *env* **where** *env* ∈ *set envs vmatch p v* = *Some env*

unfolding ‹*map2 vmatch ps vs* = -› **by** *auto*
moreover hence *fmdom env* = *patvars p*
using *constr* ‹*name* = *name'*› ‹*length ps* = *length vs*› ‹*p* ∈ *set ps*› ‹*v* ∈ *set vs*›
by *fastforce*
ultimately have *names* ∈ *set* (*map fmdom envs*)
unfolding ‹*names* = -› **by** *auto*
thus *names* |∈| *fset-of-list* (*map fmdom envs*)
unfolding *fset-of-list-elem* **.**
qed

ultimately show *?case*
by (*auto simp*: *fmdom-foldl-add*)
qed *auto*

fun *vfind-match* :: *sclauses* ⇒ *value* ⇒ ((*name*, *value*) *fmap* × *term* × *sterm*) *option* **where**
vfind-match [] - = *None* |
vfind-match ((*pat*, *rhs*) # *cs*) *t* =
(*case vmatch* (*mk-pat pat*) *t of*
Some env ⇒ *Some* (*env*, *pat*, *rhs*)
| *None* ⇒ *vfind-match cs t*)

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

inductive *veq-structure* :: *value* ⇒ *value* ⇒ *bool* **where**
abs-abs: *veq-structure* (*Vabs* - -) (*Vabs* - -) |
recabs-recabs: *veq-structure* (*Vrecabs* - - -) (*Vrecabs* - - -) |
constr-constr: *list-all2 veq-structure ts us* ⟹ *veq-structure* (*Vconstr name ts*) (*Vconstr name us*)

lemma *veq-structure-simps*[*code*, *simp*]:
veq-structure (*Vabs* cs_1 Γ_1) (*Vabs* cs_2 Γ_2)
veq-structure (*Vrecabs* css_1 $name_1$ Γ_1) (*Vrecabs* css_2 $name_2$ Γ_2)
veq-structure (*Vconstr* $name_1$ *ts*) (*Vconstr* $name_2$ *us*) ⟷ $name_1 = name_2$ ∧ *list-all2 veq-structure ts us*
by (*auto intro*: *veq-structure.intros elim*: *veq-structure.cases*)

lemma *veq-structure-refl*[*simp*]: *veq-structure t t*
by (*induction t*) (*auto simp*: *list.rel-refl-strong*)

global-interpretation *vno-abs*: *value-pred* λ- -. *False* λ-. *True* λ-. *False*
defines *vno-abs* = *vno-abs.pred* **.**

lemma *veq-structure-eq-left*:

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

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

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

lemma *vmatch-vmatch'-eq*: *vmatch p v* = *vmatch' p v*
proof (*induction rule*: *vmatch.induct*)
case (*2 name ps name' vs*)
then show *?case*
apply *auto*
apply (*rule map-option-cong*[*OF - refl*])
apply (*rule arg-cong*[**where** f = *those*])
apply (*rule map2-cong*[*OF refl refl*])
apply *blast*
done
qed *auto*

locale *value-struct-rel* =
fixes Q :: *value* ⇒ *value* ⇒ *bool*
assumes *Q-impl-struct*: $Q\ t_1\ t_2 \Longrightarrow$ *veq-structure* $t_1\ t_2$

assumes *Q-def*[*simp*]: Q (*Vconstr name ts*) (*Vconstr name′ us*) $\longleftrightarrow$ *name* = *name′* $\wedge$ *list-all2 Q ts us*
begin

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

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

context begin

private lemma *vmatch′-rel*:
assumes $Q\ t_1\ t_2$
shows *rel-option* (*fmrel Q*) (*vmatch′ p* t_1) (*vmatch′ p* t_2)
using *assms*(*1*) **proof** (*induction p arbitrary*: $t_1\ t_2$)
case (*Patconstr name ps*)
with *Q-impl-struct* **have** *veq-structure* $t_1\ t_2$
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 ‹t_1 = -› ‹t_2 = -›
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*) (*map2 vmatch′ ps us*)
apply (*rule Cons*)
subgoal

apply (*rule Cons*)
unfolding ‹*ps0* = -› **apply** *simp*
by *assumption*
subgoal
using ‹*Q* (*Vconstr name′* (*t* # *ts*)) (*Vconstr name′* (*u* # *us*))› ‹*length ts* = *length us*›
by (*simp add*: *list-all-iff*)
subgoal
using ‹*length ps0* = *length* (*t* # *ts*)›
unfolding ‹*ps0* = -› **by** *simp*
done

ultimately show *?case*
unfolding ‹*ps0* = -›
by *auto*
qed *auto*

hence *rel-option* (*list-all2* (*fmrel Q*)) (*those* (*map2 vmatch′ ps ts*)) (*those* (*map2 vmatch′ ps us*))
by (*rule rel-funD*[*OF those-transfer*])

have
rel-option (*fmrel Q*)
(*map-option* (*foldl* ($++_f$) *fmempty*) (*those* (*map2 vmatch′ ps ts*)))
(*map-option* (*foldl* ($++_f$) *fmempty*) (*those* (*map2 vmatch′ ps us*)))
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 = -› ‹t_2 = -›
apply *auto*
apply (*rule* ∗)
by *simp*
qed *auto*
qed *auto*

lemma *vmatch-rel*: $Q\ t_1\ t_2 \Longrightarrow$ *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*) cs_1 cs_2
assumes $Q\ t_1\ t_2$
shows *rel-option* (*rel-prod* (*fmrel Q*) (*rel-prod* (=) *R*)) (*vfind-match* cs_1 t_1)

($vfind\text{-}match\ cs_2\ t_2$)
using *assms*(*1*) **proof** *induction*
  case ($Cons\ c_1\ cs_1\ c_2\ cs_2$)
  moreover obtain $pat_1\ rhs_1$ **where** $c_1 = (pat_1,\ rhs_1)$ **by** *fastforce*
  moreover obtain $pat_2\ rhs_2$ **where** $c_2 = (pat_2,\ rhs_2)$ **by** *fastforce*

  ultimately have $pat_1 = pat_2\ R\ rhs_1\ rhs_2$
    by *auto*

  have *rel-option* (*fmrel Q*) (*vmatch* (*mk-pat* pat_1) t_1) (*vmatch* (*mk-pat* pat_1) t_2)
    by (*rule vmatch-rel*) *fact*
  thus *?case*
    proof *cases*
      case *None*
      thus *?thesis*
        unfolding ‹$c_1 = -$› ‹$c_2 = -$› ‹$pat_1 = -$›
        using *Cons* **by** *auto*
    next
      case (*Some* $\Gamma_1\ \Gamma_2$)
      thus *?thesis*
        unfolding ‹$c_1 = -$› ‹$c_2 = -$› ‹$pat_1 = -$›
        using ‹$R\ rhs_1\ rhs_2$›
        by *auto*
    qed
qed *simp*

lemmas *vfind-match-rel*′ =
  vfind-match-rel[
    where $R = (=)$ **and** $cs_1 = cs$ **and** $cs_2 = cs$ **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**
$env\text{-}eq \equiv fmrel\ (\lambda v\ t.\ t = value\text{-}to\text{-}sterm\ v)$

lemma *env-eq-eq*:
  assumes *env-eq venv senv*
  shows $senv = fmmap\ value\text{-}to\text{-}sterm\ venv$
proof (*rule fmap-ext*, *unfold fmlookup-map*)
  fix *name*
  from *assms* **have** *rel-option* ($\lambda v\ t.\ t = value\text{-}to\text{-}sterm\ v$) (*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* ($++_f$) Γ) (*those* (*map2 vmatch ps vs*)))
   (*map-option* (*foldl* ($++_f$) 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* Γ_1 Γ_2)
     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 $\equiv$ ($\lambda(\Gamma_1, pat_1, rhs_1)$ (Γ_2, pat_2, rhs_2). $rhs_1 = rhs_2 \wedge pat_1 = pat_2 \wedge$ *env-eq* Γ_1 Γ_2)

lemma (**in** *constructors*) *find-match-eq*:
  assumes *list-all* (*linear* $\circ$ *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* $\Rightarrow$ *value* $\Rightarrow$ *bool* (‹- $\approx_e$ -›) **where**
constr: *list-all2 erelated ts us* $\Longrightarrow$ *Vconstr name ts* $\approx_e$ *Vconstr name us* |
abs: *fmrel-on-fset* (*ids* (*Sabs cs*)) *erelated* Γ_1 Γ_2 $\Longrightarrow$ *Vabs cs* Γ_1 $\approx_e$ *Vabs cs* Γ_2 |
rec-abs:
  pred-fmap (λ*cs*. *fmrel-on-fset* (*ids* (*Sabs cs*)) *erelated* Γ_1 Γ_2) *css* $\Longrightarrow$
    Vrecabs css name Γ_1 $\approx_e$ *Vrecabs css name* Γ_2

code-pred *erelated* **.**

global-interpretation *erelated*: *value-struct-rel erelated*
proof
  fix v_1 v_2
  assume $v_1 \approx_e v_2$
  thus *veq-structure* v_1 v_2
    by *induction* (*auto intro*: *list.rel-mono-strong*)
next
  fix *name name′* **and** *ts us* :: *value list*
  show *Vconstr name ts* $\approx_e$ *Vconstr name′ us* $\longleftrightarrow$ (*name* = *name′* $\wedge$ *list-all2 erelated ts us*)
    by (*auto intro*: *erelated.intros elim*: *erelated.cases*)
qed

lemma *erelated-refl*[*intro*]: $t \approx_e t$
proof (*induction t*)
  case *Vrecabs*
  thus *?case*
    apply (*auto intro*!: *erelated.intros fmpredI fmrel-on-fset-refl-strong*)
    apply (*auto intro*: *fmran′I*)

done
qed (*auto intro*!: *erelated.intros list.rel-refl-strong fmrel-on-fset-refl-strong fmran'I*)

export-code
value-to-sterm 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
checking *Scala*

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.Mat e cs*) ⟷ *is-cupcake-exp e* ∧ *list-all* (λ(*p*, *e*). *is-cupcake-pat p* ∧ *is-cupcake-exp e*) *cs* ∧ *cake-linear-clauses cs* |
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 es* -) ⟷ *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*
          Matching.fold2 (*λp v m. case m of*

Match env ⇒ *cupcake-pmatch cenv p v env*
| *m* ⇒ *m*) *Match-type-error ps vs* (*Match env*)
else
No-match
else
Match-type-error
| - => *Match-type-error*) |
cupcake-pmatch cenv - - - = *Match-type-error*

fun *cupcake-match-result* :: - ⇒ *v* ⇒(*pat*∗*exp*)*list* ⇒ *v* ⇒ (*exp* × *pat* × (*char list* × *v*) *list*, *v*)*result* **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′ ⇒ *Rval* (*e*, *p*, *env*′) |
No-match ⇒ *cupcake-match-result cenv v0 pes err-v* |
Match-type-error ⇒ *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* (λ(*p*, *e*). *cupcake-pmatch cenv p v0* [] = *No-match* ∧ *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* (λ(*p*, *e*). *cupcake-pmatch cenv p v0* [] = *No-match* ∧ *distinct* (*pat-bindings p* [])) *init*
assume *distinct* (*pat-bindings p* [])
assume *cupcake-pmatch cenv p v0* [] = *Match env*′

moreover have *pe* # *pes* = ((*p0*, *e0*) # *init*) @ (*p*, *e*) # *rest*
unfolding ‹*pes* = -› ‹*pe* = -› **by** *simp*

moreover have *list-all* (λ(*p*, *e*). *cupcake-pmatch cenv p v0* [] = *No-match* ∧ *distinct* (*pat-bindings p* [])) ((*p0*, *e0*) # *init*)
apply *auto*
subgoal by *fact*
subgoal using *True* **by** *simp*
subgoal using ‹*list-all* - -› **by** *simp*
done

moreover have *distinct* (*pat-bindings p* [])
by *fact*

ultimately show *thesis*
using *Cons* **by** *blast*
next
show *cupcake-match-result cenv v0 pes err-v* = *Rval* (*e*, *p*, *env'*)
using *Cons No-match True* **unfolding** ‹*pe* = -› **by** *auto*
qed
next
case *Match*
with *Cons* **show** *?thesis*
using *True* **unfolding** ‹*pe* = -› **by** *force*
next
case *Match-type-error*
with *Cons* **show** *?thesis*
using *True* **unfolding** ‹*pe* = -› **by** *force*
qed
next
case *False*
hence *False*
using *Cons* **unfolding** ‹*pe* = -› **by** *force*
thus *?thesis* **..**
qed
qed *simp*

lemma *cupcake-pmatch-eq*:
is-cupcake-pat pat ⟹ *pmatch-single envC s pat v0 env* = *cupcake-pmatch envC pat v0 env*
proof (*induct rule*: *pmatch-single.induct*)
case *4*
from *is-cupcake-pat.elims*(*2*)[*OF 4*(*2*)] **show** *?case*
proof *cases*
case *2*
then show *?thesis*
using *4*(*1*) **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 $\Longrightarrow$
match-result env s v pes err-v =
map-result (λ(*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 $\Longrightarrow$ *is-cupcake-value v0* $\Longrightarrow$ *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* (λ(-, -, *e*). *is-cupcake-exp e*) *fs*
shows *is-cupcake-ns* (*build-rec-env fs cl-env env*)
proof −
have *is-cupcake-ns* (*foldr* (λ(*f*, -) *env′*. *nsBind f* (*Recclosure cl-env fs0 f*) *env′*) *fs env*)
if *list-all* (λ(-, -, *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 assms* **by** *auto*
qed *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*))
shows *is-cupcake-all-env* (*sem-env.update-v f env*)

using *assms*
 by (*metis is-cupcake-all-env.simps*(*1*) *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* $\Longrightarrow$ *is-cupcake-ns ns2* $\Longrightarrow$ *is-cupcake-ns* (*nsAppend ns1 ns2*)
by (*auto elim*!: *is-cupcake-ns.elims*)

lemma *cupcake-alist-to-ns-preserve*: *list-all* (*is-cupcake-value* ∘ *snd*) *env* $\Longrightarrow$ *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′*› ‹*is-cupcake-value cl*› ‹*is-cupcake-value v0*› ‹*vs* = [*cl*, *v0*]› *recclosure*
    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*
  by *simp*+

qed

context begin

lemma *cup-pmatch-list-length-neq*:
  length vs $\neq$ *length ps* $\Longrightarrow$ *Matching.fold2*(λ*p v m. case m of*
    Match env $\Rightarrow$ *cupcake-pmatch cenv p v env*
    | *m* $\Rightarrow$ *m*) *Match-type-error ps vs m* = *Match-type-error*
  by (*induction ps vs arbitrary*:*m rule*:*List.list-induct2'*) *auto*

lemma *cup-pmatch-list-nomatch*:
  length vs = *length ps* $\Longrightarrow$ *Matching.fold2*(λ*p v m. case m of*
    Match env $\Rightarrow$ *cupcake-pmatch cenv p v env*
    | *m* $\Rightarrow$ *m*) *Match-type-error ps vs No-match* = *No-match*
  by (*induction ps vs rule*:*List.list-induct2'*) *auto*

lemma *cup-pmatch-list-typerr*:
  length vs = *length ps* $\Longrightarrow$ *Matching.fold2*(λ*p v m. case m of*
    Match env $\Rightarrow$ *cupcake-pmatch cenv p v env*
    | *m* $\Rightarrow$ *m*) *Match-type-error ps vs Match-type-error* = *Match-type-error*
  by (*induction ps vs rule*:*List.list-induct2'*) *auto*

private lemma *cupcake-pmatch-list-preserve*:
  assumes $\bigwedge$*p v env. p* $\in$ *set ps* $\wedge$ *v* $\in$ *set vs* $\longrightarrow$ *list-all* (*is-cupcake-value* $\circ$ *snd*)
env $\longrightarrow$ *if-match* (*list-all* (*is-cupcake-value* $\circ$ *snd*)) (*cupcake-pmatch cenv p v env*)
list-all (*is-cupcake-value* $\circ$ *snd*) *env*
  shows *if-match* (*list-all* (λ*a. is-cupcake-value* (*snd a*))) (*Matching.fold2*
      (λ*p v m. case m of*
        Match env $\Rightarrow$ *cupcake-pmatch cenv p v env*
      | *m* $\Rightarrow$ *m*)
      Match-type-error ps vs (*Match env*))
  using *assms* **proof** (*induction ps vs arbitrary*: *env rule*:*list-induct2'*)
  case (*4 p ps v vs*)
  show *?case*
  proof (*cases cupcake-pmatch cenv p v env*)
    case *No-match*
    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* $\circ$ *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 $\Longrightarrow$
is-cupcake-value v0 $\Longrightarrow$
list-all (*is-cupcake-value* $\circ$ *snd*) *env* $\Longrightarrow$
cupcake-c-ns envC $\Longrightarrow$
if-match (*list-all* (*is-cupcake-value* $\circ$ *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* $\Longrightarrow$ *is-cupcake-pat p* **for** *p*
using *2* **by** (*metis Ball-set is-cupcake-pat.simps*(*2*))
have *v*:*v* $\in$ *set vs* $\Longrightarrow$ *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 $\Longrightarrow$
is-cupcake-value v0 $\Longrightarrow$
list-all (*is-cupcake-value* $\circ$ *snd*) *env* $\Longrightarrow$
cupcake-c-ns envC $\Longrightarrow$
cupcake-pmatch envC pat v0 env = *Match env'* $\Longrightarrow$
list-all (*is-cupcake-value* $\circ$ *snd*) *env'*
by (*metis if-match.simps*(*1*) *cupcake-pmatch-preserve0*)+

end

lemma *cupcake-match-result-preserve*:
cupcake-c-ns envC $\Longrightarrow$
cupcake-clauses pes $\Longrightarrow$
is-cupcake-value v $\Longrightarrow$
if-rval (λ(*e*, *p*, *env'*). *is-cupcake-pat p* $\wedge$ *is-cupcake-exp e* $\wedge$ *list-all* (*is-cupcake-value* $\circ$ *snd*) *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) $\in$ *set pes*
by *simp*
thus *is-cupcake-exp e is-cupcake-pat p*
using *assms* **by** (*auto simp*: *list-all-iff*)

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

end

fun *list-all2-shortcircuit* **where**
list-all2-shortcircuit P (x # *xs*) (y # *ys*) $\longleftrightarrow$ (*case* y *of* *Rval* - $\Rightarrow$ P x y $\wedge$ *list-all2-shortcircuit* P *xs* *ys* | *Rerr* - $\Rightarrow$ P x y) |
list-all2-shortcircuit P [] [] $\longleftrightarrow$ *True* |
list-all2-shortcircuit P - - $\longleftrightarrow$ *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 $\bigwedge x$ *xs* y *ys*. P x (*Rval* y) $\Longrightarrow$ *list-all2-shortcircuit P xs ys* $\Longrightarrow$ R *xs ys* $\Longrightarrow$ R (x # *xs*) (*Rval* y # *ys*)
assumes $\bigwedge x$ *xs* y *ys*. P x (*Rerr* y) $\Longrightarrow$ R (x # *xs*) (*Rerr* y # *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 *list-all2-shortcircuit* $R \leq$ *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* $\Longrightarrow$ ($\bigwedge$*xs ys*. P *xs ys* $\Longrightarrow$ Q *xs ys*) $\Longrightarrow$ *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 (*map Rval ys*) $\longleftrightarrow$ *list-all2* (λx y. P x (*Rval* y)) *xs ys*
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)

proof
  assume *?lhs* **thus** *?rhs*
   by (*induction map Rval ys*::(′*b*, ′*c*) *result list arbitrary*: *ys rule*: *list-all2-shortcircuit-induct*) *auto*
next
  assume *?rhs* **thus** *?lhs*
    by (*induction rule*: *list-all2-induct*) *auto*
qed

inductive *cupcake-evaluate-single* :: *all-env* ⇒ *exp* ⇒ (*v*, *v*) *result* ⇒ *bool* **where**
con1:
  do-con-check (*c env*) *cn* (*length es*) ⟹
   list-all2-shortcircuit (*cupcake-evaluate-single env*) (*rev es*) *rs* ⟹
   sequence-result rs = *Rval vs* ⟹
   build-conv (*c env*) *cn* (*rev vs*) = *Some v0* ⟹
   cupcake-evaluate-single env (*Con cn es*) (*Rval v0*) |
con2:
  ¬ *do-con-check* (*c env*) *cn* (*List.length es*) ⟹
   cupcake-evaluate-single env (*Con cn es*) (*Rerr* (*Rabort Rtype-error*)) |
con3:
  do-con-check (*c env*) *cn* (*List.length es*) ⟹
   list-all2-shortcircuit (*cupcake-evaluate-single env*) (*rev es*) *rs* ⟹
   sequence-result rs = *Rerr err* ⟹
   cupcake-evaluate-single env (*Con cn es*) (*Rerr err*) |
var1:
  nsLookup (*sem-env.v env*) *n* = *Some v0* ⟹ *cupcake-evaluate-single env* (*Var n*) (*Rval v0*) |
var2:
  nsLookup (*sem-env.v env*) *n* = *None* ⟹ *cupcake-evaluate-single env* (*Var n*) (*Rerr* (*Rabort Rtype-error*)) |
fn:
  cupcake-evaluate-single env (*Fun n e*) (*Rval* (*Closure env n e*)) |
app1:
  list-all2-shortcircuit (*cupcake-evaluate-single env*) (*rev es*) *rs* ⟹
   sequence-result rs = *Rval vs* ⟹
   do-opapp (*rev vs*) = *Some* (*env*′, *e*) ⟹
   cupcake-evaluate-single env′ *e bv* ⟹
   cupcake-evaluate-single env (*App Opapp es*) *bv* |
app3:
  list-all2-shortcircuit (*cupcake-evaluate-single env*) (*rev es*) *rs* ⟹
   sequence-result rs = *Rval vs* ⟹
   do-opapp (*rev vs*) = *None* ⟹
   cupcake-evaluate-single env (*App Opapp es*) (*Rerr* (*Rabort Rtype-error*)) |
app6:
  list-all2-shortcircuit (*cupcake-evaluate-single env*) (*rev es*) *rs* ⟹
   sequence-result rs = *Rerr err* ⟹
   cupcake-evaluate-single env (*App op0 es*) (*Rerr err*) |
mat1:
  cupcake-evaluate-single env e (*Rval v0*) ⟹

cupcake-match-result (*c env*) *v0 pes Bindv* = *Rval* (*e′*, -, *env′*) $\Longrightarrow$
cupcake-evaluate-single (*env* (| *sem-env.v* := *nsAppend* (*alist-to-ns env′*) (*sem-env.v env*) |)) *e′ bv* $\Longrightarrow$
cupcake-evaluate-single env (*Mat e pes*) *bv* |
mat1error:
cupcake-evaluate-single env e (*Rval v0*) $\Longrightarrow$
cupcake-match-result (*c env*) *v0 pes Bindv* = *Rerr err* $\Longrightarrow$
cupcake-evaluate-single env (*Mat e pes*) (*Rerr err*) |
mat2:
cupcake-evaluate-single env e (*Rerr err*) $\Longrightarrow$
cupcake-evaluate-single env (*Mat e pes*) (*Rerr err*)

context *cakeml-static-env* **begin**

context begin

private lemma *cupcake-list-preserve0*:
list-all2-shortcircuit
(λ*e r*. *cupcake-evaluate-single env e r* $\wedge$ (*is-cupcake-all-env env* $\longrightarrow$ *is-cupcake-exp e* $\longrightarrow$ *if-rval is-cupcake-value r*)) *es rs* $\Longrightarrow$
is-cupcake-all-env env $\Longrightarrow$ *list-all is-cupcake-exp es* $\Longrightarrow$ *sequence-result rs* = *Rval vs* $\Longrightarrow$ *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 $\Longrightarrow$ *is-cupcake-all-env env* $\Longrightarrow$ *is-cupcake-exp e* $\Longrightarrow$ *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-value* (*rev vs*) **and** *c env* = *static-cenv*
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′ bv*)
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* ∘ *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*
(λ*e 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.cons1*) *fact+*
qed (*auto intro*:*evaluate-list.intros*)

private lemma *cupcake-list-correct-rerr*:
assumes *list-all2-shortcircuit*
(λ*e r*.
cupcake-evaluate-single env e r $\wedge$
(*is-cupcake-all-env env* $\longrightarrow$ *is-cupcake-exp e* $\longrightarrow$ ($\forall$ (*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*
(λ*e r*.
cupcake-evaluate-single env e r $\wedge$
(*is-cupcake-all-env env* $\longrightarrow$ *is-cupcake-exp e* $\longrightarrow$ ($\forall$ (*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-Unclocked-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 = *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 = *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' bv*)
  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* ∘ *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 (*rule cupcake-alist-to-ns-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
($\lambda s\ e\ r$. *evaluate env s e r* $\wedge$ (*is-cupcake-all-env env* $\longrightarrow$ *is-cupcake-exp e* $\longrightarrow$ *cupcake-evaluate-single env e* (*snd r*))) *s1 es res* $\Longrightarrow$
is-cupcake-all-env env $\Longrightarrow$ *list-all is-cupcake-exp es* $\Longrightarrow$ $\exists\, rs$. *list-all2-shortcircuit* (*cupcake-evaluate-single env*) *es rs* $\wedge$ *sequence-result rs* = (*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 $\Longrightarrow$ *is-cupcake-all-env env* $\Longrightarrow$ *is-cupcake-exp e* $\Longrightarrow$ *cupcake-evaluate-single env e* (*snd res*)

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 env′ 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 env′*
    using *app1 cupcake-opapp-preserve* **by** *fastforce*+
  hence *cupcake-evaluate-single env′ 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 Bindv* = *Rval* (*e′*, *uu*, *env′*)
using *cupcake-match-result-eq*[*OF pes*,**where** *env* = *env* **and** *s*= *s2* **and** *v* = *v1* **and** *err-v* = *Bindv*, *unfolded mat1*]
by (*cases cupcake-match-result* (*c env*) *v1 pes Bindv*) *auto*

ultimately have *list-all* (*is-cupcake-value* ∘ *snd*) *env′ is-cupcake-exp e′*
using *cupcake-match-result-preserve*[**where** *envC* = *c env* **and** *v* = *v1* **and** *pes* = *pes* **and** *err-v* = *Bindv*]
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 (*rule cupcake-alist-to-ns-preserve*)
apply *fact*
apply (*rule is-cupcake-all-envD*)
apply *fact*
done

ultimately have *cupcake-evaluate-single env e* (*Rval v1*)
and *cupcake-evaluate-single* (*update-v* (λ-. *nsAppend* (*alist-to-ns env′*) (*sem-env.v env*)) *env*) *e′* (*snd bv*)
using *mat1* **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* (*Rval 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*] **unfolding** *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 $(s', res) \Longrightarrow$ *is-cupcake-all-env env* $\Longrightarrow$ *is-cupcake-exp e* $\Longrightarrow$ *cupcake-evaluate-single env e res*
by (*fastforce dest*:*cupcake-single-complete0*)

lemma *cupcake-list-complete*:
evaluate-list (*evaluate env*) *s1 es res* $\Longrightarrow$
is-cupcake-all-env env $\Longrightarrow$ *list-all is-cupcake-exp es* $\Longrightarrow$ $\exists$ *rs. list-all2-shortcircuit* (*cupcake-evaluate-single env*) *es rs* $\wedge$ *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* (λ*s e res. Big-Step-Unclocked-Single.evaluate env s e res* $\wedge$ (*is-cupcake-all-env env* $\longrightarrow$ *is-cupcake-exp e* $\longrightarrow$ *s* = *fst res*)) *s es res*
list-all is-cupcake-exp es is-cupcake-all-env env
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** *rs*: *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 *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-complete cupcake-single-preserve*)+

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

ultimately have *list-all* (*is-cupcake-value* ∘ *snd*) *env' is-cupcake-exp e'*
using *cupcake-match-result-preserve*[**where** *envC* = *c env* **and** *v* = *v1* **and** *pes* = *pes* **and** *err-v* = *Bindv*]
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 (*rule cupcake-alist-to-ns-preserve*)
apply *fact*
apply (*rule is-cupcake-all-envD*)
apply *fact*
done
ultimately show *?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*) ⟹ *is-cupcake-all-env env* ⟹ *is-cupcake-exp e* ⟹ *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 $R :: 'a::term \Rightarrow 'a \Rightarrow bool$
  assumes *R-fun*: $R\ t\ t' \Longrightarrow R\ (app\ t\ u)\ (app\ t'\ u)$
  assumes *R-arg*: $R\ u\ u' \Longrightarrow R\ (app\ t\ u)\ (app\ t\ u')$
begin

lemma *rt-fun*:
  $R^{**}\ t\ t' \Longrightarrow R^{**}\ (app\ t\ u)\ (app\ t'\ u)$
by (*induct rule*: *rtranclp.induct*) (*auto intro*: *rtranclp.rtrancl-into-rtrancl R-fun*)

lemma *rt-arg*:
  $R^{**}\ u\ u' \Longrightarrow R^{**}\ (app\ t\ u)\ (app\ t\ u')$
by (*induct rule*: *rtranclp.induct*) (*auto intro*: *rtranclp.rtrancl-into-rtrancl R-arg*)

lemma *rt-comb*:
  $R^{**}\ t_1\ u_1 \Longrightarrow R^{**}\ t_2\ u_2 \Longrightarrow R^{**}\ (app\ t_1\ t_2)\ (app\ u_1\ u_2)$
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

end

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* ⟹ $r \vdash t \rightarrow u$ ⟹ *rs* ⊢ $t \longrightarrow u$ |
beta: *rs* ⊢ (Λ t \$ t') ⟶ t $[t']_\beta$ |
fun: *rs* ⊢ $t \longrightarrow t'$⟹ *rs* ⊢ t \$ u ⟶ t' \$ u |
arg: *rs* ⊢ $u \longrightarrow u'$⟹ *rs* ⊢ t \$ u ⟶ t \$ u'

global-interpretation *rewrite*: *rewriting rewrite rs* **for** *rs*
by *standard* (*auto intro*: *rewrite.intros 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 *arity*: *arity-compatibles rs*
assumes *fmap*: *is-fmap rs*
assumes *patterns*: *pattern-compatibles rs*
assumes *nonempty*: $rs \neq \{||\}$
assumes *not-shadows*: *fBall rs* (λ(*lhs*, -). $\neg$ *shadows-consts lhs*)
assumes *welldefined-rs*: *fBall rs* (λ(-, *rhs*). *welldefined rhs*)
begin

lemma *rewrite-wellformed*:
assumes $rs \vdash t \longrightarrow t'$ *wellformed t*
shows *wellformed* t'
using *assms*
proof (*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 wellformed.rewrite-step*)
apply (*rule step*(*2*)[*unfolded* ‹*r* = -›])
apply *fact+*
done
next
case (*beta u t*)
show *?case*
unfolding *wellformed-term-def*
apply (*rule replace-bound-wellformed*)
using *beta* **by** *auto*
qed *auto*

lemma *rewrite-rt-wellformed*: $rs \vdash t \longrightarrow* t' \Longrightarrow$ *wellformed t* $\Longrightarrow$ *wellformed* t'
by (*induction rule*: *rtranclp.induct*) (*auto intro*: *rewrite-wellformed simp del*: *wellformed-term-def*)

lemma *rewrite-closed*: $rs \vdash t \longrightarrow t' \Longrightarrow$ *closed t* $\Longrightarrow$ *closed* t'
proof (*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 *simp*
moreover have (*lhs*, *rhs*) $\vdash t \rightarrow u$
using *step* **unfolding** ‹*r* = -› **by** *simp*
show *?case*
apply (*rule rewrite-step-closed*)
apply *fact+*

done
next
case (*beta t t'*)
have *frees t* $[t']_\beta$ |⊆| *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* (‹-/ $\vdash_n$/ - ⟶/ -›
[*50*,*0*,*50*] *50*) **for** *rs* **where**

step: r |∈| $rs \Longrightarrow r \vdash t \to u \Longrightarrow rs \vdash_n t \longrightarrow u$ |
beta: $rs \vdash_n ((\Lambda_n\ x.\ t)\ \$_n\ t') \longrightarrow$ *subst* t (*fmap-of-list* $[(x,\ t')]$) |
fun: $rs \vdash_n t \longrightarrow t' \Longrightarrow rs \vdash_n t\ \$_n\ u \longrightarrow t'\ \$_n\ u$ |
arg: $rs \vdash_n u \longrightarrow u' \Longrightarrow rs \vdash_n t\ \$_n\ u \longrightarrow t\ \$_n\ u'$

global-interpretation *nrewrite*: *rewriting nrewrite rs* **for** *rs*
by *standard* (*auto intro*: *nrewrite.intros simp*: *app-nterm-def*)+

abbreviation *nrewrite-rt* :: *nrule fset* ⇒ *nterm* ⇒ *nterm* ⇒ *bool* (‹-/ $\vdash_n$/ - ⟶∗/ -› [*50*,*0*,*50*] *50*) **where**
nrewrite-rt rs ≡ (*nrewrite rs*)**

lemma (**in** *nrules*) *nrewrite-closed*:
  assumes $rs \vdash_n t \longrightarrow t'$ *closed* t
  shows *closed* t'
using *assms* **proof** *induction*
  case (*step r t u*)
  obtain *lhs rhs* **where** r = (*lhs*, *rhs*)
    by *force*
  with *step* **have** *nrule* (*lhs*, *rhs*)
    using *all-rules* **by** *blast*
  hence *frees rhs* |⊆| *frees lhs*
    by *simp*
  have (*lhs*, *rhs*) $\vdash t \to u$
    using *step* **unfolding** ‹r = -› **by** *simp*

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

corollary (**in** *nrules*) *nrewrite-rt-closed*:
  assumes $rs \vdash_n t \longrightarrow\!* \ t'$ *closed* t
  shows *closed* t'
using *assms*
by *induction* (*auto intro*: *nrewrite-closed*)

3.2.3 Translation from *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 i)
  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])
    apply (intro conjI)
    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 t_2) x_2)) |⊆| *fset-of-list* Γ |∪| *frees* t_2 |∪| T_2
and *fBall* T_2 ((<) x_2)
apply *atomize-elim*
apply (*rule app*(*2*))
using *app* **by** *auto*
moreover obtain $t_2{}'$ x' **where** *run-state* (*term-to-nterm* Γ t_2) x_2 = ($t_2{}'$, x')
by *fastforce*
ultimately have *all-frees* $t_2{}'$ |⊆| *fset-of-list* Γ |∪| *frees* t_2 |∪| 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* Γ t_1) x_1 = -›
apply *simp*
unfolding ‹*run-state* (*term-to-nterm* Γ t_2) x_2 = -›
apply *simp*
apply (*rule exI*[**where** x = T_1 |∪| T_2])
apply (*intro conjI*)
subgoal using ‹*all-frees* $t_1{}'$ |⊆| -› **by** *auto*
subgoal using ‹*all-frees* $t_2{}'$ |⊆| -› **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 *auto*
done
qed *auto*

lemma *term-to-nterm-all-vars*:
assumes *wellformed t fdisjnt* (*frees t*) *S*
shows *fdisjnt* (*all-frees* (*fresh-frun* (*term-to-nterm* [] *t*) (*T* |∪| *S*))) *S*
proof −
let *?*Γ = []
let *?x* = *fresh-fNext* (*T* |∪| *S*)
from *assms* **have** *wellformed′* (*length ?*Γ) *t*
by *simp*
then obtain *F*
where *all-frees* (*fst* (*run-state* (*term-to-nterm ?*Γ *t*) *?x*)) |⊆| *fset-of-list ?*Γ |∪| *frees t* |∪| *F*
and *fBall F* (λ*y*. *y* > *?x*)
by (*metis term-to-nterm-all-vars0*)

have *fdisjnt F* (*T* |∪| *S*) **if** *S* ≠ {||}

apply (*rule fdisjnt-ge-max*)
apply (*rule fBall-pred-weaken*[*OF* - ‹*fBall F* (λ*y*. *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'*) |∈| *rs*
and *rhs*: *rhs* = *fresh-frun* (*term-to-nterm* [] *rhs'*) (*frees lhs* |∪| *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* (*fst* (*strip-comb lhs*)) ¬ *is-const lhs* **by** *auto*

have *frees rhs* |⊆| *frees rhs'*
unfolding *rhs* **using** *rule*
by (*metis rule.simps term-to-nterm-vars*)
also have *frees rhs'* |⊆| *frees lhs*
using *rule* **by** *auto*
finally show *frees rhs* |⊆| *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) |∈| *compile* (lhs_2, rhs_2) |∈| *compile*
then obtain rhs_1' rhs_2' **where** (lhs_1, rhs_1') |∈| *rs* (lhs_2, rhs_2') |∈| *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* (*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* = -› *compile-def* **by** *auto*
with *assms* **have** *rule* (*pat*, *rhs*)
using *all-rules* **by** *blast*
hence *rhs* = *nterm-to-term* [] *rhs′*
unfolding ‹*rhs′* = -›
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* $\rightarrow$ *nterm-to-term* [] *u′*
apply (*subst nterm-to-term-eq-closed*)
apply *fact*

```
      apply simp
      apply (rule exI[where x = env])
      apply (rule conjI)
       apply fact
      unfolding ‹rhs = -›
      apply (subst nrelated-subst)
        apply fact
       apply fact
      unfolding fdisjnt-alt-def apply simp
      unfolding ‹u' = subst rhs' 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 rs ⊢ t ⟶ t' closed t wellformed t
  obtains u' where compile ⊢_n fst (run-state (term-to-nterm [] t) s) ⟶ u' u' ≈_α
fst (run-state (term-to-nterm [] t') s')
apply atomize-elim
using assms proof (induction t t' arbitrary: s s')
  case (step r t t')
  let ?t_n = fst (run-state (term-to-nterm [] t) s)
  let ?t_n' = fst (run-state (term-to-nterm [] t') s')
  from step have closed t closed ?t_n
    using term-to-nterm-vars0[where Γ = []]
    by force+
  from step have nterm-to-term' ?t_n = t
    by (auto intro!: term-to-nterm-nterm-to-term0)

  obtain pat rhs' where r = (pat, rhs')
    by fastforce
```

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* |∪| *all-consts*) (*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 ‹- = *rhs'*›[*symmetric*]
  apply (*subst nrelated-subst*)
    apply *fact*+
  using ‹- = *u'*›
  by *simp*+

have *compile* $\vdash_n$ $?t_n \longrightarrow u'$
  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'* -› **have** $u' \approx_\alpha$ $?t_n{}'$
 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* ($t\ [t']_\beta$) *closed* (Λ *t* \$ *t'*) *closed t'*
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 ∗∗:$t\ [t']_\beta$ = *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 nrewrite.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 *wellform*:*wellformed t wellformed u*
by *fastforce+*
from *fun* **obtain** *u'*
where *compile* $\vdash_n$ *fst* (*run-state* (*term-to-nterm* [] *t*) *s*) ⟶ *u'*
u' $\approx_\alpha$ *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 *wellform*:*wellformed t wellformed u*
by *fastforce*+
from *arg* **obtain** *t′*
where *compile* $\vdash_n$ *fst* (*run-state* (*term-to-nterm* [] *u*) (*snd* (*run-state* (*term-to-nterm* [] *t*) *s*))) $\longrightarrow$ *t′*
t′ $\approx_\alpha$ *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*
qed

lemma *compile-complete*:
assumes *rs* $\vdash$ *t* $\longrightarrow$ *t′ closed t wellformed t*
obtains *u′* **where** *compile* $\vdash_n$ *term-to-nterm′ t* $\longrightarrow$ *u′ u′* $\approx_\alpha$ *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($pats_1$, -) ($pats_2$, -). *length* $pats_1$ = *length* $pats_2$)

lemma *arity-compatible-length*:
assumes *arity-compatibles rs* (*pats*, *rhs*) |∈| *rs*

shows *length pats* = *arity rs*
proof −
have *fBall rs* (λ(*pats*$_1$, -). *fBall rs* (λ(*pats*$_2$, -). *length pats*$_1$ = *length pats*$_2$))
using *assms* **unfolding** *fpairwise-alt-def* **by** *blast*
hence *fBall rs* (λ*x*. *fBall rs* (λ*y*. (*length* ∘ *fst*) *x* = (*length* ∘ *fst*) *y*))
by *force*
hence (*length* ∘ *fst*) (*pats*, *rhs*) = *arity rs*
using *assms*(*2*) **unfolding** *arity-def fthe-elem′-eq* **by** (*rule singleton-fset-holds*)
thus *?thesis*
by *simp*
qed

locale *pre-crules* = *constants C-info fst* |'| *rs* **for** *C-info* **and** *rs* :: *crule-set*

locale *crules* = *pre-crules* +
assumes *fmap*: *is-fmap rs*
assumes *nonempty*: *rs* ≠ {||}
assumes *inner*:
fBall rs (λ(-, *crs*).
arity-compatibles crs ∧
is-fmap crs ∧
patterns-compatibles crs ∧
crs ≠ {||} ∧
fBall crs (λ(*pats*, *rhs*).
linears pats ∧
pats ≠ [] ∧
fdisjnt (*freess pats*) *all-consts* ∧
¬ *shadows-consts rhs* ∧
frees rhs |⊆| *freess pats* ∧
welldefined rhs))

lemma (**in** *pre-crules*) *crulesI*:
assumes ⋀*name crs*. (*name*, *crs*) |∈| *rs* ⟹ *arity-compatibles crs*
assumes ⋀*name crs*. (*name*, *crs*) |∈| *rs* ⟹ *is-fmap crs*
assumes ⋀*name crs*. (*name*, *crs*) |∈| *rs* ⟹ *patterns-compatibles crs*
assumes ⋀*name crs*. (*name*, *crs*) |∈| *rs* ⟹ *crs* ≠ {||}
assumes ⋀*name crs pats rhs*. (*name*, *crs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *crs* ⟹ *linears pats*
assumes ⋀*name crs pats rhs*. (*name*, *crs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *crs* ⟹ *pats* ≠ []
assumes ⋀*name crs pats rhs*. (*name*, *crs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *crs* ⟹ *fdisjnt* (*freess pats*) *all-consts*
assumes ⋀*name crs pats rhs*. (*name*, *crs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *crs* ⟹ ¬ *shadows-consts rhs*
assumes ⋀*name crs pats rhs*. (*name*, *crs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *crs* ⟹ *frees rhs* |⊆| *freess pats*
assumes ⋀*name crs pats rhs*. (*name*, *crs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *crs* ⟹ *welldefined rhs*
assumes *is-fmap rs rs* ≠ {||}

shows *crules C-info rs*
proof *unfold-locales*
  show *is-fmap rs*
    using *assms*(*11*) **.**
next
  show *rs* ≠ {||}
    using *assms*(*12*) **.**
next
  show *fBall rs* (λ(-, *crs*). *Rewriting-Nterm.arity-compatibles crs* ∧ *is-fmap crs* ∧
    patterns-compatibles crs ∧ *crs* ≠ {||} ∧
    fBall crs (λ(*pats*, *rhs*). *linears pats* ∧ *pats* ≠ [] ∧ *fdisjnt* (*freess pats*) *all-consts*
∧
      ¬ *shadows-consts rhs* ∧ *frees rhs* |⊆| *freess pats* ∧ *consts rhs* |⊆| *all-consts*))
    using *assms*(*1*−*10*)
    by (*intro prod-BallI conjI*) *metis+*
qed

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* $\neq$ {||}
unfolding *consts-of-def*
by (*meson femptyE fgroup-by-nonempty-inner*)

show *arity-compatibles crs patterns-compatibles crs*
proof *safe*
fix $pats_1$ rhs_1 $pats_2$ rhs_2
assume ($pats_1$, rhs_1) |∈| *crs* ($pats_2$, rhs_2) |∈| *crs*
with *crs* **obtain** lhs_1 lhs_2
where *rs*: (lhs_1, rhs_1) |∈| *rs* (lhs_2, rhs_2) |∈| *rs* **and**
split: *split-rule* (lhs_1, rhs_1) = (*name*, ($pats_1$, rhs_1))
split-rule (lhs_2, rhs_2) = (*name*, ($pats_2$, rhs_2))
unfolding *consts-of-def* **by** (*force simp*: *split-beta*)

hence *arity*: *arity-compatible* lhs_1 lhs_2
using *arity* **by** (*force dest*: *fpairwiseD*)

from *rs* **have** *const*: *is-const* (*fst* (*strip-comb* lhs_1)) *is-const* (*fst* (*strip-comb* lhs_2))
using *all-rules* **by** *force+*

have *name* = *const-name* (*fst* (*strip-comb* lhs_1)) *name* = *const-name* (*fst* (*strip-comb* lhs_2))
using *split* **by** (*auto simp*: *split-beta*)
with *const* **have** *fst* (*strip-comb* lhs_1) = *Const name fst* (*strip-comb* lhs_2) = *Const name*
apply (*fold const-term-def*)
subgoal by *simp*
subgoal by *fastforce*
done
hence *fst*: *fst* (*strip-comb* lhs_1) = *fst* (*strip-comb* lhs_2)
by *simp*

with *arity* **have** *length* (*snd* (*strip-comb* lhs_1)) = *length* (*snd* (*strip-comb* lhs_2))
unfolding *arity-compatible-def*
by (*simp add*: *split-beta*)

with *split* **show** *length* $pats_1$ = *length* $pats_2$
by (*auto simp*: *split-beta*)

have *pattern-compatible* lhs_1 lhs_2
using *rs patterns* **by** (*auto dest*: *fpairwiseD*)
moreover have lhs_1 = *name* \$\$ $pats_1$
using *split*(*1*) *const*(*1*) **by** (*auto simp*: *split-beta*)
moreover have lhs_2 = *name* \$\$ $pats_2$
using *split*(*2*) *const*(*2*) **by** (*auto simp*: *split-beta*)
ultimately have *pattern-compatible* (*name* \$\$ $pats_1$) (*name* \$\$ $pats_2$)
by *simp*

```
    thus patterns-compatible pats₁ pats₂
      using ‹length pats₁ = -› by (auto dest: pattern-compatible-combD)
  qed

show is-fmap crs
  proof
    fix pats rhs₁ rhs₂
    assume (pats, rhs₁) |∈| crs (pats, rhs₂) |∈| crs
    with crs obtain lhs₁ lhs₂
      where rs: (lhs₁, rhs₁) |∈| rs (lhs₂, rhs₂) |∈| rs and
            split: split-rule (lhs₁, rhs₁) = (name, (pats, rhs₁))
                   split-rule (lhs₂, rhs₂) = (name, (pats, rhs₂))
      unfolding consts-of-def by (force simp: split-beta)

    have lhs₁ = lhs₂
      proof (rule ccontr)
        assume lhs₁ ≠ lhs₂
        then consider (fst) fst (strip-comb lhs₁) ≠ fst (strip-comb lhs₂)
                    | (snd) snd (strip-comb lhs₁) ≠ snd (strip-comb lhs₂)
          by (metis list-strip-comb)
        thus False
          proof cases
            case fst
          moreover have is-const (fst (strip-comb lhs₁)) is-const (fst (strip-comb
lhs₂))
              using rs all-rules by force+
            ultimately show ?thesis
              using split const-name-simps by (fastforce simp: split-beta)
          next
            case snd
            with split show ?thesis
              by (auto simp: split-beta)
          qed
      qed

    with rs show rhs₁ = rhs₂
      using ‹is-fmap rs› by (auto dest: is-fmapD)
  qed

fix pats rhs
assume (pats, rhs) |∈| crs
then obtain lhs where (lhs, rhs) |∈| rs pats = snd (strip-comb lhs)
  using crs unfolding consts-of-def by (force simp: split-beta)
hence nrule (lhs, rhs)
  using all-rules by blast
hence linear lhs frees rhs |⊆| frees lhs
  by auto
thus linears pats
  unfolding ‹pats = -› by (intro linears-strip-comb)
```

have ¬ *is-const lhs is-const* (*fst* (*strip-comb lhs*))
  using ‹*nrule* -› **by** *auto*
thus *pats* ≠ []
  unfolding ‹*pats* = -› **using** ‹*linear lhs*›
  apply (*cases lhs*)
    apply (*fold app-term-def*)
  by (*auto split*: *prod.splits*)

from ‹*nrule* (*lhs*, *rhs*)› **have** *frees* (*fst* (*strip-comb lhs*)) = {||}
  by (*cases fst* (*strip-comb lhs*)) (*auto simp*: *is-const-def*)
hence *frees lhs* = *freess* (*snd* (*strip-comb lhs*))
  by (*subst frees-strip-comb*) *auto*
thus *frees rhs* |⊆| *freess pats*
  unfolding ‹*pats* = -› **using** ‹*frees rhs* |⊆| *frees lhs*› **by** *simp*

have ¬ *shadows-consts rhs*
  using ‹(*lhs*, *rhs*) |∈| *rs*› *not-shadows*
  by *force*
thus ¬ *pre-constants.shadows-consts C-info* (*fst* |‘| *consts-of rs*) *rhs*
  by (*simp add*: *consts-of-heads*)

have *fdisjnt all-consts* (*frees lhs*)
  using ‹(*lhs*, *rhs*) |∈| *rs*› *not-shadows*
  by (*force simp*: *shadows-consts-def*)
moreover have *freess pats* |⊆| *frees lhs*
  unfolding ‹*pats* = -› ‹*frees lhs* = -›
  by *simp*
ultimately have *fdisjnt* (*freess pats*) *all-consts*
  by (*metis fdisjnt-subset-right fdisjnt-swap*)

thus *fdisjnt* (*freess pats*) (*pre-constants.all-consts C-info* (*fst* |‘| *consts-of rs*))
  by (*simp add*: *consts-of-heads*)

show *pre-constants.welldefined C-info* (*fst* |‘| *consts-of rs*) *rhs*
  using *welldefined-rs* ‹(*lhs*, *rhs*) |∈| *rs*›
  by (*force simp*: *consts-of-heads*)
qed

sublocale *nrules* ⊆ *nrules-as-crules?*: *crules C-info consts-of rs*
by (*fact consts-rules*)

3.2.7 Computability

export-code
  translate-rule consts-of arity nterm-to-term
  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* = (*term list* × *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 irs (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 $\bigwedge$*name irs pats rhs.* (*name*, *irs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *irs* ⟹ ¬ *shadows-consts rhs*
assumes $\bigwedge$*name irs pats rhs.* (*name*, *irs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *irs* ⟹ *welldefined rhs*
assumes *is-fmap rs rs* ≠ {||}
shows *irules C-info rs*
proof *unfold-locales*
show *is-fmap rs*
using *assms*(*12*) **.**
next
show *rs* ≠ {||}
using *assms*(*13*) **.**
next
show *fBall rs* (λ(-, *irs*). *Rewriting-Nterm.arity-compatibles irs* ∧ *is-fmap irs* ∧
patterns-compatibles irs ∧ *irs* ≠ {||} ∧
fBall irs (λ(*pats*, *rhs*). *linears pats* ∧ *abs-ish pats rhs* ∧ *closed-except rhs* (*freess pats*) ∧
fdisjnt (*freess pats*) *all-consts* ∧ *pre-strong-term-class.wellformed rhs* ∧
¬ *shadows-consts rhs* ∧ *consts rhs* |⊆| *all-consts*))
using *assms*(*1*−*11*)
by (*intro prod-BallI conjI*) *metis+*
qed

lemmas *irulesI*[*intro*!] = *pre-irules.irulesI*[*unfolded pre-irules-def*]

Translation from *nterm* to *pterm*

fun *nterm-to-pterm* :: *nterm* ⇒ *pterm* **where**
nterm-to-pterm (*Nvar s*) = *Pvar s* |
nterm-to-pterm (*Nconst s*) = *Pconst s* |
nterm-to-pterm (t_1 $\$_n$ t_2) = *nterm-to-pterm* t_1 $\$_p$ *nterm-to-pterm* t_2 |
nterm-to-pterm (Λ_n *x*. *t*) = (Λ_p *x*. *nterm-to-pterm t*)

lemma *nterm-to-pterm-inj*: *nterm-to-pterm x* = *nterm-to-pterm y* ⟹ *x* = *y*
by (*induction y arbitrary*: *x*) (*auto elim*: *nterm-to-pterm.elims*)

lemma *nterm-to-pterm*:
assumes *no-abs t*
shows *nterm-to-pterm t* = *convert-term t*
using *assms*
apply *induction*
apply *auto*
by (*auto simp*: *free-nterm-def free-pterm-def const-nterm-def const-pterm-def app-nterm-def app-pterm-def*)

lemma *nterm-to-pterm-frees*[*simp*]: *frees* (*nterm-to-pterm t*) = *frees t*
by (*induct t*) *auto*

lemma *closed-nterm-to-pterm*[*intro*]: *closed-except* (*nterm-to-pterm t*) (*frees t*)

unfolding *closed-except-def* **by** *simp*

lemma (**in** *constants*) *shadows-nterm-to-pterm*[*simp*]: *shadows-consts* (*nterm-to-pterm t*) = *shadows-consts t*
by (*induct t*) (*auto simp*: *shadows-consts-def fdisjnt-alt-def*)

lemma *wellformed-nterm-to-pterm*[*intro*]: *wellformed* (*nterm-to-pterm t*)
by (*induct t*) *auto*

lemma *consts-nterm-to-pterm*[*simp*]: *consts* (*nterm-to-pterm t*) = *consts t*
by (*induct t*) *auto*

Translation from *crule-set* to *irule-set*

definition *translate-crules* :: *crules* ⇒ *irules* **where**
translate-crules = *fimage* (*map-prod id nterm-to-pterm*)

definition *compile* :: *crule-set* ⇒ *irule-set* **where**
compile = *fimage* (*map-prod id translate-crules*)

lemma *compile-heads*: *fst* |'| *compile rs* = *fst* |'| *rs*
unfolding *compile-def* **by** *simp*

lemma (**in** *crules*) *compile-rules*: *irules C-info* (*compile rs*)
proof
  have *is-fmap rs*
    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 (*metis* (*no-types*, *lifting*) *case-prodD*)
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 *linears pats pats* ≠ [] *frees rhs*′ |⊆| *freess pats* ¬ *shadows-consts rhs*′
using *fbspec*[*OF inner* ‹(*name*, *irs*′) |∈| *rs*›]
by *blast*+

show *linears pats* **by** *fact*
show *closed-except rhs* (*freess pats*)
unfolding ‹*rhs* = *nterm-to-pterm rhs*′›
using ‹*frees rhs*′ |⊆| *freess pats*›
by (*metis dual-order.trans closed-nterm-to-pterm closed-except-def*)

show *wellformed rhs*
unfolding ‹*rhs* = *nterm-to-pterm rhs*′› **by** *auto*

have *fdisjnt* (*freess pats*) *all-consts*
using ‹(*pats*, *rhs*′) |∈| *irs*′› ‹(*name*, *irs*′) |∈| *rs*› *inner*
by *auto*
thus *fdisjnt* (*freess pats*) (*pre-constants.all-consts C-info* (*fst* |‘| *compile rs*))
unfolding *compile-def* **by** *simp*

have ¬ *shadows-consts rhs*
unfolding ‹*rhs* = -› **using** ‹¬ *shadows-consts* -› **by** *simp*
thus ¬ *pre-constants.shadows-consts C-info* (*fst* |‘| *compile rs*) *rhs*

unfolding *compile-heads* **.**

show *abs-ish pats rhs*
using ‹*pats* ≠ []› **unfolding** *abs-ish-def* **by** *simp*

have *welldefined rhs'*
using *fbspec*[*OF inner* ‹(*name*, *irs'*) |∈| *rs*›, *simplified*]
using ‹(*pats*, *rhs'*) |∈| *irs'*›
by *blast*

thus *pre-constants.welldefined C-info* (*fst* |'| *compile rs*) *rhs*
unfolding *compile-def* ‹*rhs* = -›
by *simp*
qed

sublocale *crules* ⊆ *crules-as-irules*: *irules C-info compile rs*
by (*fact compile-rules*)

Transformation of *irule-set*

definition *transform-irules* :: *irules* ⇒ *irules* **where**
transform-irules rs = (
if arity rs = 0 then rs
else map-prod id Pabs |'| *fgroup-by* (λ(*pats*, *rhs*). (*butlast pats*, (*last pats*, *rhs*)))
rs)

lemma *arity-compatibles-transform-irules*:
assumes *arity-compatibles rs*
shows *arity-compatibles* (*transform-irules rs*)
proof (*cases arity rs* = *0*)
case *True*
thus *?thesis*
unfolding *transform-irules-def* **using** *assms* **by** *simp*
next
case *False*
let *?rs'* = *transform-irules rs*
let *?f* = λ(*pats*, *rhs*). (*butlast pats*, (*last pats*, *rhs*))
let *?grp* = *fgroup-by ?f rs*
have *rs'*: *?rs'* = *map-prod id Pabs* |'| *?grp*
using *False* **unfolding** *transform-irules-def* **by** *simp*
show *?thesis*
proof *safe*
fix $pats_1$ rhs_1 $pats_2$ rhs_2
assume ($pats_1$, rhs_1) |∈| *?rs'* ($pats_2$, rhs_2) |∈| *?rs'*
then obtain rhs_1' rhs_2' **where** ($pats_1$, rhs_1') |∈| *?grp* ($pats_2$, rhs_2') |∈| *?grp*
unfolding *rs'* **by** *auto*
then obtain $pats_1'$ $pats_2'$ *x y* — dummies
where *fst* (*?f* ($pats_1'$, *x*)) = $pats_1$ ($pats_1'$, *x*) |∈| *rs*
and *fst* (*?f* ($pats_2'$, *y*)) = $pats_2$ ($pats_2'$, *y*) |∈| *rs*

by (*fastforce simp*: *split-beta elim*: *fgroup-byE2*)
hence $pats_1 = butlast\ pats_1{}'$ $pats_2 = butlast\ pats_2{}'$ $length\ pats_1{}' = length\ pats_2{}'$
using *assms* **by** (*force dest*: *fpairwiseD*)+
thus $length\ pats_1 = length\ pats_2$
by *auto*
qed
qed

lemma *arity-transform-irules*:
assumes *arity-compatibles rs rs* $\neq$ {||}
shows *arity* (*transform-irules rs*) = (*if arity rs* = *0 then 0 else arity rs* − *1*)
proof (*cases arity rs* = *0*)
case *True*
thus *?thesis*
unfolding *transform-irules-def* **by** *simp*
next
case *False*
let *?f* = λ(*pats*, *rhs*). (*butlast pats*, (*last pats*, *rhs*))
let *?grp* = *fgroup-by ?f rs*
let *?rs'* = *map-prod id Pabs* |‘| *?grp*

have *arity ?rs'* = *arity rs* − *1*
proof (*rule arityI*)
show *fBall ?rs'* (λ(*pats*, -). *length pats* = *arity rs* − *1*)
proof (*rule prod-fBallI*)
fix *pats rhs*
assume (*pats*, *rhs*) |∈| *?rs'*
then obtain *cs* **where** (*pats*, *cs*) |∈| *?grp rhs* = *Pabs cs*
by *force*
then obtain *pats' x* — dummy
where *pats* = *butlast pats'* (*pats'*, *x*) |∈| *rs*
by (*fastforce simp*: *split-beta elim*: *fgroup-byE2*)
hence *length pats'* = *arity rs*
using *assms* **by** (*metis arity-compatible-length*)
thus *length pats* = *arity rs* − *1*
unfolding ‹*pats* = *butlast pats'*› **using** *False* **by** *simp*
qed
next
show *?rs'* $\neq$ {||}
using *assms* **by** (*simp add*: *fgroup-by-nonempty*)
qed

with *False* **show** *?thesis*
unfolding *transform-irules-def* **by** *simp*
qed

definition *transform-irule-set* :: *irule-set* ⇒ *irule-set* **where**
transform-irule-set = *fimage* (*map-prod id transform-irules*)

lemma *transform-irule-set-heads*: *fst* |‘| *transform-irule-set rs* = *fst* |‘| *rs*
unfolding *transform-irule-set-def* **by** *simp*

lemma (**in** *irules*) *rules-transform*: *irules C-info* (*transform-irule-set rs*)
proof
  have *is-fmap rs*
    using *fmap* **by** *simp*
  thus *is-fmap* (*transform-irule-set rs*)
   unfolding *transform-irule-set-def map-prod-def id-apply* **by** (*rule is-fmap-image*)

  show *transform-irule-set rs* ≠ {||}
    using *nonempty* **unfolding** *transform-irule-set-def* **by** *auto*

  show *constants C-info* (*fst* |‘| *transform-irule-set rs*)
    proof
      show *fdisjnt* (*fst* |‘| *transform-irule-set rs*) *C*
        using *disjnt* **unfolding** *transform-irule-set-def*
        by *force*
    next
      show *distinct all-constructors*
        by (*fact distinct-ctr*)
    qed

  fix *name irs*
  assume *irs*: (*name*, *irs*) |∈| *transform-irule-set rs*
  then obtain *irs′* **where** (*name*, *irs′*) |∈| *rs irs* = *transform-irules irs′*
    unfolding *transform-irule-set-def* **by** *force*
  hence *arity-compatibles irs′*
    using *inner*
    by (*metis* (*no-types*, *lifting*) *case-prodD*)
  thus *arity-compatibles irs*
   unfolding ‹*irs* = *transform-irules irs′*› **by** (*rule arity-compatibles-transform-irules*)

  have *irs′* ≠ {||}
    using ‹(*name*, *irs′*) |∈| *rs*› *inner* **by** *blast*
  thus *irs* ≠ {||}
    unfolding ‹*irs* = *transform-irules irs′*› *transform-irules-def*
    by (*simp add*: *fgroup-by-nonempty*)

  let *?f* = λ(*pats*, *rhs*). (*butlast pats*, (*last pats*, *rhs*))
  let *?grp* = *fgroup-by ?f irs′*

  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 |`| ?grp
      unfolding ‹irs = transform-irules irs'› transform-irules-def by simp

    show ?thesis
      proof safe
        fix pats₁ rhs₁ pats₂ rhs₂
        assume (pats₁, rhs₁) |∈| irs (pats₂, rhs₂) |∈| irs
        with irs' obtain cs₁ cs₂ where (pats₁, cs₁) |∈| ?grp (pats₂, cs₂) |∈| ?grp
          by force
        then obtain pats₁' pats₂' and x y — dummies
          where (pats₁', x) |∈| irs' (pats₂', y) |∈| irs'
            and pats₁ = butlast pats₁' pats₂ = butlast pats₂'
          unfolding irs'
          by (fastforce elim: fgroup-byE2)
        hence patterns-compatible pats₁' pats₂'
          using ‹patterns-compatibles irs'› by (auto dest: fpairwiseD)
        thus patterns-compatible pats₁ pats₂
          unfolding ‹pats₁ = -› ‹pats₂ = -›
          by auto
      qed
  qed

have is-fmap irs'
  using ‹(name, irs') |∈| rs› inner by blast
show is-fmap irs
  proof (cases arity irs' = 0)
    case True
    thus ?thesis
      unfolding ‹irs = transform-irules irs'› transform-irules-def
      using ‹is-fmap irs'› by simp
  next
    case False
    hence irs': irs = map-prod id Pabs |`| ?grp
      unfolding ‹irs = transform-irules irs'› transform-irules-def by simp

    show ?thesis
      proof
        fix pats rhs₁ rhs₂
        assume (pats, rhs₁) |∈| irs (pats, rhs₂) |∈| irs
        with irs' obtain cs₁ cs₂
          where (pats, cs₁) |∈| ?grp rhs₁ = Pabs cs₁
            and (pats, cs₂) |∈| ?grp rhs₂ = Pabs cs₂
          by force
        moreover have is-fmap ?grp
          by auto
```

ultimately show $rhs_1 = rhs_2$
by (*auto dest*: *is-fmapD*)
qed
qed

fix *pats rhs*
assume (*pats*, *rhs*) |∈| *irs*

show *linears pats*
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 *auto*
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*
using ‹(*pats*, *rhs*) |∈| *irs*› **by** *force*
then obtain *pats′ x* — dummy
where *fst* (*?f* (*pats′*, *x*)) = *pats* (*pats′*, *x*) |∈| *irs′*
by (*fastforce simp*: *split-beta elim*: *fgroup-byE2*)
hence *pats* = *butlast pats′*
by *simp*
moreover have *linears pats′*
using ‹(*pats′*, *x*) |∈| *irs′*› ‹(*name*, *irs′*) |∈| *rs*› *inner*
by *auto*
ultimately show *?thesis*
by *auto*
qed

have *fdisjnt* (*freess pats*) *all-consts*
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 *auto*
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*
using ‹(*pats*, *rhs*) |∈| *irs*› **by** *force*
then obtain *pats′ x* — dummy
where *fst* (*?f* (*pats′*, *x*)) = *pats* (*pats′*, *x*) |∈| *irs′*
by (*fastforce simp*: *split-beta elim*: *fgroup-byE2*)

```
      hence pats = butlast pats'
        by simp
      moreover have fdisjnt (freess pats') all-consts
        using ‹(pats', x) |∈| irs'› ‹(name, irs') |∈| rs› inner by auto
      ultimately show ?thesis
        by (metis subsetI in-set-butlastD freess-subset fdisjnt-subset-left)
    qed

  thus fdisjnt (freess pats) (pre-constants.all-consts C-info (fst |`| transform-irule-set
rs))
    unfolding transform-irule-set-def by simp

  show closed-except rhs (freess pats)
    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 auto
    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› 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 auto

          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 |∪| frees pat = freess pats'
            unfolding freess-def by auto

          thus closed-except t (freess pats |∪| frees 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 *auto*
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* t_1 t_2
assume (*pat*, t_1) |∈| *cs* (*pat*, t_2) |∈| *cs*
then obtain $pats_1{}'$ $pats_2{}'$
where ($pats_1{}'$, t_1) |∈| *irs′* *?f* ($pats_1{}'$, t_1) = (*pats*, (*pat*, t_1))
and ($pats_2{}'$, t_2) |∈| *irs′* *?f* ($pats_2{}'$, t_2) = (*pats*, (*pat*, t_2))
using ‹(*pats*, *cs*) |∈| *?grp*› **by** *force*
moreover hence $pats_1{}' \neq []$ $pats_2{}' \neq []$
using ‹*arity-compatibles irs′*› *False*
unfolding *prod.case*
by (*metis list.size*(*3*) *arity-compatible-length*)+
ultimately have $pats_1{}'$ = *pats* @ [*pat*] $pats_2{}'$ = *pats* @ [*pat*]
unfolding *split-beta fst-conv snd-conv*
by (*metis prod.inject snoc-eq-iff-butlast*)+
with ‹*is-fmap irs′*› **show** $t_1 = t_2$
using ‹($pats_1{}'$, t_1) |∈| *irs′*› ‹($pats_2{}'$, t_2) |∈| *irs′*›
by (*blast dest*: *is-fmapD*)
qed
next
show *pattern-compatibles cs*
proof *safe*
fix pat_1 rhs_1 pat_2 rhs_2
assume (pat_1, rhs_1) |∈| *cs* (pat_2, rhs_2) |∈| *cs*
then obtain $pats_1{}'$ $pats_2{}'$
where ($pats_1{}'$, rhs_1) |∈| *irs′* *?f* ($pats_1{}'$, rhs_1) = (*pats*, (pat_1, rhs_1))
and ($pats_2{}'$, rhs_2) |∈| *irs′* *?f* ($pats_2{}'$, rhs_2) = (*pats*, (pat_2, rhs_2))

using ‹(*pats*, *cs*) |∈| *?grp*›
by *force*
moreover hence $pats_1{}' \neq []$ $pats_2{}' \neq []$
using ‹*arity-compatibles irs'*› *False*
unfolding *prod.case*
by (*metis list.size*(*3*) *arity-compatible-length*)+
ultimately have $pats_1{}' = pats$ @ $[pat_1]$ $pats_2{}' = pats$ @ $[pat_2]$
unfolding *split-beta fst-conv snd-conv*
by (*metis prod.inject snoc-eq-iff-butlast*)+
moreover have *patterns-compatible* $pats_1{}'$ $pats_2{}'$
using ‹($pats_1{}'$, rhs_1) |∈| *irs'*› ‹($pats_2{}'$, rhs_2) |∈| *irs'*› ‹*patterns-compatibles irs'*›
by (*auto dest*: *fpairwiseD*)
ultimately show *pattern-compatible* pat_1 pat_2
by (*auto elim*: *rev-accum-rel-snoc-eqE*)
qed
next
fix *pat t*
assume (*pat*, *t*) |∈| *cs*
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*

moreover have *linears pats'*
using ‹(*pats'*, *t*) |∈| *irs'*› ‹(*name*, *irs'*) |∈| *rs*› *inner* **by** *auto*
ultimately show *linear pat*
by (*metis linears-linear*)

show *wellformed t*
using ‹(*pats'*, *t*) |∈| *irs'*› ‹(*name*, *irs'*) |∈| *rs*› *inner* **by** *auto*
qed
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 *auto*
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 auto
            next
              assume shadows-consts pat

              have fdisjnt (freess pats') all-consts
                using ‹(name, irs') |∈| rs› ‹(pats', t) |∈| irs'› inner by auto
              have fdisjnt (frees pat) all-consts
                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 auto
  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 ‹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 = -›
      apply simp
      apply (rule ffUnion-least)
      unfolding ball-simps
      apply rule
      apply (rename-tac x, case-tac x, hypsubst-thin)
      apply simp
      subgoal premises prems for pat t
        proof −
          from prems obtain pats' where (pats', t) |∈| irs'
            using ‹(pats, cs) |∈| ?grp› by auto
          hence welldefined t
            using fbspec[OF inner ‹(name, irs') |∈| rs›, simplified]
            by blast
          thus ?thesis
            unfolding transform-irule-set-def
            by simp
        qed
```

    done
  qed
 thus *pre-constants.welldefined C-info* (*fst* |'| *transform-irule-set rs*) *rhs*
  unfolding *transform-irule-set-heads* **.**
qed

Matching and rewriting

definition *irewrite-step* :: *name* ⇒ *term list* ⇒ *pterm* ⇒ *pterm* ⇒ *pterm option*
where
irewrite-step name pats rhs t = *map-option* (*subst rhs*) (*match* (*name* $$ *pats*) *t*)

abbreviation *irewrite-step'* :: *name* ⇒ *term list* ⇒ *pterm* ⇒ *pterm* ⇒ *pterm* ⇒ *bool* (‹-, -, - $\vdash_i$/ - →/ -› [*50*,*0*,*50*] *50*) **where**
name, *pats*, *rhs* $\vdash_i$ *t* → *u* ≡ *irewrite-step name pats rhs t* = *Some u*

lemma *irewrite-stepI*:
 assumes *match* (*name* $$ *pats*) *t* = *Some env subst rhs env* = *u*
 shows *name*, *pats*, *rhs* $\vdash_i$ *t* → *u*
using *assms* **unfolding** *irewrite-step-def* **by** *simp*

inductive *irewrite* :: *irule-set* ⇒ *pterm* ⇒ *pterm* ⇒ *bool* (‹-/ $\vdash_i$/ - ⟶/ -› [*50*,*0*,*50*] *50*) **for** *irs* **where**
step: ⟦ (*name*, *rs*) |∈| *irs*; (*pats*, *rhs*) |∈| *rs*; *name*, *pats*, *rhs* $\vdash_i$ *t* → *t'* ⟧ ⟹ *irs* $\vdash_i$ *t* ⟶ *t'* |
beta: ⟦ *c* |∈| *cs*; *c* ⊢ *t* → *t'* ⟧ ⟹ *irs* $\vdash_i$ *Pabs cs* $\$_p$ *t* ⟶ *t'* |
fun: *irs* $\vdash_i$ *t* ⟶ *t'* ⟹ *irs* $\vdash_i$ *t* $\$_p$ *u* ⟶ *t'* $\$_p$ *u* |
arg: *irs* $\vdash_i$ *u* ⟶ *u'* ⟹ *irs* $\vdash_i$ *t* $\$_p$ *u* ⟶ *t* $\$_p$ *u'*

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

abbreviation *irewrite-rt* :: *irule-set* ⇒ *pterm* ⇒ *pterm* ⇒ *bool* (‹-/ $\vdash_i$/ - ⟶*/ -› [*50*,*0*,*50*] *50*) **where**
irewrite-rt rs ≡ (*irewrite rs*)**

lemma (**in** *irules*) *irewrite-closed*:
 assumes *rs* $\vdash_i$ *t* ⟶ *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** *fastforce*
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*
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 $rs \vdash_i t \longrightarrow\!* u$ *closed t*
shows *closed u*
using *assms* **by** *induction* (*auto intro*: *irewrite-closed*)

Correctness of translation

abbreviation *irelated* :: *nterm* ⇒ *pterm* ⇒ *bool* (‹- $\approx_i$ -› [*0*,*50*] *50*) **where**
$n \approx_i p \equiv$ *nterm-to-pterm* $n = p$

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

lemma *irelated-vars*: $t \approx_i u \Longrightarrow$ *frees t = frees u*
by *auto*

lemma *irelated-no-abs*:
assumes $t \approx_i u$
shows *no-abs* $t \longleftrightarrow$ *no-abs* u
using *assms*
apply (*induction arbitrary*: *t*)
apply (*auto elim*!: *nterm-to-pterm.elims*)
apply (*fold const-nterm-def const-pterm-def free-nterm-def free-pterm-def app-pterm-def app-nterm-def*)
by *auto*

lemma *irelated-subst*:
assumes $t \approx_i u$ *irelated.P-env nenv penv*
shows *subst t nenv* $\approx_i$ *subst u penv*
using *assms* **proof** (*induction arbitrary*: *nenv penv u rule*: *nterm-to-pterm.induct*)
case (*1 s*)
then show *?case*
by (*auto elim*!: *fmrel-cases*[**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-drop*)
qed *auto*

lemma *related-irewrite-step*:
assumes *name*, *pats*, *nterm-to-pterm rhs* $\vdash_i u \rightarrow u'$ $t \approx_i u$
obtains t' **where** *unsplit-rule* (*name*, *pats*, *rhs*) $\vdash t \rightarrow t'$ $t' \approx_i u'$
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*) $\vdash t \rightarrow$ *subst rhs nenv*
using ‹*match ?x t* = -› **by** *auto*
next
show *subst rhs nenv* $\approx_i u'$
unfolding ‹u' = -› **using** ‹*irelated.P-env nenv env*›
by (*auto intro*: *irelated-subst*)

qed
qed

theorem (**in** *nrules*) *compile-correct*:
assumes $\mathit{compile}\ (\mathit{consts\text{-}of}\ \mathit{rs}) \vdash_i u \longrightarrow u'$ $t \approx_i u$ $\mathit{closed}\ t$
obtains t' **where** $\mathit{rs} \vdash_n t \longrightarrow t'$ $t' \approx_i u'$
using *assms*(*1*−*3*) **proof** (*induction arbitrary*: *t thesis rule*: *irewrite.induct*)
case (*step name irs pats rhs u u′*)
then obtain *crs* **where** $\mathit{irs} = \mathit{translate\text{-}crules}\ \mathit{crs}$ $(\mathit{name}, \mathit{crs})\ |{\in}|\ \mathit{consts\text{-}of}\ \mathit{rs}$
unfolding *compile-def* **by** *force*
moreover with *step* **obtain** *rhs′* **where** $\mathit{rhs} = \mathit{nterm\text{-}to\text{-}pterm}\ \mathit{rhs}'$ $(\mathit{pats}, \mathit{rhs}')\ |{\in}|\ \mathit{crs}$
unfolding *translate-crules-def* **by** *force*
ultimately obtain *rule* **where** $\mathit{split\text{-}rule}\ \mathit{rule} = (\mathit{name}, (\mathit{pats}, \mathit{rhs}'))$ $\mathit{rule}\ |{\in}|\ \mathit{rs}$
unfolding *consts-of-def* **by** *blast*
hence *nrule rule*
using *all-rules* **by** *blast*

obtain t' **where** $\mathit{unsplit\text{-}rule}\ (\mathit{name}, \mathit{pats}, \mathit{rhs}') \vdash t \rightarrow t'$ $t' \approx_i u'$
using ‹$\mathit{name}, \mathit{pats}, \mathit{rhs} \vdash_i u \rightarrow u'$› ‹$t \approx_i u$› **unfolding** ‹$\mathit{rhs} = \mathit{nterm\text{-}to\text{-}pterm}\ \mathit{rhs}'$›
by (*elim related-irewrite-step*)
hence $\mathit{rule} \vdash t \rightarrow t'$
using ‹*nrule rule*› ‹$\mathit{split\text{-}rule}\ \mathit{rule} = (\mathit{name}, (\mathit{pats}, \mathit{rhs}'))$›
by (*metis unsplit-split*)

show *?case*
proof (*rule step.prems*)
show $\mathit{rs} \vdash_n t \longrightarrow t'$
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 *pat rhs* **where** $c = (\mathit{pat}, \mathit{rhs})$ $(\mathit{pat}, \mathit{rhs})\ |{\in}|\ \mathit{cs}$
by (*cases c*) *auto*
obtain *v w* **where** $t = v\ \$_n\ w$ $v \approx_i \mathit{Pabs}\ \mathit{cs}$ $w \approx_i u$
using ‹$t \approx_i \mathit{Pabs}\ \mathit{cs}\ \$_p\ u$› **by** (*auto elim*: *nterm-to-pterm.elims*)
obtain *x nrhs irhs* **where** $v = (\Lambda_n\ x.\ \mathit{nrhs})$ $\mathit{cs} = \{|\ (\mathit{Free}\ x, \mathit{irhs})\ |\}$ $\mathit{nrhs} \approx_i \mathit{irhs}$
using ‹$v \approx_i \mathit{Pabs}\ \mathit{cs}$› **by** (*auto elim*: *nterm-to-pterm.elims*)
hence $t = (\Lambda_n\ x.\ \mathit{nrhs})\ \$_n\ w$ $\Lambda_n\ x.\ \mathit{nrhs} \approx_i \Lambda_p\ x.\ \mathit{irhs}$
unfolding ‹$t = v\ \$_n\ w$› **using** ‹$v \approx_i \mathit{Pabs}\ \mathit{cs}$› **by** *auto*

have $\mathit{pat} = \mathit{Free}\ x$ $\mathit{rhs} = \mathit{irhs}$

```
      using ‹cs = {| (Free x, irhs) |}› ‹(pat, rhs) |∈| cs› by auto
    hence (Free x, irhs) ⊢ u → u'
      using beta ‹c = -› by simp
    hence u' = subst irhs (fmap-of-list [(x, u)])
      by simp

    show ?case
      proof (rule beta.prems)
        show rs ⊢n t ⟶ subst nrhs (fmap-of-list [(x, w)])
          unfolding ‹t = (Λn x. nrhs) $n w›
          by (rule nrewrite.beta)
      next
        show subst nrhs (fmap-of-list [(x, w)]) ≈i u'
          unfolding ‹u' = subst irhs -›
          apply (rule irelated-subst)
           apply fact
          apply simp
          apply rule
           apply rule
          apply fact
          done
      qed
  next
    case (fun v v' u)
    obtain w x where t = w $n x w ≈i v x ≈i u
      using ‹t ≈i v $p u› by (auto elim: nterm-to-pterm.elims)
    with fun obtain w' where rs ⊢n w ⟶ w' w' ≈i v'
      unfolding closed-except-def by auto

    show ?case
      proof (rule fun.prems)
        show rs ⊢n t ⟶ w' $n x
          unfolding ‹t = w $n x›
          by (rule nrewrite.fun) fact
      next
        show w' $n x ≈i v' $p u
          by auto fact+
      qed
  next
    case (arg u u' v)
    obtain w x where t = w $n x w ≈i v x ≈i u
      using ‹ t ≈i v $p u› by (auto elim: nterm-to-pterm.elims)
    with arg obtain x' where rs ⊢n x ⟶ x' x' ≈i u'
      unfolding closed-except-def by auto

    show ?case
      proof (rule arg.prems)
        show rs ⊢n t ⟶ w $n x'
          unfolding ‹t = w $n x›
```

        by (*rule nrewrite.arg*) *fact*
    next
      show $w\ \$_n\ x' \approx_i v\ \$_p\ u'$
        by *auto fact+*
    qed
qed

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

  obtain t' **where** $rs \vdash_n t \longrightarrow\!* t'$ $t' \approx_i u'$
    using *step* **by** *blast*

  obtain t'' **where** $rs \vdash_n t' \longrightarrow\!* t''$ $t'' \approx_i u''$
    proof (*rule compile-correct*)
      show *compile* (*consts-of rs*) $\vdash_i u' \longrightarrow u''$ $t' \approx_i u'$
        by *fact+*
    next
      show *closed* t'
        using ‹$rs \vdash_n t \longrightarrow\!* t'$› ‹*closed* t›
        by (*rule nrewrite-rt-closed*)
    qed *blast*

  show *?case*
    proof (*rule step.prems*)
      show $rs \vdash_n t \longrightarrow\!* t''$
        using ‹$rs \vdash_n t \longrightarrow\!* t'$› ‹$rs \vdash_n t' \longrightarrow\!* t''$› **by** *auto*
    qed *fact*
qed *blast*

Completeness of translation

lemma (**in** *nrules*) *compile-complete*:
  assumes $rs \vdash_n t \longrightarrow t'$ *closed* t
  shows *compile* (*consts-of rs*) $\vdash_i$ *nterm-to-pterm* $t \longrightarrow$ *nterm-to-pterm* t'
  using *assms* **proof** *induction*
  case (*step r t* t')
  then obtain *pat* rhs' **where** $r = (pat,\ rhs')$
    by *force*
  then have $(pat,\ rhs')$ $|\in|$ rs $(pat,\ rhs') \vdash t \rightarrow t'$
    using *step* **by** *blast+*
  then have *nrule* $(pat,\ rhs')$
    using *all-rules* **by** *blast*
  then obtain *name pats* **where** $(name,\ (pats,\ rhs')) =$ *split-rule* r $pat = name$
$$ *pats*
    unfolding *split-rule-def* ‹$r = -$›

apply *atomize-elim*
by (*auto simp*: *split-beta*)

obtain *crs* **where** (*name*, *crs*) |∈| *consts-of rs* (*pats*, *rhs'*) |∈| *crs*
using *step* ‹- = *split-rule r*› ‹*r* = -›
by (*metis consts-of-def fgroup-by-complete fst-conv snd-conv*)
then obtain *irs* **where** *irs* = *translate-crules crs*
by *blast*
then have (*name*, *irs*) |∈| *compile* (*consts-of rs*)
unfolding *compile-def*
using ‹(*name*, -) |∈| -›
by (*metis fimageI id-def map-prod-simp*)
obtain *rhs* **where** *rhs* = *nterm-to-pterm rhs'* (*pats*, *rhs*) |∈| *irs*
using ‹*irs* = -› ‹- |∈| *crs*›
unfolding *translate-crules-def*
by (*metis fimageI id-def map-prod-simp*)

from *step* **obtain** *env'* **where** *match pat t* = *Some env'* t' = *subst rhs' env'*
unfolding ‹*r* = -› **using** *rewrite-step.simps*
by *force*
then obtain *env* **where** *match pat* (*nterm-to-pterm t*) = *Some env irelated.P-env env' env*
by (*metis irelated.match-rel option-rel-Some1*)
then have *subst rhs env* = *nterm-to-pterm t'*
unfolding ‹t' = -›
apply −
apply (*rule sym*)
apply (*rule irelated-subst*)
unfolding ‹*rhs* = -›
by *auto*

have *name*, *pats*, *rhs* $\vdash_i$ *nterm-to-pterm t* → *nterm-to-pterm t'*
apply (*rule irewrite-stepI*)
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* (λ*env*. (*last pats*, *subst rhs env*)) (*matchs* (*butlast pats*) *args*))

context *irules* **begin**

inductive *prelated* :: *pterm* ⇒ *pterm* ⇒ *bool* (‹- $\approx_p$ -› [*0*,*50*] *50*) **where**
const: *Pconst x* $\approx_p$ *Pconst x* |
var: *Pvar x* $\approx_p$ *Pvar x* |
app: $t_1 \approx_p u_1 \Longrightarrow t_2 \approx_p u_2 \Longrightarrow t_1$ \$$_p$ $t_2 \approx_p u_1$ \$$_p$ u_2 |
pat: *rel-fset* (*rel-prod* (=) *prelated*) cs_1 $cs_2 \Longrightarrow$ *Pabs* $cs_1 \approx_p$ *Pabs* cs_2 |
defer:
(*name*, *rsi*) |∈| *rs* ⟹ *0* < *arity rsi* ⟹
rel-fset (*rel-prod* (=) *prelated*) (*irules-deferred-matches args rsi*) *cs* ⟹
list-all closed args ⟹
name \$\$ *args* $\approx_p$ *Pabs cs*

inductive-cases *prelated-absE*[*consumes 1*, *case-names pat defer*]: *t* $\approx_p$ *Pabs cs*

lemma *prelated-refl*[*intro*!]: *t* $\approx_p$ *t*
proof (*induction t*)

case *Pabs*
thus *?case*
by (*auto simp*: *snds.simps intro*!: *prelated.pat rel-fset-refl-strong rel-prod.intros*)
qed (*auto intro*: *prelated.intros*)

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

lemma *prelated-pvars*:
assumes $t \approx_p u$
shows *frees* t = *frees* u
using *assms* **proof** (*induction rule*: *prelated.induct*)
case (*pat* cs_1 cs_2)
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 *pat t*
assume (*pat*, *t*) |∈| *cs*
with *defer* **obtain** t'
where (*pat*, t') |∈| *irules-deferred-matches args rsi frees t* = *frees* t'
by (*auto elim*: *rel-fsetE2*)
then obtain *pats rhs env*
where *pat* = *last pats* (*pats*, *rhs*) |∈| *rsi*
and *matchs* (*butlast pats*) *args* = *Some env* t' = *subst rhs env*
by *auto*

have *closed-except rhs* (*freess pats*) *linears pats*
using ‹(*pats*, *rhs*) |∈| *rsi*› ‹(*name*, *rsi*) |∈| *rs*› *inner* **by** *auto*

have *arity-compatibles rsi*
using *defer inner*
by (*metis* (*no-types*, *lifting*) *case-prodD*)
have *length pats* > *0*
by (*subst arity-compatible-length*) *fact+*
hence *pats* = *butlast pats* @ [*last pats*]
by *simp*

note ‹*frees t* = *frees* t'›
also have *frees* t' = *frees rhs* − *fmdom env*
unfolding ‹t' = -›

apply (*rule subst-frees*)
apply (*rule closed.matchs*)
apply *fact*+
done
also have ... = *frees rhs* − *freess* (*butlast pats*)
using ‹*matchs* - - = -› **by** (*metis matchs-dom*)
also have ... |⊆| *freess pats* − *freess* (*butlast pats*)
using ‹*closed-except* - -›
by (*auto simp*: *closed-except-def*)
also have ... = *frees* (*last pats*) |−| *freess* (*butlast pats*)
by (*subst* ‹*pats* = -›) (*simp add*: *funion-fminus*)
also have ... = *frees* (*last pats*)
proof (*rule fminus-triv*)
have *fdisjnt* (*freess* (*butlast pats*)) (*freess* [*last pats*])
using ‹*linears pats*› ‹*pats* = -›
by (*metis linears-appendD*)
thus *frees* (*last pats*) |∩| *freess* (*butlast pats*) = {||}
by (*fastforce simp*: *fdisjnt-alt-def*)
qed
also have ... = *frees pat* **unfolding** ‹*pat* = -› **..**
finally have *frees t* |⊆| *frees pat* **.**
}
hence *closed* (*Pabs cs*)
unfolding *closed-except-simps*
by (*auto simp*: *closed-except-def*)
moreover have *closed* (*name* $$ *args*)
unfolding *closed-list-comb* **by** *fact*
ultimately show *?case*
unfolding *closed-except-def* **by** *simp*
qed *auto*

corollary *prelated-closed*: $t \approx_p u \Longrightarrow$ *closed* $t \longleftrightarrow$ *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: $t \approx_p u \Longrightarrow$ *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 $[\![$*match ?x ?u* = *Some ?env*; *?t* $\approx_p$ *?u*; $\bigwedge$*env'*. $[\![$*match ?x ?t* = *Some env'*; *prelated.P-env env' ?env*$]\!] \Longrightarrow$ *?thesis*$]\!] \Longrightarrow$ *?thesis*.

```
lemma prelated-subst:
  assumes t1 ≈p t2 prelated.P-env env1 env2
  shows subst t1 env1 ≈p subst t2 env2
using assms proof (induction arbitrary: env1 env2 rule: prelated.induct)
  case (var x)
  thus ?case
    proof (cases rule: fmrel-cases[where x = x])
      case none
      thus ?thesis
        by (auto intro: prelated.var)
    next
      case (some t u)
      thus ?thesis
        by simp
    qed
next
  case (pat cs1 cs2)
  let ?drop = λenv. λ(pat::term). fmdrop-fset (frees pat) env
  from pat have prelated.P-env (?drop env1 pat) (?drop env2 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 $$ args ≈p Pabs cs
    apply (rule prelated.defer)
       apply fact+
     apply (rule fset.rel-mono-strong)
      apply fact
     apply force
    apply fact
    done
  moreover have closed (name $$ args)
    unfolding closed-list-comb by fact
  ultimately have closed (Pabs cs)
    by (metis prelated-closed)

  let ?drop = λenv. λpat. fmdrop-fset (frees pat) env
  let ?f = λenv. (λ(pat, rhs). (pat, subst rhs (?drop env pat)))

  have name $$ args ≈p Pabs (?f env2 |`| cs)
    proof (rule prelated.defer)
```

show (*name*, *rsi*) |∈| *rs 0* < *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) |∈| *cs*
assume $pat_1 = pat_2$ $rhs_1 \approx_p rhs_2$
have $rhs_1 \approx_p$ *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* (λt. *subst t* env_1) *args* = *args*
apply (*rule map-idI*)
apply (*rule subst-closed-id*)
using *defer* **by** (*simp add*: *list-all-iff*)

ultimately show *?case*
by (*simp add*: *subst-list-comb*)
qed (*auto intro*: *prelated.intros*)

lemma *prelated-step*:
assumes *name*, *pats*, *rhs* $\vdash_i u \rightarrow u'$ $t \approx_p u$
obtains t' **where** *name*, *pats*, *rhs* $\vdash_i t \rightarrow t'$ $t' \approx_p u'$
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* $env' \approx_p$ *subst rhs env*
using *assms* **by** (*auto intro*: *prelated-subst*)

show *thesis*
proof
show *name*, *pats*, *rhs* $\vdash_i t \rightarrow$ *subst rhs* env'
unfolding *irewrite-step-def* **using** ‹*match ?lhs t* = *Some* env'›
by *simp*
next
show *subst rhs* $env' \approx_p u'$
unfolding ‹u' = *subst rhs env*›

by *fact*
qed
qed

lemma *prelated-beta*: — same problem as *prelated.related-match*
assumes $(pat,\ rhs_2) \vdash t_2 \rightarrow u_2$ $rhs_1 \approx_p rhs_2$ $t_1 \approx_p t_2$
obtains u_1 **where** $(pat,\ rhs_1) \vdash t_1 \rightarrow u_1$ $u_1 \approx_p u_2$
proof −
from *assms* **obtain** env_2 **where** $match\ pat\ t_2 = Some\ env_2$ $u_2 = subst\ rhs_2\ env_2$
by *auto*
with *assms* **obtain** env_1 **where** $match\ pat\ t_1 = Some\ env_1$ $prelated.P\text{-}env\ env_1$ env_2
by (*auto elim*: *prelated.related-match*)
with *assms* **have** $subst\ rhs_1\ env_1 \approx_p subst\ rhs_2\ env_2$
by (*auto intro*: *prelated-subst*)

show *thesis*
proof
show $(pat,\ rhs_1) \vdash t_1 \rightarrow subst\ rhs_1\ env_1$
using ‹$match\ pat\ t_1 = -$› **by** *simp*
next
show $subst\ rhs_1\ env_1 \approx_p u_2$
unfolding ‹$u_2 = -$› **by** *fact*
qed
qed

theorem *transform-correct*:
assumes $transform\text{-}irule\text{-}set\ rs \vdash_i u \longrightarrow u'$ $t \approx_p u$ $closed\ t$
obtains t' **where** $rs \vdash_i t \longrightarrow{*}\ t'$ — zero or one step **and** $t' \approx_p u'$
using *assms*(*1*−*3*) **proof** (*induction arbitrary*: *t thesis rule*: *irewrite.induct*)
case $(beta\ c\ cs_2\ u_2\ x_2')$
obtain $v\ u_1$ **where** $t = v\ \$_p\ u_1$ $v \approx_p Pabs\ cs_2$ $u_1 \approx_p u_2$
using ‹$t \approx_p Pabs\ cs_2\ \$_p\ u_2$› **by** *cases*
with *beta* **have** $closed\ u_1$
by (*simp add*: *closed-except-def*)

obtain $pat\ rhs_2$ **where** $c = (pat,\ rhs_2)$ **by** (*cases c*) *auto*

from ‹$v \approx_p Pabs\ cs_2$› **show** *?case*
proof (*cases rule*: *prelated-absE*)
case $(pat\ cs_1)$
with *beta* ‹$c = -$› **obtain** rhs_1 **where** $(pat,\ rhs_1)\ |{\in}|\ cs_1$ $rhs_1 \approx_p rhs_2$
by (*auto elim*: *rel-fsetE2*)
with *beta* **obtain** x_1' **where** $(pat,\ rhs_1) \vdash u_1 \rightarrow x_1'$ $x_1' \approx_p x_2'$
using ‹$u_1 \approx_p u_2$› *assms* ‹$c = -$›
by (*auto elim*: *prelated-beta simp del*: *rewrite-step.simps*)

show *?thesis*

proof (*rule beta.prems*)
show $rs \vdash_i t \longrightarrow\!* x_1{}'$
unfolding ‹t = -› ‹v = -›
by (*intro r-into-rtranclp irewrite.beta*) *fact+*
next
show $x_1{}' \approx_p x_2{}'$
by *fact*
qed
next
case (*defer name rsi args*)
with *beta* ‹c = -› **obtain** $rhs_1{}'$ **where** (pat, $rhs_1{}'$) |∈| *irules-deferred-matches args rsi* $rhs_1{}' \approx_p rhs_2$
by (*auto elim*: *rel-fsetE2*)
then obtain env_a rhs_1 *pats*
where *matchs* (*butlast pats*) *args* = *Some* env_a *pat* = *last pats* $rhs_1{}'$ = *subst* rhs_1 env_a
and (*pats*, rhs_1) |∈| *rsi*
by *auto*
hence *linears pats*
using ‹(*name*, *rsi*) |∈| *rs*› *inner* **unfolding** *irules-def* **by** *auto*

have *arity-compatibles rsi*
using *defer inner*
by (*metis* (*no-types*, *lifting*) *case-prodD*)
have *length pats* > *0*
by (*subst arity-compatible-length*) *fact+*
hence *pats* = *butlast pats* @ [*pat*]
unfolding ‹*pat* = -› **by** *simp*

from *beta* ‹c = -› **obtain** env_b **where** *match pat* u_2 = *Some* env_b $x_2{}'$ = *subst* rhs_2 env_b
by *fastforce*
with ‹$u_1 \approx_p u_2$› **obtain** $env_b{}'$ **where** *match pat* u_1 = *Some* $env_b{}'$ *prelated.P-env* $env_b{}'$ env_b
by (*metis prelated.related-match*)

have *closed-env* env_a
by (*rule closed.matchs*) *fact+*
have *closed-env* $env_b{}'$
apply (*rule closed.matchs*[**where** *pats* = [*pat*] **and** *ts* = [u_1]])
apply *simp*
apply *fact*
apply *simp*
apply *fact*
done

have *fmdom* env_a = *freess* (*butlast pats*)
by (*rule matchs-dom*) *fact*
moreover have *fmdom* $env_b{}'$ = *frees pat*

```
        by (rule match-dom) fact
      moreover have fdisjnt (freess (butlast pats)) (frees pat)
        using ‹pats = -› ‹linears pats›
        by (metis freess-single linears-appendD(3))
      ultimately have fdisjnt (fmdom env_a) (fmdom env_b')
        by simp

      show ?thesis
        proof (rule beta.prems)
          have rs ⊢_i name $$ args $_p u_1 ⟶ subst rhs_1' env_b'
            proof (rule irewrite.step)
              show (name, rsi) |∈| rs (pats, rhs_1) |∈| rsi
                by fact+
            next
              show name, pats, rhs_1 ⊢_i name $$ args $_p u_1 → subst rhs_1' env_b'
                apply (rule irewrite-stepI)
                 apply (fold app-pterm-def)
                 apply (subst list-comb-snoc)
                 apply (subst matchs-match-list-comb)
                 apply (subst ‹pats = -›)
                 apply (rule matchs-appI)
                  apply fact
                 apply simp
                 apply fact
                unfolding ‹rhs_1' = -›
                apply (rule subst-indep')
                 apply fact+
                done
            qed
          thus rs ⊢_i t ⟶* subst rhs_1' env_b'
            unfolding ‹t = -› ‹v = -›
            by (rule r-into-rtranclp)
        next
          show subst rhs_1' env_b' ≈_p x_2'
            unfolding ‹x_2' = -›
            by (rule prelated-subst) fact+
        qed
    qed
next
  case (step name rs_2 pats rhs u u')
  then obtain rs_1 where rs_2 = transform-irules rs_1 (name, rs_1) |∈| rs
    unfolding transform-irule-set-def by force
  hence arity-compatibles rs_1
    using inner
    by (metis (no-types, lifting) case-prodD)

  show ?case
    proof (cases arity rs_1 = 0)
      case True
```

hence $rs_2 = rs_1$
  unfolding ‹rs_2 = -› *transform-irules-def* **by** *simp*
with *step* **have** $(pats, rhs)$ |∈| rs_1
  by *simp*
from *step* **obtain** t' **where** $name, pats, rhs \vdash_i t \rightarrow t'$ $t' \approx_p u'$
  using *assms*
  by (*auto elim*: *prelated-step*)

show *?thesis*
  proof (*rule step.prems*)
    show $rs \vdash_i t \longrightarrow* t'$
      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* rs_1

  case *False*
  hence rs_2 = *map-prod id Pabs* |‘| *?grp*
    unfolding ‹rs_2 = -› *transform-irules-def* **by** *simp*
  with *step* **obtain** *cs* **where** *rhs* = *Pabs cs* (*pats*, *cs*) |∈| *?grp*
    by *force*

  from *step* **obtain** env_2 **where** *match* (*name* $$ *pats*) *u* = *Some* env_2 u' = *subst rhs* env_2
    unfolding *irewrite-step-def* **by** *auto*
  then obtain $args_2$ **where** *u* = *name* $$ $args_2$ *matchs pats* $args_2$ = *Some* env_2
    by (*auto elim*: *match-list-combE*)
  with *step* **obtain** $args_1$ **where** *t* = *name* $$ $args_1$ *list-all2 prelated* $args_1$ $args_2$
    by (*auto elim*: *prelated.list-combE*)

  then obtain env_1 **where** *matchs pats* $args_1$ = *Some* env_1 *prelated.P-env* env_1 env_2
    using ‹*matchs pats* $args_2$ = -› **by** (*metis prelated.related-matchs*)
  hence *fmdom* env_1 = *freess pats*
    by (*auto simp*: *matchs-dom*)

  obtain cs' **where** u' = *Pabs* cs'
    unfolding ‹u' = -› ‹*rhs* = -› **by** *auto*

  hence cs' = (λ(*pat*, *rhs*). (*pat*, *subst rhs* (*fmdrop-fset* (*frees pat*) env_2))) |‘| *cs*
    using ‹u' = *subst rhs* env_2› **unfolding** ‹*rhs* = -›
    by *simp*

  show *?thesis*
    proof (*rule step.prems*)
      show $rs \vdash_i t \longrightarrow* t$

by (*rule rtranclp.rtrancl-refl*)
next
show $t \approx_p u'$
unfolding ‹u' = *Pabs* cs'› ‹t = -›
proof (*intro prelated.defer rel-fsetI*; *safe?*)
show (*name*, rs_1) |∈| *rs*
by *fact*
next
show *0* < *arity* rs_1
using *False* **by** *simp*
next
show *list-all closed* $args_1$
using ‹*closed t*› **unfolding** ‹t = -› *closed-list-comb* **.**
next
fix *pat* rhs'
assume (*pat*, rhs') |∈| *irules-deferred-matches* $args_1$ rs_1
then obtain $pats'$ *rhs env*
where ($pats'$, *rhs*) |∈| rs_1
and *matchs* (*butlast* $pats'$) $args_1$ = *Some env pat* = *last* $pats'$ rhs' = *subst rhs env*
by *auto*
with *False* **have** $pats' \neq []$
using ‹*arity-compatibles* rs_1›
by (*metis list.size*(*3*) *arity-compatible-length*)
hence *butlast* $pats'$ @ [*last* $pats'$] = $pats'$
by *simp*

from ‹(*pats*, *cs*) |∈| *?grp*› **obtain** $pats_e$ rhs_e
where ($pats_e$, rhs_e) |∈| rs_1 *pats* = *butlast* $pats_e$
by (*auto elim*: *fgroup-byE2*)

have *patterns-compatible* (*butlast* $pats'$) *pats*
unfolding ‹*pats* = -›
apply (*rule rev-accum-rel-butlast*)
using ‹($pats'$, *rhs*) |∈| rs_1› ‹($pats_e$, rhs_e) |∈| rs_1› ‹(*name*, rs_1) |∈| *rs*› *inner*
by (*blast dest*: *fpairwiseD*)

interpret $irules'$: *irules C-info transform-irule-set rs* **by** (*rule rules-transform*)

have *butlast* $pats'$ = *pats env* = env_1
apply (*rule matchs-compatible-eq*)
subgoal by *fact*
subgoal
apply (*rule linears-butlastI*)
using ‹($pats'$, *rhs*) |∈| rs_1› ‹(*name*, rs_1) |∈| *rs*› *inner* **by** *auto*
subgoal
using ‹(*pats*, -) |∈| rs_2› ‹(*name*, rs_2) |∈| *transform-irule-set rs*›

```
    using irules'.inner by auto
    apply fact+
  subgoal
    apply (rule matchs-compatible-eq)
        apply fact
      apply (rule linears-butlastI)
    using ‹(pats', rhs) |∈| rs₁› ‹(name, rs₁) |∈| rs› inner
    apply auto []
    using ‹(pats, -) |∈| rs₂› ‹(name, rs₂) |∈| transform-irule-set rs›
    using irules'.inner apply auto[]
    by fact+
  done

let ?rhs-subst = λenv. subst rhs (fmdrop-fset (frees pat) env)

have fmdom env₂ = freess pats
  using ‹match (- $$ -) - = Some env₂›
  by (simp add: match-dom)

show fBex cs' (rel-prod (=) prelated (pat, rhs'))
  unfolding ‹rhs' = -›
  proof (rule fBexI, rule rel-prod.intros)
    have fdisjnt (freess (butlast pats')) (frees (last pats'))
      apply (subst freess-single[symmetric])
      apply (rule linears-appendD)
      apply (subst ‹butlast pats' @ [last pats'] = pats'›)
      using ‹(pats', rhs) |∈| rs₁› ‹(name, rs₁) |∈| rs› inner
      by auto

    show subst rhs env ≈p ?rhs-subst env₂
      apply (rule prelated-subst)
       apply (rule prelated-refl)
      unfolding fmfilter-alt-defs
      apply (subst fmfilter-true)
      subgoal premises prems for x y
        using fmdomI[OF prems]
        unfolding ‹pat = -› ‹fmdom env₂ = -›
        apply (subst (asm) ‹butlast pats' = pats›[symmetric])
        using ‹fdisjnt (freess (butlast pats')) (frees (last pats'))›
        by (auto simp: fdisjnt-alt-def)
      subgoal
        unfolding ‹env = -›
        by fact
      done
  next
    have (pat, rhs) |∈| cs
      unfolding ‹pat = -›
      apply (rule fgroup-byD[where a = (x, y) for x y])
        apply fact
```

```
          apply simp
          apply (intro conjI)
            apply fact
           apply (rule refl)+
          apply fact
          done
        thus (pat, ?rhs-subst env₂) |∈| cs'
          unfolding ‹cs' = -› by force
      qed simp
    next
      fix pat rhs'
      assume (pat, rhs') |∈| cs'
      then obtain rhs
        where (pat, rhs) |∈| cs
          and rhs' = subst rhs (fmdrop-fset (frees pat) env₂ )
        unfolding ‹cs' = -› by auto
      with ‹(pats, cs) |∈| ?grp› obtain pats'
        where (pats', rhs) |∈| rs₁ pats = butlast pats' pat = last pats'
        by auto
      with False have length pats' ≠ 0
        using ‹arity-compatibles -› by (metis arity-compatible-length)
      hence pats' = pats @ [pat]
        unfolding ‹pats = -› ‹pat = -› by auto
      moreover have linears pats'
        using ‹(pats', rhs) |∈| rs₁› ‹(name, rs₁) |∈| -› inner by auto
      ultimately have fdisjnt (fmdom env₁) (frees pat)
        unfolding ‹fmdom env₁ = -›
        by (auto dest: linears-appendD)

      let ?rhs-subst = λenv. subst rhs (fmdrop-fset (frees pat) env)

      show fBex (irules-deferred-matches args₁ rs₁) (λe. rel-prod (=) prelated
e (pat, rhs'))
        unfolding ‹rhs' = -›
        proof (rule fBexI, rule rel-prod.intros)
          show ?rhs-subst env₁ ≈p ?rhs-subst env₂
            using ‹prelated.P-env env₁ env₂› inner
            by (auto intro: prelated-subst)
        next
          have matchs (butlast pats') args₁ = Some env₁
            using ‹matchs pats args₁ = -› ‹pats = -› by simp
          moreover have subst rhs env₁ = ?rhs-subst env₁
            apply (rule arg-cong[where f = subst rhs])
            unfolding fmfilter-alt-defs
            apply (rule fmfilter-true[symmetric])
            using ‹fdisjnt (fmdom env₁) -›
            by (auto simp: fdisjnt-alt-def intro: fmdomI)
          ultimately show (pat, ?rhs-subst env₁) |∈| irules-deferred-matches
args₁ rs₁
```

using ‹$(pats', rhs)$ |∈| rs_1› ‹$pat = last\ pats'$›
by *auto*
qed *simp*
qed
qed
qed
next
case $(fun\ v\ v'\ u)$
obtain $w\ x$ **where** $t = w\ \$_p\ x\ w \approx_p v\ x \approx_p u\ closed\ w$
using ‹$t \approx_p v\ \$_p\ u$› ‹$closed\ t$› **by** *cases* (*auto simp*: *closed-except-def*)
with *fun* **obtain** w' **where** $rs \vdash_i w \longrightarrow* w'\ w' \approx_p v'$
by *blast*

show *?case*
proof (*rule fun.prems*)
show $rs \vdash_i t \longrightarrow* w'\ \$_p\ x$
unfolding ‹$t = $ -›
by (*intro irewrite.rt-comb*[*unfolded app-pterm-def*] *rtranclp.rtrancl-refl*) *fact*
next
show $w'\ \$_p\ x \approx_p v'\ \$_p\ u$
by (*rule prelated.app*) *fact+*
qed
next
case $(arg\ u\ u'\ v)$
obtain $w\ x$ **where** $t = w\ \$_p\ x\ w \approx_p v\ x \approx_p u\ closed\ x$
using ‹$t \approx_p v\ \$_p\ u$› ‹$closed\ t$› **by** *cases* (*auto simp*: *closed-except-def*)
with *arg* **obtain** x' **where** $rs \vdash_i x \longrightarrow* x'\ x' \approx_p u'$
by *blast*

show *?case*
proof (*rule arg.prems*)
show $rs \vdash_i t \longrightarrow* w\ \$_p\ x'$
unfolding ‹$t = w\ \$_p\ x$›
by (*intro irewrite.rt-comb*[*unfolded app-pterm-def*] *rtranclp.rtrancl-refl*) *fact*
next
show $w\ \$_p\ x' \approx_p v\ \$_p\ u'$
by (*rule prelated.app*) *fact+*
qed
qed

end

Completeness of transformation

lemma (**in** *irules*) *transform-completeness*:
assumes $rs \vdash_i t \longrightarrow t'\ closed\ t$
shows $transform\text{-}irule\text{-}set\ rs \vdash_i t \longrightarrow* t'$
using *assms* **proof** *induction*
case $(step\ name\ irs'\ pats'\ rhs'\ t\ t')$

then obtain *irs* **where** $irs = transform\text{-}irules\ irs'$ $(name, irs)$ |∈| *transform-irule-set rs*
  unfolding *transform-irule-set-def*
  by (*metis fimageI id-apply map-prod-simp*)
show *?case*
proof (*cases arity irs′ = 0*)
  case *True*
  hence $irs = irs'$
    unfolding ‹*irs* = -›
    unfolding *transform-irules-def*
    by *simp*
  with *step* **have** $(pats', rhs')$ |∈| *irs name*, $pats', rhs' \vdash_i t \to t'$
    by *blast+*
  have *transform-irule-set rs* $\vdash_i t \longrightarrow* t'$
    apply (*rule r-into-rtranclp*)
    apply *rule*
    by *fact+*

  show *?thesis* **by** *fact*
next
  let $?f = \lambda(pats, rhs). (butlast\ pats, last\ pats, rhs)$
  let *?grp = fgroup-by ?f irs′*
  note *closed-except-def* [*simp add*]
  case *False*
  then have *irs = map-prod id Pabs* |'| *?grp*
    unfolding ‹*irs* = -›
    unfolding *transform-irules-def*
    by *simp*
  with *False* **have** *irs = transform-irules irs′*
    unfolding *transform-irules-def*
    by *simp*
  obtain *pat pats* **where** *pat = last pats′ pats = butlast pats′*
    by *blast*
  from *step False* **have** $length\ pats' \neq 0$
    using *arity-compatible-length inner*
    by (*smt* (*verit*, *ccfv-threshold*) *case-prodD*)
  then have *pats′ = pats* @ [*pat*]
    unfolding ‹*pat* = -› ‹*pats* = -›
    by *simp*
  from *step* **have** *linears pats′*
    using *inner* **by** *auto*
  then have *fdisjnt* (*freess pats*) (*frees pat*)
    unfolding ‹*pats′* = -›
    using *linears-appendD*(*3*) *freess-single*
    by *force*

  from *step* **obtain** *cs* **where** (*pats*, *cs*) |∈| *?grp*
    unfolding ‹*pats* = -›
    by (*metis* (*no-types*, *lifting*) *fgroup-by-complete fst-conv prod.simps*(*2*))

with *step* **have** (*pat*, *rhs'*) |∈| *cs*
unfolding ‹*pat* = -› ‹*pats* = -›
by (*meson fgroup-byD old.prod.case*)
have (*pats*, *Pabs cs*) |∈| *irs*
using ‹*irs* = *map-prod id Pabs* |'| *?grp*› ‹(*pats*, *cs*) |∈| -›
by (*metis* (*no-types*, *lifting*) *eq-snd-iff fst-conv fst-map-prod id-def rev-fimage-eqI snd-map-prod*)
from *step* **obtain** *env'* **where** *match* (*name* $$ *pats'*) *t* = *Some env'* *subst rhs' env'* = *t'*
using *irewrite-step-def* **by** *auto*
have *name* $$ *pats'* = (*name* $$ *pats*) $ *pat*
unfolding ‹*pats'* = -›
by (*simp add*: *app-term-def*)
then obtain t_0 t_1 env_0 env_1 **where** $t = t_0$ $\$_p$ t_1 *match* (*name* $$ *pats*) t_0 = *Some* env_0 *match pat* t_1 = *Some* env_1 *env'* = env_0 $++_f$ env_1
using *match-appE-split*[*OF* ‹*match* (*name* $$ *pats'*) - = -›[*unfolded* ‹*name* $$ *pats'* = -›], *unfolded app-pterm-def*]
by *blast*
with *step* **have** *closed* t_0 *closed* t_1
by *auto*
then have *closed-env* env_0 *closed-env* env_1
using *match-vars*[*OF* ‹*match* - t_0 = -›] *match-vars*[*OF* ‹*match* - t_1 = -›]
unfolding *closed-except-def*
by *auto*
obtain t_0' **where** *subst* (*Pabs cs*) env_0 = t_0'
by *blast*
then obtain *cs'* **where** t_0' = *Pabs cs'* *cs'* = ((λ(*pat*, *rhs*). (*pat*, *subst rhs* (*fmdrop-fset* (*frees pat*) env_0))) |'| *cs*)
using *subst-pterm.simps*(*3*) **by** *blast*
obtain *rhs* **where** *subst rhs'* (*fmdrop-fset* (*frees pat*) env_0) = *rhs*
by *blast*
then have (*pat*, *rhs*) |∈| *cs'*
unfolding ‹*cs'* = -›
using ‹- |∈| *cs*›
by (*metis* (*mono-tags*, *lifting*) *old.prod.case rev-fimage-eqI*)
have env_0 $++_f$ env_1 = (*fmdrop-fset* (*frees pat*) env_0) $++_f$ env_1
apply (*subst fmadd-drop-left-dom*[*symmetric*])
using ‹*match pat* - = -› *match-dom*
by *metis*
have *fdisjnt* (*fmdom* env_0) (*fmdom* env_1)
using *match-dom*
using ‹*match pat* - = -› ‹*match* (*name* $$ *pats*) - = -›
using ‹*fdisjnt* - -›
unfolding *fdisjnt-alt-def*
by (*metis matchs-dom match-list-combE*)
have *subst rhs* env_1 = *t'*
unfolding ‹- = *rhs*›[*symmetric*]
unfolding ‹- = *t'*›[*symmetric*]
unfolding ‹*env'* = -›

unfolding ‹env_0 $++_f$ - = -›
apply (*subst subst-indep*′)
using ‹*closed-env* env_0›
  apply *blast*
using ‹*fdisjnt* (*fmdom* -) -›
unfolding *fdisjnt-alt-def*
by *auto*

show *?thesis*
unfolding ‹*t* = -›
apply *rule*
 apply (*rule r-into-rtranclp*)
 apply (*rule irewrite.intros*(*3*))
 apply *rule*
  apply *fact*+
 apply (*rule irewrite-stepI*)
  apply *fact*+
unfolding ‹t_0′ = -›
apply *rule*
 apply *fact*
using ‹*match pat* t_1 = -› ‹*subst rhs* - = -›
by *force*
qed
qed (*auto intro*: *irewrite.rt-comb*[*unfolded app-pterm-def*] *intro*!: *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* (‹-/ $\vdash_p$/ - ⟶/ -› [*50*,*0*,*50*] *50*) **for** *rs* **where**
step: (*name*, *rhs*) |∈| *rs* ⟹ *rs* $\vdash_p$ *Pconst name* ⟶ *rhs* |
beta: *c* |∈| *cs* ⟹ *c* ⊢ *t* → *t′* ⟹ *rs* $\vdash_p$ *Pabs cs* $\$_p$ *t* ⟶ *t′* |
fun: *rs* $\vdash_p$ *t* ⟶ *t′* ⟹ *rs* $\vdash_p$ *t* $\$_p$ *u* ⟶ *t′* $\$_p$ *u* |
arg: *rs* $\vdash_p$ *u* ⟶ *u′* ⟹ *rs* $\vdash_p$ *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* (‹-/ $\vdash_p$/ - ⟶∗/ -› [*50*,*0*,*50*] *50*) **where**
prewrite-rt rs ≡ (*prewrite rs*)**

Translation from *irule-set* to *prule fset*

definition *finished* :: *irule-set* ⇒ *bool* **where**
finished rs = *fBall rs* (λ(-, *irs*). *arity irs* = *0*)

definition *translate-rhs* :: *irules* ⇒ *pterm* **where**
translate-rhs = *snd* ∘ *fthe-elem*

definition *compile* :: *irule-set* ⇒ *prule fset* **where**
compile = *fimage* (*map-prod id translate-rhs*)

lemma *compile-heads*: *fst* |‘| *compile rs* = *fst* |‘| *rs*
unfolding *compile-def* **by** *simp*

Correctness of translation

lemma *arity-zero-shape*:
assumes *arity-compatibles rs arity rs* = *0* *is-fmap rs rs* ≠ {||}
obtains *t* **where** *rs* = {| ([], *t*) |}
proof −
from *assms* **obtain** *ppats prhs* **where** (*ppats*, *prhs*) |∈| *rs*
by *fast*

moreover {
fix *pats rhs*
assume (*pats*, *rhs*) |∈| *rs*
with *assms* **have** *length pats* = *0*
by (*metis arity-compatible-length*)
hence *pats* = []

```
      by simp
  }
  note all = this

  ultimately have proto: ([], prhs) |∈| rs by auto

  have fBall rs (λ(pats, rhs). pats = [] ∧ rhs = prhs)
    proof safe
      fix pats rhs
      assume cur: (pats, rhs) |∈| rs
      with all show pats = [] .
      with cur have ([], rhs) |∈| rs by auto

      with proto show rhs = prhs
        using assms by (auto dest: is-fmapD)
    qed
  hence fBall rs (λr. r = ([], prhs))
    by blast
  with assms have rs = {| ([], prhs) |}
    by (simp add: singleton-fset-is)
  thus thesis
    by (rule that)
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). ¬ shadows-consts 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 irs ≠ {||} arity irs = 0
        using assms inner unfolding finished-def by blast+
      moreover have arity-compatibles irs
```

using ‹(*name*, *irs*) |∈| *rs*› *inner*
by (*metis* (*no-types*, *lifting*) *case-prodD*)
ultimately obtain *u* **where** *irs* = {| ([], *u*) |}
by (*metis arity-zero-shape*)
hence *rhs* = *u* **and** *u*: ([], *u*) |∈| *irs*
unfolding ‹*rhs* = -› *translate-rhs-def* **by** *simp*+
hence *abs-ish* [] *u*
using *inner* ‹(*name*, *irs*) |∈| *rs*› **by** *fastforce*
thus *is-abs rhs*
unfolding *abs-ish-def* ‹*rhs* = *u*› **by** *simp*

show *wellformed rhs*
using *u* ‹(*name*, *irs*) |∈| *rs*› *inner* **unfolding** ‹*rhs* = *u*›
by *fastforce*

have *closed-except u* {||}
using *u inner* ‹(*name*, *irs*) |∈| *rs*›
by *fastforce*
thus *closed rhs*
unfolding ‹*rhs* = *u*› **.**

{
assume *shadows-consts rhs*
hence *shadows-consts u*
unfolding *compile-def* ‹*rhs* = *u*› **by** *simp*
moreover have ¬ *shadows-consts u*
using *inner* ‹([], *u*) |∈| *irs*› ‹(*name*, *irs*) |∈| *rs*› **by** *fastforce*
ultimately show *False* **by** *blast*
}

have *welldefined u*
using *fbspec*[*OF inner* ‹(*name*, *irs*) |∈| *rs*›, *simplified*] ‹([], *u*) |∈| *irs*›
by *blast*
thus *welldefined rhs*
unfolding ‹*rhs* = *u*› *compile-def*
by *simp*
qed
thus
fBall (*compile rs*) *prule*
fBall (*compile rs*) (λ(-, *rhs*). ¬ *pre-constants.shadows-consts C-info* (*fst* |'| *compile rs*) *rhs*)
fBall (*compile rs*) (λ(-, *rhs*). *pre-constants.welldefined C-info* (*fst* |'| *compile rs*) *rhs*)
unfolding *compile-heads* **by** *auto*
next
show *distinct all-constructors*
by (*fact distinct-ctr*)
qed

theorem (**in** *irules*) *compile-correct*:
  assumes *compile rs* $\vdash_p$ $t \longrightarrow t'$ *finished rs*
  shows *rs* $\vdash_i$ $t \longrightarrow t'$
using *assms*(*1*) **proof** *induction*
  case (*step name rhs*)
  then obtain *irs* **where** *rhs* = *translate-rhs irs* (*name*, *irs*) |∈| *rs*
    unfolding *compile-def* **by** *force*
  hence *arity-compatibles irs*
    using *inner*
    by (*metis* (*no-types*, *lifting*) *case-prodD*)

  have *is-fmap irs irs* ≠ {||} *arity irs* = *0*
    using *assms inner* ‹(*name*, *irs*) |∈| *rs*› **unfolding** *finished-def* **by** *blast+*
  then obtain *u* **where** *irs* = {| ([], *u*) |}
    using ‹*arity-compatibles irs*›
    by (*metis arity-zero-shape*)

  show *?case*
    unfolding ‹*rhs* = -›
    apply (*rule irewrite.step*)
      apply *fact*
    unfolding ‹*irs* = -› *translate-rhs-def irewrite-step-def*
    by (*auto simp*: *const-term-def*)
qed (*auto intro*: *irewrite.intros*)

theorem (**in** *irules*) *compile-complete*:
  assumes *rs* $\vdash_i$ $t \longrightarrow t'$ *finished rs*
  shows *compile rs* $\vdash_p$ $t \longrightarrow t'$
using *assms*(*1*) **proof** *induction*
  case (*step name irs params rhs t t*′)
  hence *arity-compatibles irs*
    using *inner*
    by (*metis* (*no-types*, *lifting*) *case-prodD*)

  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* $\vdash_i$ $t \to t'$
    by *simp*
  hence *t* = *Pconst name*
    unfolding *irewrite-step-def*
    by (*cases t*) (*auto split*: *if-splits simp*: *const-term-def*)
  hence $t' = u$
    using ‹*name*, [], *u* $\vdash_i$ $t \to t'$›
    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* ∘ *snd*)

primrec *srule* :: *srule* ⇒ *bool* **where**
srule (-, *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*). *welldefined 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 *srewrite-step* **where**
cons-match: *srewrite-step* ((*name*, *rhs*) # *rest*) *name rhs* |
cons-nomatch: *name* ≠ *name′* ⟹ *srewrite-step rs name rhs* ⟹ *srewrite-step* ((*name′*, *rhs′*) # *rs*) *name rhs*

lemma *srewrite-stepI0*:
assumes (*name*, *rhs*) ∈ *set rs is-map* (*set rs*)
shows *srewrite-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*) $\in$ *set rs* $\Longrightarrow$ *srewrite-step rs name rhs*
using *map*
by (*metis srewrite-stepI0*)

hide-fact *srewrite-stepI0*

inductive *srewrite* :: *srule list* $\Rightarrow$ *sterm* $\Rightarrow$ *sterm* $\Rightarrow$ *bool* (‹-/ $\vdash_s$/ - $\longrightarrow$/ -› [*50*,*0*,*50*] *50*) **for** *rs* **where**
step: *srewrite-step rs name rhs* $\Longrightarrow$ *rs* $\vdash_s$ *Sconst name* $\longrightarrow$ *rhs* |
beta: *rewrite-first cs t t'* $\Longrightarrow$ *rs* $\vdash_s$ *Sabs cs* $\$_s$ *t* $\longrightarrow$ *t'* |
fun: *rs* $\vdash_s$ *t* $\longrightarrow$ *t'* $\Longrightarrow$ *rs* $\vdash_s$ *t* $\$_s$ *u* $\longrightarrow$ *t'* $\$_s$ *u* |
arg: *rs* $\vdash_s$ *u* $\longrightarrow$ *u'* $\Longrightarrow$ *rs* $\vdash_s$ *t* $\$_s$ *u* $\longrightarrow$ *t* $\$_s$ *u'*

code-pred *srewrite* **.**

abbreviation *srewrite-rt* :: *srule list* $\Rightarrow$ *sterm* $\Rightarrow$ *sterm* $\Rightarrow$ *bool* (‹-/ $\vdash_s$/ - $\longrightarrow$*/ -› [*50*,*0*,*50*] *50*) **where**
srewrite-rt rs $\equiv$ (*srewrite rs*)**

global-interpretation *srewrite*: *rewriting srewrite rs* **for** *rs*
by *standard* (*auto intro*: *srewrite.intros simp*: *app-sterm-def*)+

code-pred (*modes*: *i* $\Rightarrow$ *i* $\Rightarrow$ *o* $\Rightarrow$ *bool*) *srewrite-step* **.**
code-pred (*modes*: *i* $\Rightarrow$ *i* $\Rightarrow$ *o* $\Rightarrow$ *bool*) *srewrite* **.**

Translation from *pterm* to *sterm*

In principle, any function of type ($'a \times 'b$) *fset* $\Rightarrow$ ($'a \times 'b$) *list* that orders by keys would do here. However, For simplicity's sake, we choose a fixed one (*ordered-fmap*) here.

primrec *pterm-to-sterm* :: *pterm* $\Rightarrow$ *sterm* **where**
pterm-to-sterm (*Pconst name*) = *Sconst name* |
pterm-to-sterm (*Pvar name*) = *Svar name* |
pterm-to-sterm (*t* $\$_p$ *u*) = *pterm-to-sterm t* $\$_s$ *pterm-to-sterm u* |
pterm-to-sterm (*Pabs cs*) = *Sabs* (*ordered-fmap* (*map-prod id pterm-to-sterm* |'| *cs*))

lemma *pterm-to-sterm*:
  assumes *no-abs t*
  shows *pterm-to-sterm 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

sterm-to-pterm has to be defined, for technical reasons, in *CakeML-Codegen.Pterm*.

lemma *pterm-to-sterm-wellformed*:
assumes *wellformed t*
shows *wellformed* (*pterm-to-sterm 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*)
using *Pabs* **apply** *simp*
apply (*rule fBallI*)
apply (*erule fimageE*)
apply *auto*[]
using *Pabs*(*2*) **apply** *auto*[]
apply (*rule Pabs*)
using *Pabs*(*2*) **by** *auto*
subgoal
apply (*rule ordered-fmap-distinct*)
apply (*rule is-fmap-image*)
using *Pabs*(*2*) **by** *simp*
subgoal
apply (*subgoal-tac cs* $\neq$ {||})
including *fset.lifting* **apply** *transfer*
unfolding *ordered-map-def*
using *Pabs*(*2*) **by** *auto*
done
qed *auto*

lemma *pterm-to-sterm-sterm-to-pterm*:
assumes *wellformed t*
shows *sterm-to-pterm* (*pterm-to-sterm t*) = *t*
using *assms* **proof** (*induction t*)

case (*Pabs cs*)
note *fset-of-list-map*[*simp del*]
show *?case*
apply *simp*
unfolding *map-prod-def id-apply*
apply (*subst ordered-fmap-image*)
subgoal
apply (*rule is-fmap-image*)
using *Pabs* **by** *simp*
apply (*subst ordered-fmap-set-eq*)
subgoal
apply (*rule is-fmap-image*)
apply (*rule is-fmap-image*)
using *Pabs* **by** *simp*
subgoal
apply (*subst fset.map-comp*)
apply (*subst map-prod-def*[*symmetric*])+
unfolding *o-def*
apply (*subst prod.map-comp*)
apply (*subst id-def*[*symmetric*])+
apply *simp*
apply (*subst map-prod-def*)
unfolding *id-def*
apply (*rule fset-map-snd-id*)
apply *simp*
apply (*rule Pabs*)
using *Pabs*(*2*) **by** (*auto simp*: *snds.simps*)
done
qed *auto*

corollary *pterm-to-sterm-frees*: *wellformed t* $\Longrightarrow$ *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 $\Longrightarrow$ *wellformed t* $\Longrightarrow$ *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* $\Longrightarrow$ *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 $\Longrightarrow$ *shadows-consts t* $\longleftrightarrow$ *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* $\Rightarrow$ *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 (*subst ordered-fmap-set-eq*)
   apply (*subst map-prod-def*, *subst 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 (*subst 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
interpret *c*: *constants* - *fset-of-list* (*map fst* (*compile rs*))
by (*simp add*: *constants-axioms compile-heads*)
have *all-consts*: *c.all-consts* = *all-consts*
by (*simp add*: *compile-heads*)

note *fset-of-list-map*[*simp del*]
have *list-all* (λ(-, *rhs*). ¬ *shadows-consts rhs*) (*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 fBall-pred-weaken*[**where** P = λ(-, *rhs*). ¬ *shadows-consts rhs*])
subgoal for *x*
apply (*cases x*, *simp*)
apply (*subst* (*asm*) *pterm-to-sterm-shadows*)
using *all-rules* **apply** *force*
by *simp*
subgoal
using *not-shadows* **by** *force*

done
thus *list-all* (λ(-, *rhs*). ¬ *pre-constants.shadows-consts C-info* (*fst* |\`| *fset-of-list* (*compile rs*)) *rhs*) (*compile rs*)
unfolding *compile-heads all-consts* **.**
next
show *fdisjnt* (*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* (λp *s*. p = *sterm-to-pterm s*)
proof (*standard*, *goal-cases*)
case (*5 name t*)
then show *?case* **by** (*cases t*) (*auto simp*: *const-sterm-def const-pterm-def split*: *option.splits*)
next
case (*6* u_1 u_2 *t*)
then show *?case* **by** (*cases t*) (*auto simp*: *app-sterm-def app-pterm-def split*: *option.splits*)
qed (*auto simp*: *const-sterm-def const-pterm-def app-sterm-def app-pterm-def*)

lemma *srelated-subst*:
assumes *srelated.P-env penv senv*
shows *subst* (*sterm-to-pterm t*) *penv* = *sterm-to-pterm* (*subst t senv*)
using *assms*
proof (*induction t arbitrary*: *penv senv*)
case (*Svar name*)
thus *?case*
by (*cases rule*: *fmrel-cases*[**where** *x* = *name*]) *auto*
next
case (*Sabs cs*)
show *?case*
apply *simp*
including *fset.lifting*
apply (*transfer′ fixing*: *cs penv senv*)

unfolding *set-map image-comp*
apply (*rule image-cong*[*OF refl*])
unfolding *comp-apply*
apply (*case-tac x*)
apply *hypsubst-thin*
apply *simp*
apply (*rule Sabs*)
apply *assumption*
apply (*simp add*: *snds.simps*)
apply *rule*
apply (*rule Sabs*)
done
qed *auto*

context begin

private lemma *srewrite-step-non-empty*: *srewrite-step* rs' *name rhs* $\Longrightarrow$ $rs' \neq []$
by (*induct rule*: *srewrite-step.induct*) *auto*

private lemma *compile-consE*:
assumes (*name*, rhs') # *rest* = *compile rs is-fmap rs*
obtains *rhs* **where** rhs' = *pterm-to-sterm rhs* (*name*, *rhs*) $|\in|$ *rs rest* = *compile* (*rs* − {| (*name*, *rhs*) |})
proof −
from *assms* **have** *ordered-fmap* (*map-prod id pterm-to-sterm* |\`| *rs*) = (*name*, rhs') # *rest*
unfolding *compile-def*
by *simp*
hence (*name*, rhs') $\in$ *set* (*ordered-fmap* (*map-prod id pterm-to-sterm* |\`| *rs*))
by *simp*

have (*name*, rhs') $|\in|$ *map-prod id pterm-to-sterm* |\`| *rs*
apply (*rule ordered-fmap-sound*)
subgoal
unfolding *map-prod-def id-apply*
apply (*rule is-fmap-image*)
apply *fact*
done
subgoal by *fact*
done
then obtain *rhs* **where** rhs' = *pterm-to-sterm rhs* (*name*, *rhs*) $|\in|$ *rs*
by *auto*

have *rest* = *compile* (*rs* − {| (*name*, *rhs*) |})
unfolding *compile-def*
apply (*subst inj-on-fimage-set-diff*[**where** *C* = *rs*])
subgoal
apply (*rule inj-onI*)
apply *safe*

```
      apply auto
     using ‹is-fmap rs› by (blast dest: is-fmapD)
    subgoal by simp
    subgoal using ‹(name, rhs) |∈| rs› by simp
    subgoal
     apply simp
     apply (subst ordered-fmap-remove)
       apply (subst map-prod-def)
     unfolding id-apply
       apply (rule is-fmap-image)
       apply fact
     using ‹(name, rhs) |∈| rs› apply force
     apply (subst ‹rhs' = pterm-to-sterm rhs›[symmetric])
     apply (subst ‹ordered-fmap - = -›[unfolded id-def])
     by simp
    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, sterm-to-pterm 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-sterm-to-pterm)
    using fbspec[OF ‹fBall rs prule› ‹(name, rhs) |∈| rs›] apply force
    by fact
next
  case (cons-nomatch name name_1 rest rhs rhs_1')
  then obtain rhs_1 where rhs_1' = pterm-to-sterm rhs_1 (name_1, rhs_1) |∈| rs rest
= compile (rs − {| (name_1, rhs_1) |})
    by (auto elim: compile-consE)

  let ?rs' = rs − {| (name_1, rhs_1) |}
  have (name, sterm-to-pterm rhs) |∈| ?rs'
    proof (intro cons-nomatch)
     show rest = compile ?rs'
       by fact

     show is-fmap (rs |−| {|(name_1, rhs_1)|})
       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 ⊢s u ⟶ u′ prules C rs
  shows rs ⊢p sterm-to-pterm u ⟶ sterm-to-pterm 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-pterm 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-pterm rhs) env′ = sterm-to-pterm (subst rhs env)
    by (simp add: srelated-subst)

  let ?rhs′ = sterm-to-pterm rhs

  have (pat, ?rhs′) |∈| fset-of-list (map (map-prod id sterm-to-pterm) cs)
    using ‹(pat, rhs) ∈ set cs›
    including fset.lifting
    by transfer′ force

  note fset-of-list-map[simp del]
  show ?case
    apply simp
    apply (rule prewrite.intros)
     apply fact
    unfolding rewrite-step.simps
    apply (subst map-option-eq-Some)
    apply (intro exI conjI)
     apply fact
    unfolding ‹t′ = -›
    by fact
next
  case (step name rhs)
  hence (name, sterm-to-pterm rhs) |∈| rs
    unfolding prules-def prules-axioms-def
    by (metis compile-correct-step)
  thus ?case
    by (auto intro: prewrite.intros)
qed (auto intro: prewrite.intros)
```

end

lemma (**in** *prules*) *compile-correct*:
  assumes *compile rs* $\vdash_s$ *u* $\longrightarrow$ *u'*
  shows *rs* $\vdash_p$ *sterm-to-pterm u* $\longrightarrow$ *sterm-to-pterm u'*
by (*rule compile-correct0*) (*fact* | *standard*)+

hide-fact *compile-correct0*

Completeness of translation

global-interpretation *srelated'*: *term-struct-rel-strong* (λ*p s. pterm-to-sterm p* = *s*)
proof (*standard*, *goal-cases*)
  case (*1 t name*)
  then show *?case* **by** (*cases t*) (*auto simp*: *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*: *app-sterm-def app-pterm-def split*: *option.splits*)
qed (*auto simp*: *const-sterm-def const-pterm-def app-sterm-def app-pterm-def*)

corollary *srelated-env-unique*:
  srelated'.P-env penv senv $\Longrightarrow$ *srelated'.P-env penv senv'* $\Longrightarrow$ *senv* = *senv'*
apply (*subst* (*asm*) *fmrel-iff*)+
apply (*subst* (*asm*) *option.rel-sel*)+
apply (*rule fmap-ext*)
by (*metis option.exhaust-sel*)

lemma *srelated-subst'*:
  assumes *srelated'.P-env penv senv wellformed t*
  shows *pterm-to-sterm* (*subst t penv*) = *subst* (*pterm-to-sterm t*) *senv*
using *assms* **proof** (*induction t arbitrary*: *penv senv*)
  case (*Pvar name*)
  thus *?case*
    by (*cases rule*: *fmrel-cases*[**where** *x* = *name*]) *auto*
next
  case (*Pabs cs*)
  hence *is-fmap cs*
    by *force*

  show *?case*
    apply *simp*
    unfolding *map-prod-def id-apply*
    apply (*subst ordered-fmap-image*[*symmetric*])
     apply *fact*
    apply (*subst fset.map-comp*[*symmetric*])

apply (*subst ordered-fmap-image*[*symmetric*])
subgoal by (*rule is-fmap-image*) *fact*
apply (*subst ordered-fmap-image*[*symmetric*])
apply *fact*
apply *auto*
apply (*drule ordered-fmap-sound*[*OF* ‹*is-fmap cs*›])
subgoal for *pat rhs*
apply (*rule Pabs*)
apply *assumption*
apply *auto*
using *Pabs* **by** *force*+
done
qed *auto*

lemma *srelated-find-match*:
assumes *find-match cs t* = *Some* (*penv*, *pat*, *rhs*) *srelated'.P-env penv senv*
shows *find-match* (*map* (*map-prod id pterm-to-sterm*) *cs*) (*pterm-to-sterm t*) = *Some* (*senv*, *pat*, *pterm-to-sterm rhs*)
proof −
let *?cs'* = *map* (*map-prod id pterm-to-sterm*) *cs*
let *?t'* = *pterm-to-sterm t*
have ∗: *list-all2* (*rel-prod* (=) (λ*p s*. *pterm-to-sterm p* = *s*)) *cs ?cs'*
unfolding *list.rel-map*
by (*auto intro*: *list.rel-refl*)

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 $rs \vdash_p t \longrightarrow t'$ *wellformed t*
shows *compile* $rs \vdash_s$ *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 (metis fimage-eqI id-def map-prod-simp)
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-sterm 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
begin

3.5.1 Big-step semantics evaluating to irreducible *sterm*s

inductive (**in** *constructors*) *seval* :: *srule list* ⇒ (*name*, *sterm*) *fmap* ⇒ *sterm* ⇒ *sterm* ⇒ *bool* (‹-, -/ $\vdash_s$/ - ↓/ -› [*50*,*0*,*50*] *50*) **for** *rs* **where**
const: (*name*, *rhs*) ∈ *set rs* ⟹ *rs*, Γ $\vdash_s$ *Sconst name* ↓ *rhs* |
var: *fmlookup* Γ *name* = *Some val* ⟹ *rs*, Γ $\vdash_s$ *Svar name* ↓ *val* |
abs: *rs*, Γ $\vdash_s$ *Sabs cs* ↓ *Sabs* (*map* (λ(*pat*, *t*). (*pat*, *subst t* (*fmdrop-fset* (*frees pat*) Γ))) *cs*) |
comb:
rs, Γ $\vdash_s$ *t* ↓ *Sabs cs* ⟹ *rs*, Γ $\vdash_s$ *u* ↓ *u'* ⟹
find-match cs u' = *Some* (*env*, -, *rhs*) ⟹
rs, Γ $++_f$ *env* $\vdash_s$ *rhs* ↓ *val* ⟹
rs, Γ $\vdash_s$ *t* $\$_s$ *u* ↓ *val* |
constr:
name |∈| *C* ⟹
list-all2 (*seval rs* Γ) *ts us* ⟹
rs, Γ $\vdash_s$ *name* $$ *ts* ↓ *name* $$ *us*

lemma (**in** *constructors*) *seval-closed*:
assumes *rs*, Γ $\vdash_s$ *t* ↓ *u closed-srules rs closed-env* Γ *closed-except t* (*fmdom* Γ)
shows *closed u*
using *assms* **proof** *induction*
case (*const name rhs* Γ)

```
  thus ?case
    by (auto simp: list-all-iff)
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))
        using comb by simp
    qed fact
next
  case (abs Γ cs)
  show ?case
    apply (subst subst-sterm.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*, $\Gamma \vdash_s$ *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* ($\Gamma \mathbin{++_f}$ *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 (abs Γ cs)
  show ?case
    apply (subst subst-sterm.simps[symmetric])
    apply (rule subst-shadows)
     apply fact+
    done
next
  case (constr name Γ ts us)
  have list-all (Not ∘ shadows-consts) 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 const-sterm-def app-sterm-def)
    qed simp
  thus ?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, Γ ⊢s name $$ args ↓ Sabs cs
  shows name ∈ dom (map-of rs)
using assms
```

```
proof (induction Γ 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 ?case ..
next
  case (comb Γ t cs′ u 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 fmsubset-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 Γ cs)
  hence subst (subst (Sabs cs) Γ') Γ = subst (Sabs cs) Γ
    by (metis subst-twice fmsubset-pred)
  moreover have rs, Γ ⊢s subst (Sabs cs) Γ' ↓ subst (subst (Sabs cs) Γ') Γ
    apply simp
    apply (subst map-map[symmetric])
    apply (rule seval.abs)
    done
  ultimately have rs, Γ ⊢s subst (Sabs cs) Γ' ↓ subst (Sabs cs) Γ
    by metis
  thus ?case by simp
next
  case (constr name Γ ts us)
  hence list-all2 (λt. seval rs Γ (subst t Γ')) 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*, $\Gamma \vdash_s t \downarrow u$ *fmrestrict-fset* S Γ = *fmrestrict-fset* S Γ' *closed-except* t S
assumes S $|\subseteq|$ *fmdom* Γ *closed-srules rs closed-env* Γ
shows *rs*, $\Gamma' \vdash_s t \downarrow u$
using *assms* **proof** (*induction arbitrary*: Γ' S)
case (*var* Γ *name val*)
hence *name* $|\in|$ S
by (*simp add*: *closed-except-def*)
hence *fmlookup* Γ *name* = *fmlookup* Γ' *name*
using ‹*fmrestrict-fset* S Γ = -›
unfolding *fmfilter-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* Γ *cs*)

— Intentionally local: not really a useful lemma outside of its scope
have $*$: *fmdrop-fset* S (*fmrestrict-fset* T m) = *fmrestrict-fset* (T $|\cup|$ 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) $\in$ *set cs*
with *abs* **have** *closed-except* t (S $|\cup|$ *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*, Γ′ $\vdash_s$ *t* ↓ *Sabs cs rs*, Γ′ $\vdash_s$ *u* ↓ *u′*
using *comb* **by** (*auto simp*: *Sterm.closed-except-simps*)
next
show *rs*, Γ′ $++_f$ *env* $\vdash_s$ *rhs* ↓ *val*
proof (*rule comb*)
have *fmrestrict-fset* (*S* |∪| *fmdom env*) (Γ $++_f$ *env*) = *fmrestrict-fset* (*S* |∪| *fmdom env*) (Γ′ $++_f$ *env*)
using *comb*(*8*)
unfolding *fmfilter-alt-defs*
including *fmap.lifting* **and** *fset.lifting*
by *transfer′* (*auto simp*: *map-filter-def fun-eq-iff map-add-def split*: *option.splits if-splits*)

thus *fmrestrict-fset* (*S* |∪| *frees pat*) (Γ $++_f$ *env*) = *fmrestrict-fset* (*S* |∪| *frees pat*) (Γ′ $++_f$ *env*)
unfolding ‹*fmdom env* = -› **.**
next
have *closed-except t S*
using *comb* **by** (*simp add*: *Sterm.closed-except-simps*)

have *closed* (*Sabs cs*)
apply (*rule seval-closed*)
apply *fact*+
using ‹*closed-except t S*› ‹*S* |⊆| *fmdom* Γ›
unfolding *closed-except-def* **apply** *simp*

```
          done

        have (pat, rhs) ∈ set cs
          using ‹find-match - - = -› by (rule find-match-elem)
        hence closed-except rhs (frees pat)
          using ‹closed (Sabs cs)› by (auto dest: closed-except-sabs)
        thus closed-except rhs (S |∪| frees pat)
          unfolding closed-except-def by auto
      next
        show S |∪| frees pat |⊆| fmdom (Γ ++f env)
          apply simp
          apply (intro conjI)
          using comb(10) apply blast
          unfolding ‹fmdom env = -› by blast
      next
        have closed-except u S
          using comb by (auto simp: closed-except-def)

        show closed-env (Γ ++f env)
          apply rule
           apply fact
          apply (rule closed.match[where t = u' and pat = pat])
          subgoal
            by (rule 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 *srewrite*

context *srules* **begin context begin**

private lemma *seval-correct0*:

assumes *rs*, $\Gamma \vdash_s t \downarrow u$ *closed-except t* (*fmdom* Γ) *closed-env* Γ
shows *rs* $\vdash_s$ *subst t* $\Gamma \longrightarrow\!*\ u$
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*: *Sterm.closed-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*) $\in$ *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* $|\subseteq|$ *frees pat*
by (*simp add*: *closed-except-def*)
moreover have *fmdom env* = *frees pat*
using ‹*match pat* u' = -› **by** (*auto simp*: *match-dom*)
ultimately have *frees rhs* $|\subseteq|$ *fmdom env*
by *simp*
hence *subst rhs* (Γ $++_f$ *env*) = *subst rhs env*
by (*rule subst-add-shadowed-env*)

have *rs* $\vdash_s$ *subst t* Γ $\$_s$ *subst u* $\Gamma \longrightarrow\!*$ *Sabs cs* $\$_s$ u'
using *comb* **by** (*force intro*: *srewrite.rt-comb*[*unfolded app-sterm-def*] *simp*: *Sterm.closed-except-simps*)
also have *rs* $\vdash_s$ *Sabs cs* $\$_s$ $u' \longrightarrow\!*$ *subst rhs env*
using *comb* ‹*closed* u'› **by** (*force intro*: *srewrite.beta find-match-rewrite-first*)
also have *rs* $\vdash_s$ *subst rhs env* $\longrightarrow\!*$ *subst rhs* (Γ $++_f$ *env*)
unfolding ‹*subst rhs* (Γ $++_f$ *env*) = -› **by** *simp*
also have *rs* $\vdash_s$ *subst rhs* (Γ $++_f$ *env*) $\longrightarrow\!*$ *val*
proof (*rule comb*)
show *closed-except rhs* (*fmdom* (Γ $++_f$ *env*))
using *comb* ‹*match pat* u' = *Some env*› ‹*fmdom env* = -› ‹*frees rhs* $|\subseteq|$ *frees pat*›
by (*auto simp*: *closed-except-def*)
next
show *closed-env* (Γ $++_f$ *env*)
using *comb* ‹*match pat* u' = *Some env*› ‹*closed* u'›
by (*blast intro*: *closed.match*)

    qed

  finally show *?case* **by** *simp*
next
  case (*constr name* Γ *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 *rs*, *fmempty* $\vdash_s$ *t* ↓ *u closed t*
  shows *rs* $\vdash_s$ *t* ⟶∗ *u*
proof −
  have *closed-except t* (*fmdom fmempty*)
    using *assms* **by** *simp*
  with *assms* **have** *rs* $\vdash_s$ *subst t fmempty* ⟶∗ *u*
    by (*fastforce intro*!: *seval-correct0*)
  thus *?thesis*
    by *simp*
qed

end end

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*) ⟷ *vwellformed 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* (‹-, -/ $\vdash_v$/ - ↓/ -› [*50*,*0*,*50*] *50*) **for** *rs* **where**
const: (*name*, *rhs*) ∈ *set rs* ⟹ *rs*, Γ $\vdash_v$ *Sconst name* ↓ *rhs* |
var: *fmlookup* Γ *name* = *Some val* ⟹ *rs*, Γ $\vdash_v$ *Svar name* ↓ *val* |
abs: *rs*, Γ $\vdash_v$ *Sabs cs* ↓ *Vabs cs* Γ |
comb:
rs, Γ $\vdash_v$ *t* ↓ *Vabs cs* Γ′ ⟹ *rs*, Γ $\vdash_v$ *u* ↓ *u′* ⟹
vfind-match cs u′ = *Some* (*env*, -, *rhs*) ⟹
rs, Γ′ $++_f$ *env* $\vdash_v$ *rhs* ↓ *val* ⟹
rs, Γ $\vdash_v$ *t* $\$_s$ *u* ↓ *val* |
rec-comb:
rs, Γ $\vdash_v$ *t* ↓ *Vrecabs css name* Γ′ ⟹
fmlookup css name = *Some cs* ⟹
rs, Γ $\vdash_v$ *u* ↓ *u′* ⟹
vfind-match cs u′ = *Some* (*env*, -, *rhs*) ⟹
rs, Γ′ $++_f$ *env* $\vdash_v$ *rhs* ↓ *val* ⟹
rs, Γ $\vdash_v$ *t* $\$_s$ *u* ↓ *val* |
constr:
name |∈| *C* ⟹
list-all2 (*veval rs* Γ) *ts us* ⟹
rs, Γ $\vdash_v$ *name* $$ *ts* ↓ *Vconstr name us*

lemma (**in** *vrules*) *veval-wellformed*:
assumes *rs*, Γ $\vdash_v$ *t* ↓ *v wellformed t wellformed-venv* Γ
shows *vwellformed v*
using *assms* **proof** *induction*
case *const*
thus *?case*
using *all-rules*
by (*auto simp*: *list-all-iff*)
next
case *comb*
show *?case*

apply (*rule comb*)
using *comb* **by** (*auto simp*: *list-all-iff dest*: *vfind-match-elem intro*: *vwellformed.vmatch-env*)
next
case (*rec-comb* Γ *t css name* Γ' *cs u* u' *env pat rhs val*)
hence (*pat*, *rhs*) $\in$ *set cs vmatch* (*mk-pat pat*) u' = *Some env*
by (*metis vfind-match-elem*)+

show *?case*
proof (*rule rec-comb*)
have *wellformed t*
using *rec-comb* **by** *simp*
have *vwellformed* (*Vrecabs css name* Γ')
by (*rule rec-comb*) *fact*+
thus *wellformed rhs*
using *rec-comb* ‹(*pat*, *rhs*) $\in$ *set cs*›
by (*auto simp*: *list-all-iff*)

have *wellformed-venv* Γ'
using ‹*vwellformed* (*Vrecabs css name* Γ')› **by** *simp*
moreover have *wellformed-venv env*
using *rec-comb* ‹*vmatch* (*mk-pat pat*) u' = *Some env*›
by (*auto intro*: *vwellformed.vmatch-env*)
ultimately show *wellformed-venv* (Γ' $++_f$ *env*)
by *blast*
qed
next
case (*constr name* Γ *ts us*)
have *list-all vwellformed 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*, $\Gamma \vdash_v t \downarrow v$ *closed-except t* (*fmdom* 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*) $\in$ *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 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 wellformed-venv env
        using comb pat
        by (auto intro: vwellformed.vmatch-env veval-wellformed)
      ultimately show wellformed-venv (Γ' ++f env)
        by blast

      have vclosed (Vabs cs Γ')
        using comb by (auto simp: Sterm.closed-except-simps)
      hence closed-except rhs (fmdom Γ' |∪| frees pat)
        using pat by (auto simp: list-all-iff)

      moreover have fmdom env = frees pat
        using ‹vwellformed (Vabs cs Γ')› pat
        by (auto simp: vmatch-dom mk-pat-frees list-all-iff)

      ultimately show closed-except rhs (fmdom (Γ' ++f env))
        using ‹vclosed (Vabs cs Γ')›
        by simp
    qed
next
  case (rec-comb Γ t css name Γ' cs u 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 rec-comb)
      have vclosed u′
        using rec-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 rec-comb by (auto simp: Sterm.closed-except-simps)
    next
      have wellformed t
        using rec-comb by simp
      have vwellformed (Vrecabs css name Γ′)
        by (rule veval-wellformed) fact+
      thus wellformed rhs
        using pat rec-comb by (auto simp: list-all-iff)

      have wellformed-venv Γ′
        using ‹vwellformed (Vrecabs css name Γ′)› by simp
      moreover have wellformed-venv env
        using rec-comb pat
        by (auto intro: vwellformed.vmatch-env veval-wellformed)
      ultimately show wellformed-venv (Γ′ ++f env)
        by blast

      have wellformed-clauses cs
       using ‹vwellformed (Vrecabs css name Γ′)› ‹fmlookup css name = Some cs›
        by auto

      have vclosed (Vrecabs css name Γ′)
        using rec-comb by (auto simp: Sterm.closed-except-simps)
      hence closed-except rhs (fmdom Γ′ |∪| frees pat)
        using rec-comb pat by (auto simp: list-all-iff)
      moreover have fmdom env = frees pat
        using ‹wellformed-clauses cs› pat
        by (auto simp: list-all-iff vmatch-dom mk-pat-frees)
      ultimately show closed-except rhs (fmdom (Γ′ ++f env))
        using ‹vclosed (Vrecabs css name Γ′)›
        by simp
    qed
next
  case (constr name Γ ts us)
  have list-all vclosed us
    using ‹list-all2 - - -› ‹closed-except (- $$ -) -› ‹wellformed (name $$ ts)›
    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: list-all-iff Sterm.closed-except-simps)
    qed simp
  thus ?case
```

    by (*simp add*: *list-all-iff*)
qed (*auto simp*: *Sterm.closed-except-simps*)

lemma (**in** *vrules*) *veval-constructor-value*:
  assumes *rs*, $\Gamma \vdash_v t \downarrow v$ *vconstructor-value-env* Γ
  shows *vconstructor-value v*
using *assms* **proof** *induction*
  case (*comb* Γ *t cs* Γ' *u u' env pat rhs val*)
  hence (*pat*, *rhs*) $\in$ *set cs vmatch* (*mk-pat pat*) *u'* = *Some env*
    by (*metis vfind-match-elem*)+
  hence *vconstructor-value-env* (Γ' $++_f$ *env*)
    using *comb* **by** (*auto intro*: *vconstructor-value.vmatch-env*)
  thus *?case*
    using *comb* **by** *auto*
next
  case (*constr name* Γ *ts us*)
  hence *list-all vconstructor-value us*
    by (*auto elim*: *list-all2-rightE*)
  with *constr* **show** *?case*
    by *simp*
next
  case *const*
  thus *?case*
    using *vconstructor-value-rs*
    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*) $\in$ *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*, $\Gamma \vdash_v t \downarrow v$ *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 pat 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 fmpred (λ-. vwelldefined) (Γ′ ++f env)
        proof
          show fmpred (λ-. vwelldefined) env
           using rec-comb by (auto dest: vfind-match-elem intro: vwelldefined.vmatch-env)
        next
          show fmpred (λ-. vwelldefined) Γ′
            using rec-comb by auto
        qed
    next
      have vwelldefined (Vrecabs css name Γ′)
        using rec-comb by auto

      thus welldefined rhs
        apply simp
        apply (elim conjE)
        apply (drule fmpredD[where m = css])
        using ‹(pat, rhs) ∈ set cs› rec-comb by (auto simp: list-all-iff)
    qed
next
  case (constr name Γ ts us)
  have list-all vwelldefined us
    using ‹list-all2 - - -› ‹welldefined (- $$ -)›
    proof (induction ts us rule: list.rel-induct)
      case (Cons v vs u us)
      with constr show ?case
```

unfolding *welldefined.list-comb*
by *auto*
qed *simp*
with *constr* **show** *?case*
by (*simp add*: *list-all-iff all-consts-def*)
next
case *abs*
thus *?case*
unfolding *welldefined-sabs* **by** *auto*
qed *auto*

Correctness wrt *constructors.seval*

context *vrules* **begin**

definition *rs′* :: *srule list* **where**
rs′ = *value-to-sterm-rules rs*

lemma *value-to-sterm-srules*: *srules C-info rs′*
proof
show *distinct* (*map 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

end

When we evaluate *sterm*s 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* ⇀ *value* (which is usually called Γ′). The contained case-lambda might not be closed.

The proof idea is that we can always substitute with Γ′ and obtain a regular *sterm* 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 Γ at all. But pattern matching and substitution proceeds only with that hidden environment.

context *vrules* **begin**

context begin

private lemma *veval-correct0*:
assumes *rs*, $\Gamma \vdash_v t \downarrow v$ *wellformed t wellformed-venv* Γ
assumes *closed-except t* (*fmdom* Γ) *closed-venv* Γ
assumes *vconstructor-value-env* Γ
shows *rs′*, *fmmap value-to-sterm* $\Gamma \vdash_s t \downarrow$ *value-to-sterm v*
using *assms* **proof** *induction*

case (*constr name* Γ *ts us*)

have *list-all2* (*seval rs′* (*fmmap value-to-sterm* Γ)) *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* ∈ *set ts u* ∈ *set us*
assume *rs*, $\Gamma \vdash_v t \downarrow u$

show *wellformed t closed-except t* (*fmdom* Γ)
using ‹*t* ∈ *set ts*› *constr*
unfolding *wellformed.list-comb closed.list-comb list-all-iff*
by *auto*
qed (*rule constr* | *assumption*)+

thus *?case*
using ‹*name* |∈| *C*›
by (*auto intro*: *seval.constr*)
next
case (*comb* Γ *t cs* Γ′ *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*
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* ∘ *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 $(pat, rhs) \in set\ cs\ match\ pat\ (value\text{-}to\text{-}sterm\ u') = Some\ senv$
  by (*auto dest*: *find-match-elem*)
hence $fmdom\ senv = frees\ pat$
  by (*simp add*: *match-dom*)

moreover have $senv = fmmap\ value\text{-}to\text{-}sterm\ venv$
  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.vmatch-env*) *fact*+

have *wellformed u*
  using *comb* **by** *simp*
have *vwellformed u′*
  by (*rule veval-wellformed*) *fact*+
have *wellformed-venv venv*
  by (*rule vwellformed.vmatch-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.vmatch-env*) *fact*+

have *closed-venv* Γ′
  using ‹*vclosed* (*Vabs cs* Γ′)› **by** *simp*

let $?subst = \lambda pat\ t.\ subst\ t\ (fmdrop\text{-}fset\ (frees\ pat)\ (fmmap\ value\text{-}to\text{-}sterm\ \Gamma'))$

1. We know the following (induction hypothesis): rs', $fmmap\ value\text{-}to\text{-}sterm\ (\Gamma' +\!+_f\ venv) \vdash_s rhs \downarrow value\text{-}to\text{-}sterm\ val$
2. ... first, we can deduce using *ssubst-eval* that this is equivalent to substituting *rhs* first: rs', $fmmap\ value\text{-}to\text{-}sterm\ (\Gamma' +\!+_f\ venv) \vdash_s subst\ rhs\ (fmdrop\text{-}fset\ (frees\ pat)\ (fmmap\ value\text{-}to\text{-}sterm\ \Gamma')) \downarrow value\text{-}to\text{-}sterm\ val$

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* $(\Gamma ++_f$ *venv*$) \vdash_s$ *subst rhs* (*fmdrop-fset* (*frees pat*) (*fmmap value-to-sterm* Γ')) $\downarrow$ *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* $\Gamma \vdash_s t \,\$_s\, u \downarrow$ *value-to-sterm val*

have *rs'*, *fmmap value-to-sterm* $(\Gamma' ++_f$ *venv*$) \vdash_s$ *?subst pat rhs* $\downarrow$ *value-to-sterm val*
  proof (*rule ssubst-eval*)
    show *rs'*, *fmmap value-to-sterm* $(\Gamma' ++_f$ *venv*$) \vdash_s$ *rhs* $\downarrow$ *value-to-sterm val*
      proof (*rule comb*)
        have *linear pat closed-except rhs* (*fmdom* Γ' $|\cup|$ *frees pat*)
          using ‹(*pat*, *rhs*) $\in$ *set cs*› ‹*vwellformed* (*Vabs cs* Γ')› ‹*vclosed* (*Vabs cs* Γ')›
          by (*auto simp*: *list-all-iff*)
        hence *closed-except rhs* (*fmdom* Γ' $|\cup|$ *fmdom venv*)
          using ‹*vmatch* (*mk-pat pat*) u' = *Some venv*›
          by (*auto simp*: *mk-pat-frees vmatch-dom*)
        thus *closed-except rhs* (*fmdom* $(\Gamma' ++_f$ *venv*))
          by *simp*
      next
        show *wellformed-venv* $(\Gamma' ++_f$ *venv*)
          using ‹*wellformed-venv* Γ'› ‹*wellformed-venv venv*›
          by *blast*
      next
        show *closed-venv* $(\Gamma' ++_f$ *venv*)
          using ‹*closed-venv* Γ'› ‹*closed-venv venv*›
          by *blast*
      next
        show *vconstructor-value-env* $(\Gamma' ++_f$ *venv*)
          using ‹*vconstructor-value-env* Γ'› ‹*vconstructor-value-env venv*›
          by *blast*
      next
        show *wellformed rhs*
          using ‹(*pat*, *rhs*) $\in$ *set cs*› ‹*vwellformed* (*Vabs cs* Γ')›
          by (*fastforce simp*: *list-all-iff*)
      qed
  next
    have *fmdrop-fset* (*fmdom venv*) $\Gamma' \subseteq_f \Gamma' ++_f$ *venv*
      including *fmap.lifting* **and** *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* (*frees pat*) (*fmmap value-to-sterm* Γ') $\subseteq_f$ *fmmap value-to-sterm* $(\Gamma' ++_f$ *venv*)
      unfolding ‹*fmdom venv* = *frees pat*›
      by (*metis fmdrop-fset-fmmap fmmap-subset*)
  next

show *closed-env* (*fmmap value-to-sterm* (Γ' $++_f$ *venv*))
apply *auto*
apply *rule*
apply (*rule vclosed.value-to-sterm-env*, *fact*)+
done
next
show *value-env* (*fmmap value-to-sterm* (Γ' $++_f$ *venv*))
apply *auto*
apply *rule*
apply (*rule vconstructor-value.value-to-sterm-env*, *fact*)+
done
qed

show *?case*
proof (*rule seval.comb*)
have *rs'*, *fmmap value-to-sterm* $\Gamma \vdash_s t \downarrow$ *value-to-sterm* (*Vabs cs* Γ')
using *comb* **by** (*auto simp*: *closed-except-def*)
thus *rs'*, *fmmap value-to-sterm* $\Gamma \vdash_s t \downarrow$ *Sabs* (*map* (λ(*pat*, *t*). (*pat*, *?subst pat t*)) *cs*)
by *simp*
next
show *rs'*, *fmmap value-to-sterm* $\Gamma \vdash_s u \downarrow$ *value-to-sterm* u'
using *comb* **by** (*simp add*: *closed-except-def*)
next
show *rs'*, *fmmap value-to-sterm* Γ $++_f$ *senv* $\vdash_s$ *?subst pat rhs* $\downarrow$ *value-to-sterm val*
proof (*rule seval-agree-eq*)
show *rs'*, *fmmap value-to-sterm* Γ' $++_f$ *fmmap value-to-sterm venv* $\vdash_s$ *?subst pat rhs* $\downarrow$ *value-to-sterm val*
using ‹*rs'*, *fmmap value-to-sterm* (Γ' $++_f$ *venv*) $\vdash_s$ *?subst pat rhs* $\downarrow$ *value-to-sterm val*› **by** *simp*
next
show *fmrestrict-fset* (*frees pat*) (*fmmap value-to-sterm* Γ' $++_f$ *fmmap value-to-sterm venv*) =
fmrestrict-fset (*frees pat*) (*fmmap value-to-sterm* Γ $++_f$ *senv*)
unfolding ‹*senv* = -›
apply (*subst* ‹*fmdom venv* = *frees pat*›[*symmetric*])+
apply (*subst fmdom-map*[*symmetric*])
apply (*subst fmadd-restrict-right-dom*)
apply (*subst fmdom-map*[*symmetric*])
apply (*subst fmadd-restrict-right-dom*)
apply *simp*
done
next
have *closed* (*value-to-sterm* (*Vabs cs* Γ'))
using ‹*vclosed* (*Vabs cs* Γ')›
by (*rule vclosed.value-to-sterm*)
thus *closed-except* (*subst rhs* (*fmdrop-fset* (*frees pat*) (*fmmap value-to-sterm* Γ'))) (*frees pat*)

using ‹(pat, rhs) ∈ *set cs*›
by (*auto simp*: *Sterm.closed-except-simps list-all-iff*)
next
show *closed-env* (*fmmap value-to-sterm* Γ' $++_f$ *fmmap value-to-sterm venv*)
using ‹*closed-venv* Γ'› ‹*closed-venv venv*›
by (*auto intro*: *vclosed.value-to-sterm-env*)
next
show *frees pat* |⊆| *fmdom* (*fmmap value-to-sterm* Γ' $++_f$ *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* (λ(*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
— Basically a verbatim copy from the *comb* case

case (*rec-comb* Γ *t css name* Γ' *cs u* u' *venv pat rhs val*)

hence *vmatch* (*mk-pat pat*) u' = *Some venv*
by (*auto dest*: *vfind-match-elem*)

have *cs* = *the* (*fmlookup css name*)
using *rec-comb* **by** *simp*

have *wellformed t*
using *rec-comb* **by** *simp*
have *vwellformed* (*Vrecabs css name* Γ')
by (*rule veval-wellformed*) *fact*+
hence *vwellformed* (*Vabs cs* Γ') — convenient hack: *cs* is not really part of a *Vabs*
using *rec-comb* **by** *auto*

hence
list-all (*linear* ∘ *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 *rec-comb* **by** (*simp add*: *closed-except-def*)+
have *vclosed* (*Vrecabs css name* Γ′)
by (*rule veval-closed*) *fact*+
hence *vclosed* (*Vabs cs* Γ′)
using *rec-comb* **by** *auto*

have *vconstructor-value* u'
by (*rule veval-constructor-value*) *fact*+
have *vconstructor-value* (*Vrecabs css name* Γ′)
by (*rule veval-constructor-value*) *fact*+
hence *vconstructor-value-env* Γ′
by *simp*
have *vconstructor-value-env venv*
by (*rule vconstructor-value.vmatch-env*) *fact*+

have *wellformed u*
using *rec-comb* **by** *simp*
have *vwellformed* u'
by (*rule veval-wellformed*) *fact*+
have *wellformed-venv venv*
by (*rule vwellformed.vmatch-env*) *fact*+

have *closed-except u* (*fmdom* Γ)
using *rec-comb* **by** (*simp add*: *closed-except-def*)

have *vclosed* u'
by (*rule veval-closed*) *fact*+
have *closed-venv venv*
by (*rule vclosed.vmatch-env*) *fact*+

have *closed-venv* Γ'
using ‹*vclosed* (*Vabs cs* Γ')› **by** *simp*

let *?subst* = λ*pat t. subst t* (*fmdrop-fset* (*frees pat*) (*fmmap value-to-sterm* Γ'))
have *rs*$'$, *fmmap value-to-sterm* (Γ' $++_f$ *venv*) $\vdash_s$ *?subst pat rhs* $\downarrow$ *value-to-sterm val*
proof (*rule ssubst-eval*)
show *rs*$'$, *fmmap value-to-sterm* (Γ' $++_f$ *venv*) $\vdash_s$ *rhs* $\downarrow$ *value-to-sterm val*
proof (*rule rec-comb*)
have *linear pat closed-except rhs* (*fmdom* Γ' |∪| *frees pat*)
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 pat*) 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*)
using ‹*wellformed-venv* Γ'› ‹*wellformed-venv venv*›
by *blast*
next
show *closed-venv* (Γ' $++_f$ *venv*)
using ‹*closed-venv* Γ'› ‹*closed-venv venv*›
by *blast*
next
show *vconstructor-value-env* (Γ' $++_f$ *venv*)
using ‹*vconstructor-value-env* Γ'› ‹*vconstructor-value-env venv*›
by *blast*
next
show *wellformed rhs*
using ‹(*pat*, *rhs*) ∈ *set cs*› ‹*vwellformed* (*Vabs cs* Γ')›
by (*fastforce simp*: *list-all-iff*)
qed
next
have *fmdrop-fset* (*fmdom venv*) Γ' $\subseteq_f$ Γ' $++_f$ *venv*
including *fmap.lifting* **and** *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* (*frees pat*) (*fmmap value-to-sterm* Γ') $\subseteq_f$ *fmmap value-to-sterm* (Γ' $++_f$ *venv*)
unfolding ‹*fmdom venv* = *frees pat*›

by (*metis fmdrop-fset-fmmap fmmap-subset*)
next
show *closed-env* (*fmmap value-to-sterm* ($\Gamma' ++_f$ *venv*))
apply *auto*
apply *rule*
apply (*rule vclosed.value-to-sterm-env*, *fact*)+
done
next
show *value-env* (*fmmap value-to-sterm* ($\Gamma' ++_f$ *venv*))
apply *auto*
apply *rule*
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* ↓ *value-to-sterm* (*Vrecabs css name* Γ')
using *rec-comb* **by** (*auto simp*: *closed-except-def*)
thus *rs'*, *fmmap value-to-sterm* $\Gamma \vdash_s$ *t* ↓ *Sabs* (*map* (λ(*pat*, *t*). (*pat*, *?subst pat t*)) *cs*)
unfolding ‹*cs* = -› **by** *simp*
next
show *rs'*, *fmmap value-to-sterm* $\Gamma \vdash_s$ *u* ↓ *value-to-sterm u'*
using *rec-comb* **by** (*simp add*: *closed-except-def*)
next
show *rs'*, *fmmap value-to-sterm* $\Gamma ++_f$ *senv* $\vdash_s$ *?subst pat rhs* ↓ *value-to-sterm val*
proof (*rule seval-agree-eq*)
show *rs'*, *fmmap value-to-sterm* $\Gamma' ++_f$ *fmmap value-to-sterm venv* $\vdash_s$ *?subst pat rhs* ↓ *value-to-sterm val*
using ‹*rs'*, *fmmap value-to-sterm* ($\Gamma' ++_f$ *venv*) $\vdash_s$ *?subst pat rhs* ↓ *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* $\Gamma ++_f$ *senv*)
unfolding ‹*senv* = -›
apply (*subst* ‹*fmdom venv* = *frees pat*›[*symmetric*])+
apply (*subst fmdom-map*[*symmetric*])
apply (*subst fmadd-restrict-right-dom*)
apply (*subst fmdom-map*[*symmetric*])
apply (*subst fmadd-restrict-right-dom*)
apply *simp*
done
next
have *closed* (*value-to-sterm* (*Vabs cs* Γ'))
using ‹*vclosed* (*Vabs cs* Γ')›

```
          by (rule vclosed.value-to-sterm)
        thus closed-except (subst rhs (fmdrop-fset (frees pat) (fmmap value-to-sterm
Γ'))) (frees pat)
          using ‹(pat, rhs) ∈ set cs›
          by (auto simp: Sterm.closed-except-simps list-all-iff)
      next
          show closed-env (fmmap value-to-sterm Γ' ++f fmmap value-to-sterm
venv)
          using ‹closed-venv Γ'› ‹closed-venv venv›
          by (auto intro: vclosed.value-to-sterm-env)
      next
            show frees pat |⊆| fmdom (fmmap value-to-sterm Γ' ++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 (λ(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

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* (λ*name cs. Vrecabs css name* Γ′) *css*

context *special-constants* **begin**

inductive *veval′* :: (*name*, *value*) *fmap* ⇒ *sterm* ⇒ *value* ⇒ *bool* (‹-/ $\vdash_v$/ - ↓/ -› [*50*,*0*,*50*] *50*) **where**
const: *name* |∉| *C* ⟹ *fmlookup* Γ *name* = *Some val* ⟹ Γ $\vdash_v$ *Sconst name* ↓ *val*
|
var: *fmlookup* Γ *name* = *Some val* ⟹ Γ $\vdash_v$ *Svar name* ↓ *val* |
abs: Γ $\vdash_v$ *Sabs cs* ↓ *Vabs cs* Γ |
comb:
  Γ $\vdash_v$ *t* ↓ *Vabs cs* Γ′ ⟹ Γ $\vdash_v$ *u* ↓ *u′* ⟹
  vfind-match cs u′ = *Some* (*env*, -, *rhs*) ⟹
  Γ′ $++_f$ *env* $\vdash_v$ *rhs* ↓ *val* ⟹
  Γ $\vdash_v$ *t* $\$_s$ *u* ↓ *val* |
rec-comb:
  Γ $\vdash_v$ *t* ↓ *Vrecabs css name* Γ′ ⟹
  fmlookup css name = *Some cs* ⟹
  Γ $\vdash_v$ *u* ↓ *u′* ⟹
  vfind-match cs u′ = *Some* (*env*, -, *rhs*) ⟹
  Γ′ $++_f$ *mk-rec-env css* Γ′ $++_f$ *env* $\vdash_v$ *rhs* ↓ *val* ⟹
  Γ $\vdash_v$ *t* $\$_s$ *u* ↓ *val* |
constr: *name* |∈| *C* ⟹ *list-all2* (*veval′* Γ) *ts us* ⟹ Γ $\vdash_v$ *name* $$ *ts* ↓ *Vconstr name us*

lemma *veval′-sabs-svarE*:
  assumes Γ $\vdash_v$ *Sabs cs* $\$_s$ *Svar n* ↓ *v*
  obtains *u′ env pat rhs*
    where *fmlookup* Γ *n* = *Some u′*
      vfind-match cs u′ = *Some* (*env*, *pat*, *rhs*)
      Γ $++_f$ *env* $\vdash_v$ *rhs* ↓ *v*
using *assms* **proof** *cases*
  case (*constr name ts*)
  hence *strip-comb* (*Sabs cs* $\$_s$ *Svar n*) = *strip-comb* (*name* $$ *ts*)
    by *simp*

```
  hence False
    apply (fold app-sterm-def)
    apply (simp add: strip-list-comb-const)
    apply (simp add: const-sterm-def)
    done
  thus ?thesis by simp
next
  case rec-comb
  hence False by cases
  thus ?thesis by simp
next
  case (comb cs' Γ' u' env pat rhs)
  moreover have fmlookup Γ n = Some u'
    using ‹Γ ⊢_v Svar n ↓ u'›
    proof cases
      case (constr name ts)
      hence False
        by (fold free-sterm-def) simp
      thus ?thesis by simp
    qed auto
  moreover have cs = cs' Γ = Γ'
    using ‹Γ ⊢_v Sabs cs ↓ Vabs cs' Γ'›
    by (cases; auto)+

  ultimately show ?thesis
    using that by auto
qed

lemma veval'-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: vwell-
formed.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 wellformed-venv (Γ' ++_f mk-rec-env css Γ' ++_f env)
        proof (intro fmpred-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.vmatch-env)
        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 (- $$ -)›
    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 ↓ 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 fmpred-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 Γ ⊢_v t ↓ v closed-except t (fmdom Γ) closed-venv Γ
  assumes wellformed t wellformed-venv Γ
  shows vclosed v
using assms proof induction
  case (comb Γ t cs Γ′ u u′ env pat rhs val)
  hence vclosed (Vabs cs Γ′)
    by (auto simp: closed-except-def)

  have (pat, rhs) ∈ set cs vmatch (mk-pat pat) u′ = Some env
    by (rule vfind-match-elem; fact)+
  hence fmdom env = patvars (mk-pat pat)
    by (simp add: vmatch-dom)

  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 fmdom env = frees pat
    unfolding ‹fmdom env = -›
    by (simp add: mk-pat-frees)

  show ?case
    proof (rule comb)
      show wellformed rhs
```

using ‹(*pat*, *rhs*) ∈ *set cs*› ‹*vwellformed* (*Vabs cs* Γ′)›
by (*auto simp*: *list-all-iff*)
next
show *closed-venv* (Γ′ $++_f$ *env*)
apply *rule*
using ‹*vclosed* (*Vabs cs* Γ′)› **apply** *auto*[]
apply (*rule vclosed.vmatch-env*)
apply (*rule vfind-match-elem*)
using *comb* **by** (*auto simp*: *closed-except-def*)
next
show *closed-except rhs* (*fmdom* (Γ′ $++_f$ *env*))
using ‹*vclosed* (*Vabs cs* Γ′)› ‹*fmdom env* = *frees pat*› ‹(*pat*, *rhs*) ∈ *set cs*›
by (*auto simp*: *list-all-iff*)
next
show *wellformed-venv* (Γ′ $++_f$ *env*)
apply *rule*
using ‹*vwellformed* (*Vabs cs* Γ′)› **apply** *auto*[]
apply (*rule vwellformed.vmatch-env*)
apply (*rule vfind-match-elem*)
apply *fact*
apply (*rule veval′-wellformed*)
using *comb* **by** *auto*
qed
next
case (*rec-comb* Γ *t css name* Γ′ *cs u u′ env pat rhs val*)
have (*pat*, *rhs*) ∈ *set cs vmatch* (*mk-pat pat*) *u′* = *Some env*
by (*rule vfind-match-elem*; *fact*)+
hence *fmdom env* = *patvars* (*mk-pat pat*)
by (*simp add*: *vmatch-dom*)

have *vwellformed* (*Vrecabs css name* Γ′)
apply (*rule veval′-wellformed*)
using *rec-comb* **by** *auto*
hence *wellformed-clauses cs*
using *rec-comb* **by** *auto*
hence *linear pat*
using ‹(*pat*, *rhs*) ∈ *set cs*›
by (*auto simp*: *list-all-iff*)
hence *fmdom env* = *frees pat*
unfolding ‹*fmdom env* = -›
by (*simp add*: *mk-pat-frees*)
show *?case*
proof (*rule rec-comb*)
show *closed-venv* (Γ′ $++_f$ *mk-rec-env css* Γ′ $++_f$ *env*)
proof (*intro fmpred-add*)
show *closed-venv* Γ′
using *rec-comb* **by** (*auto simp*: *closed-except-def*)
next
show *closed-venv env*

using *rec-comb* **by** (*auto simp*: *closed-except-def dest*: *vfind-match-elem* *intro*: *vclosed.vmatch-env*)
next
show *closed-venv* (*mk-rec-env css* Γ')
unfolding *mk-rec-env-def*
using *rec-comb* **by** (*auto simp*: *closed-except-def intro*: *fmdomI*)
qed
next
have *vclosed* (*Vrecabs css name* Γ')
using *mk-rec-env-def*
using *rec-comb* **by** (*auto simp*: *closed-except-def intro*: *fmdom'I*)
hence *closed-except rhs* (*fmdom* Γ' |∪| *frees pat*)
apply *simp*
apply (*elim conjE*)
apply (*drule fmpredD*[**where** m = *css*])
apply (*rule rec-comb*)
using ‹(*pat*, *rhs*) ∈ *set cs*›
unfolding *list-all-iff* **by** *auto*

thus *closed-except rhs* (*fmdom* (Γ' $++_f$ *mk-rec-env css* Γ' $++_f$ *env*))
unfolding *closed-except-def*
using ‹*fmdom env* = *frees pat*›
by *auto*
next
show *wellformed rhs*
using ‹*wellformed-clauses cs*› ‹(*pat*, *rhs*) ∈ *set cs*›
by (*auto simp*: *list-all-iff*)
next
show *wellformed-venv* (Γ' $++_f$ *mk-rec-env css* Γ' $++_f$ *env*)
proof (*intro fmpred-add*)
show *wellformed-venv* Γ'
using ‹*vwellformed* (*Vrecabs css name* Γ')› **by** *auto*
next
show *wellformed-venv env*
using *rec-comb* **by** (*auto dest*: *vfind-match-elem intro*: *veval'-wellformed* *vwellformed.vmatch-env*)
next
show *wellformed-venv* (*mk-rec-env css* Γ')
unfolding *mk-rec-env-def*
using ‹*vwellformed* (*Vrecabs css name* Γ')› **by** (*auto intro*: *fmdomI*)
qed
qed
next
case (*constr name* Γ *ts us*)
have *list-all vclosed us*
using ‹*list-all2* - - -› ‹*closed-except* (- $$ -) -› ‹*wellformed* (- $$ -)›
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 ⇒ bool where
vwelldefined' (Vconstr name vs) ⟷ list-all vwelldefined' vs |
vwelldefined' (Vabs cs Γ) ⟷
  pred-fmap id (fmmap vwelldefined' Γ) ∧
  list-all (λ(pat, t). consts t |⊆| (fmdom Γ |∪| C)) cs ∧
  fdisjnt C (fmdom Γ) |
vwelldefined' (Vrecabs css name Γ) ⟷
  pred-fmap id (fmmap vwelldefined' Γ) ∧
  pred-fmap (λcs.
    list-all (λ(pat, t). consts t |⊆| fmdom Γ |∪| (C |∪| fmdom css)) cs ∧
    fdisjnt C (fmdom Γ)) css ∧
  name |∈| fmdom css ∧
  fdisjnt C (fmdom css)

lemma vmatch-welldefined':
  assumes vmatch pat v = Some env vwelldefined' v
  shows fmpred (λ-. vwelldefined') env
using assms proof (induction pat v arbitrary: env rule: vmatch-induct)
  case (constr name ps name' vs)
  hence
    map-option (foldl (++_f) fmempty) (those (map2 vmatch ps vs)) = Some env
    name = name' length ps = length vs
    by (auto split: if-splits)
  then obtain envs where env = foldl (++_f) fmempty envs map2 vmatch ps vs
= map Some envs
    by (blast dest: those-someD)

  moreover have fmpred (λ-. vwelldefined') env if env ∈ set envs for env
    proof −
      from that have 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)
      hence vwelldefined' v
        using constr by (simp add: list-all-iff)
      show ?thesis
        by (rule constr; safe?) fact+
    qed

    ultimately show ?case
      by auto
qed auto
```

```
lemma sconsts-list-comb:
  consts (list-comb f xs) |⊆| S ⟷ consts f |⊆| S ∧ list-all (λx. consts x |⊆| S) xs
by (induction xs arbitrary: f) auto

lemma sconsts-sabs:
  consts (Sabs cs) |⊆| S ⟷ list-all (λ(-, t). consts t |⊆| S) 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 (in constants) veval'-welldefined':
  assumes Γ ⊢_v t ↓ v fdisjnt C (fmdom Γ)
  assumes consts t |⊆| fmdom Γ |∪| C fmpred (λ-. vwelldefined') Γ
  assumes wellformed t wellformed-venv Γ
  assumes ¬ shadows-consts t not-shadows-vconsts-env Γ
  shows vwelldefined' v
using assms proof induction
  case (abs Γ cs)
  thus ?case
    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 fmdom 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'
      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 fdisjnt C (fmdom env)
      unfolding ‹fmdom env = frees pat›
      using ‹(pat, rhs) ∈ set cs› ‹not-shadows-vconsts (Vabs cs Γ')›
      by (auto simp: list-all-iff all-consts-def fdisjnt-alt-def)
    ultimately show fdisjnt C (fmdom (Γ' ++_f env))
      unfolding fdisjnt-alt-def by auto
  next
    show wellformed rhs
      using ‹(pat, rhs) ∈ set cs› ‹vwellformed (Vabs cs Γ')›
      by (auto simp: list-all-iff)
  next
    have wellformed-venv Γ'
      using ‹vwellformed (Vabs cs Γ')› 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 (Γ' ++_f env)
      by blast
  next
    have not-shadows-vconsts-env Γ'
      using ‹not-shadows-vconsts (Vabs cs Γ')› 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* (Γ' $++_f$ *env*)
by *blast*
next
show ¬ *shadows-consts rhs*
using ‹*not-shadows-vconsts* (*Vabs cs* Γ')› ‹(*pat*, *rhs*) ∈ *set cs*›
by (*auto simp*: *list-all-iff*)
qed
next
case (*rec-comb* Γ *t css name* Γ' *cs u* u' *env pat rhs val*)
hence (*pat*, *rhs*) ∈ *set cs*
by (*auto dest*: *vfind-match-elem*)
moreover have *vwelldefined′* (*Vrecabs css name* Γ')
using *rec-comb* **by** *auto*
ultimately have *consts rhs* |⊆| *fmdom* Γ' |∪| (*C* |∪| *fmdom css*)
using ‹*fmlookup css name* = *Some cs*›
by (*auto simp*: *list-all-iff dest*!: *fmpredD*[**where** *m* = *css*])

have *vwellformed* (*Vrecabs css name* Γ')
apply (*rule veval′-wellformed*)
using *rec-comb* **by** *auto*
hence *wellformed-clauses cs*
using *rec-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 *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* Γ')
apply (*rule veval′-shadows*)
using *rec-comb* **by** *auto*

have *vwelldefined′* (*Vrecabs css name* Γ')
using *rec-comb* **by** *auto*

show *?case*
proof (*rule rec-comb*)
show *consts rhs* |⊆| *fmdom* (Γ' $++_f$ *mk-rec-env css* Γ' $++_f$ *env*) |∪| *C*
using ‹*consts rhs* |⊆| -› **unfolding** *mk-rec-env-def*

by *auto*
next
have *fmpred* (λ-. *vwelldefined'*) Γ'
using ‹*vwelldefined'* (*Vrecabs css name* Γ')› **by** *auto*
moreover have *fmpred* (λ-. *vwelldefined'*) (*mk-rec-env css* Γ')
unfolding *mk-rec-env-def*
using *rec-comb* **by** (*auto intro*: *fmdomI*)
moreover have *fmpred* (λ-. *vwelldefined'*) *env*
using *rec-comb* **by** (*auto dest*: *vfind-match-elem intro*: *vmatch-welldefined'*)
ultimately show *fmpred* (λ-. *vwelldefined'*) ($\Gamma' ++_f$ *mk-rec-env css* $\Gamma' ++_f$ *env*)
by *blast*
next
have *fdisjnt C* (*fmdom* Γ')
using *rec-comb* **by** *auto*
moreover have *fdisjnt C* (*fmdom env*)
unfolding ‹*fmdom env* = *frees pat*›
using ‹*fmlookup css name* = *Some cs*› ‹(*pat*, *rhs*) ∈ *set cs*› ‹*not-shadows-vconsts* (*Vrecabs css name* Γ')›
apply *auto*
apply (*drule fmpredD*[**where** *m* = *css*])
by (*auto simp*: *list-all-iff all-consts-def fdisjnt-alt-def*)
moreover have *fdisjnt C* (*fmdom* (*mk-rec-env css* Γ'))
unfolding *mk-rec-env-def*
using ‹*vwelldefined'* (*Vrecabs css name* Γ')›
by *simp*
ultimately show *fdisjnt C* (*fmdom* ($\Gamma' ++_f$ *mk-rec-env css* $\Gamma' ++_f$ *env*))
unfolding *fdisjnt-alt-def* **by** *auto*
next
show *wellformed rhs*
using ‹(*pat*, *rhs*) ∈ *set cs*› ‹*wellformed-clauses cs*›
by (*auto simp*: *list-all-iff*)
next
have *wellformed-venv* Γ'
using ‹*vwellformed* (*Vrecabs css name* Γ')› **by** *simp*
moreover have *wellformed-venv* (*mk-rec-env css* Γ')
unfolding *mk-rec-env-def*
using ‹*vwellformed* (*Vrecabs css name* Γ')›
by (*auto intro*: *fmdomI*)
moreover have *wellformed-venv env*
apply (*rule vwellformed.vmatch-env*)
apply (*rule vfind-match-elem*)
apply *fact*
apply (*rule veval'-wellformed*)
using *rec-comb* **by** *auto*
ultimately show *wellformed-venv* ($\Gamma' ++_f$ *mk-rec-env css* $\Gamma' ++_f$ *env*)
by *blast*
next
have *not-shadows-vconsts-env* Γ'

using ‹*not-shadows-vconsts* (*Vrecabs css name* Γ′)› **by** *simp*
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*)
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*
ultimately show *not-shadows-vconsts-env* (Γ′ $++_f$ *mk-rec-env css* Γ′ $++_f$ *env*)
by *blast*
next
show ¬ *shadows-consts rhs*
using *rec-comb* ‹*not-shadows-vconsts* (*Vrecabs css name* Γ′)› ‹(*pat*, *rhs*) ∈ *set cs*›
by (*auto simp*: *list-all-iff*)
qed
next
case (*constr name* Γ *ts us*)
have *list-all vwelldefined′ us*
using ‹*list-all2* - - -› ‹*consts* (*name* $$ *ts*) |⊆| -›
using ‹*wellformed* (*name* $$ *ts*)› ‹¬ *shadows-consts* (*name* $$ *ts*)›
proof (*induction ts us rule*: *list.rel-induct*)
case (*Cons v vs u 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* (‹$\vdash_v$/ - ≈ -› [*0*, *50*] *50*) **where**
constr: *list-all2 vrelated ts us* ⟹ $\vdash_v$ *Vconstr name ts* ≈ *Vconstr name us* |
abs:
  fmrel-on-fset (*frees* (*Sabs cs*)) *vrelated* Γ_1 Γ_2 ⟹
   fmrel-on-fset (*consts* (*Sabs cs*)) *vrelated* (*fmap-of-list rs*) Γ_2 ⟹
    $\vdash_v$ *Vabs cs* Γ_1 ≈ *Vabs cs* Γ_2 |
rec-abs:
  pred-fmap (λ*cs*.
    fmrel-on-fset (*frees* (*Sabs cs*)) *vrelated* Γ_1 Γ_2 ∧
    fmrel-on-fset (*consts* (*Sabs cs*)) *vrelated* (*fmap-of-list rs*) (Γ_2 $++_f$ *mk-rec-env css* Γ_2)) *css* ⟹
     name |∈| *fmdom css* ⟹
    $\vdash_v$ *Vrecabs css name* Γ_1 ≈ *Vrecabs css name* Γ_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 $\vdash_v$ t_1 ≈ 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* ≈ *Vconstr name′ us* ⟷ (*name* = *name′* ∧ *list-all2 vrelated ts us*)
    proof *safe*
      assume $\vdash_v$ *Vconstr name ts* ≈ *Vconstr name′ us*
      thus *name* = *name′ 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* Γ_2
  assumes ¬ *shadows-consts t not-shadows-vconsts-env* Γ_2

assumes *welldefined t*
assumes *fmpred* (λ-. *vwelldefined*) Γ_1
assumes *fmrel-on-fset* (*frees t*) *vrelated* Γ_1 Γ_2
assumes *fmrel-on-fset* (*consts t*) *vrelated* (*fmap-of-list rs*) Γ_2
obtains v_1 **where** $rs, \Gamma_1 \vdash_v t \downarrow v_1 \vdash_v v_1 \approx v_2$
apply *atomize-elim*
using *assms* **proof** (*induction arbitrary*: Γ_1)
case (*const name* Γ_2 val_2)
hence *fmrel-on-fset* {|*name*|} *vrelated* (*fmap-of-list rs*) Γ_2
by *simp*
have *rel-option vrelated* (*fmlookup* (*fmap-of-list rs*) *name*) (*fmlookup* Γ_2 *name*)
apply (*rule fmrel-on-fsetD*[**where** S = {|*name*|}])
apply *simp*
apply *fact*
done
then obtain val_1 **where** *fmlookup* (*fmap-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*: *fmap-of-list-SomeD*)

show *?case*
apply (*intro conjI exI*)
apply (*rule veval.const*)
apply *fact+*
done
next
case (*var* Γ_2 *name* val_2)
hence *fmrel-on-fset* {|*name*|} *vrelated* Γ_1 Γ_2
by *simp*
have *rel-option vrelated* (*fmlookup* Γ_1 *name*) (*fmlookup* Γ_2 *name*)
apply (*rule fmrel-on-fsetD*[**where** S = {|*name*|}])
apply *simp*
apply *fact*
done
then obtain val_1 **where** *fmlookup* Γ_1 *name* = *Some* $val_1 \vdash_v val_1 \approx val_2$
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* Γ_2 *t cs* Γ'_2 *u* u'_2 env_2 *pat rhs* val_2)
hence $\exists v.\ rs, \Gamma_1 \vdash_v t \downarrow v \wedge \vdash_v v \approx$ *Vabs cs* Γ'_2

by (*auto intro*: *fmrel-on-fsubset*)
then obtain v **where** $\vdash_v v \approx$ *Vabs cs* Γ'_2 *rs*, $\Gamma_1 \vdash_v t \downarrow v$
by *blast*
moreover then obtain Γ'_1
where $v =$ *Vabs cs* Γ'_1
and *fmrel-on-fset* (*frees* (*Sabs cs*)) *vrelated* Γ'_1 Γ'_2
and *fmrel-on-fset* (*consts* (*Sabs cs*)) *vrelated* (*fmap-of-list rs*) Γ'_2
by *cases auto*
ultimately have *rs*, $\Gamma_1 \vdash_v t \downarrow$ *Vabs cs* Γ'_1
by *simp*

have $\exists\, u_1'.$ *rs*, $\Gamma_1 \vdash_v u \downarrow u_1' \wedge \vdash_v u_1' \approx u'_2$
using *comb* **by** (*auto intro*: *fmrel-on-fsubset*)
then obtain u'_1 **where** $\vdash_v u'_1 \approx u'_2$ *rs*, $\Gamma_1 \vdash_v u \downarrow u'_1$
by *blast*

have *rel-option* (*rel-prod* (*fmrel vrelated*) (=)) (*vfind-match cs* u'_1) (*vfind-match cs* u'_2)
by (*rule vrelated.vfind-match-rel'*) *fact*
then obtain env_1 **where** *vfind-match cs* u'_1 = *Some* (env_1, *pat*, *rhs*) *fmrel vrelated* env_1 env_2
using ‹*vfind-match cs* u'_2 = -›
by *cases auto*

have (*pat*, *rhs*) $\in$ *set cs*
by (*rule vfind-match-elem*) *fact*

have *vwellformed* (*Vabs cs* Γ'_2)
apply (*rule veval'-wellformed*)
apply *fact*
using ‹*wellformed* (t $\$_s$ u)› **apply** *simp*
apply *fact+*
done
hence *wellformed-venv* Γ'_2
by *simp*

have *vwelldefined v*
apply (*rule veval-welldefined*)
apply *fact+*
using *comb* **by** *simp*
hence *vwelldefined* (*Vabs cs* Γ'_1)
unfolding ‹v = -› **.**

have *linear pat*
using ‹(*pat*, *rhs*) $\in$ *set cs*› ‹*vwellformed* (*Vabs cs* Γ'_2)›
by (*auto simp*: *list-all-iff*)

have *fmdom* env_1 = *patvars* (*mk-pat pat*)
apply (*rule vmatch-dom*)

apply (*rule vfind-match-elem*)
apply *fact*
done
with ‹*linear pat*› **have** *fmdom* env_1 = *frees pat*
by (*simp add*: *mk-pat-frees*)

have *fmdom* env_2 = *patvars* (*mk-pat pat*)
apply (*rule vmatch-dom*)
apply (*rule vfind-match-elem*)
apply *fact*
done
with ‹*linear pat*› **have** *fmdom* env_2 = *frees pat*
by (*simp add*: *mk-pat-frees*)

note *fset-of-list-map*[*simp del*]
have $\exists\, val_1.\ rs,\ \Gamma'_1 \mathbin{++_f} env_1 \vdash_v rhs \downarrow val_1 \land \vdash_v val_1 \approx val_2$
proof (*rule comb*)
show *fmrel-on-fset* (*frees rhs*) *vrelated* $(\Gamma'_1 \mathbin{++_f} env_1)$ $(\Gamma'_2 \mathbin{++_f} env_2)$
proof
fix *name*
assume *name* |∈| *frees rhs*
show *rel-option vrelated* (*fmlookup* $(\Gamma'_1 \mathbin{++_f} env_1)$ *name*) (*fmlookup* $(\Gamma'_2$ $\mathbin{++_f} env_2)$ *name*)
proof (*cases name* |∈| *frees pat*)
case *True*
hence *name* |∈| *fmdom* env_1 *name* |∈| *fmdom* env_2
using ‹*fmdom* env_1 = *frees pat*› ‹*fmdom* env_2 = *frees pat*›
by *simp+*
hence *fmlookup* $(\Gamma'_1 \mathbin{++_f} env_1)$ *name* = *fmlookup* env_1 *name fmlookup* $(\Gamma'_2 \mathbin{++_f} env_2)$ *name* = *fmlookup* env_2 *name*
by *auto*
thus *?thesis*
using ‹*fmrel vrelated* env_1 env_2›
by *auto*
next
case *False*
hence *name* |∉| *fmdom* env_1 *name* |∉| *fmdom* env_2
using ‹*fmdom* env_1 = *frees pat*› ‹*fmdom* env_2 = *frees pat*›
by *simp+*
hence *fmlookup* $(\Gamma'_1 \mathbin{++_f} env_1)$ *name* = *fmlookup* Γ'_1 *name fmlookup* $(\Gamma'_2 \mathbin{++_f} env_2)$ *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 env_2)
      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 env_2)
      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-elem ffUnion-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 env_2
          proof
            assume name |∈| fmdom env_2
            hence fmlookup env_2 name ≠ None
              by (meson fmdom-notI)
            then obtain v where fmlookup env_2 name = Some v
              by blast
            hence name |∈| fmdom env_2
              by (auto intro: fmdomI)
            hence name |∈| frees pat
              using ‹fmdom env_2 = frees pat›
              by simp
```

```
          have welldefined rhs
            using ‹vwelldefined (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 ‹(pat, rhs) ∈ set cs› ‹vwellformed (Vabs cs Γ'₂)›
      by (auto simp: list-all-iff)
  next
    have wellformed-venv Γ'₂
      by fact
    moreover have wellformed-venv env₂
      apply (rule vwellformed.vmatch-env)
       apply (rule vfind-match-elem)
       apply fact
      apply (rule veval'-wellformed)
        apply fact
      using ‹wellformed (t $_s u)› apply simp
      apply fact+
      done
    ultimately show wellformed-venv (Γ'₂ ++_f env₂)
      by blast
  next
    show welldefined rhs
      using ‹vwelldefined (Vabs cs Γ'₁)› ‹(pat, rhs) ∈ set cs›
      by (auto simp: list-all-iff)
  next
    have fmpred (λ-. vwelldefined) Γ'₁
      using ‹vwelldefined (Vabs cs Γ'₁)› by simp

    moreover have fmpred (λ-. vwelldefined) env₁
      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* (λ-. *vwelldefined*) (Γ'_1 $++_f$ env_1)
by *blast*
qed

then obtain val_1 **where** *rs*, Γ'_1 $++_f$ env_1 $\vdash_v$ *rhs* $\downarrow$ val_1 $\vdash_v$ $val_1 \approx val_2$
by *blast*

show *?case*
apply (*intro conjI exI*)
apply (*rule veval.comb*)
apply *fact+*
done
next
— Almost verbatim copy from *comb* case.
case (*rec-comb* Γ_2 *t css name* Γ'_2 *cs u* u'_2 env_2 *pat rhs* val_2)
hence $\exists\, v.$ *rs*, $\Gamma_1 \vdash_v t \downarrow v \wedge \vdash_v v \approx$ *Vrecabs css name* Γ'_2
by (*auto intro*: *fmrel-on-fsubset*)
then obtain v **where** $\vdash_v v \approx$ *Vrecabs css name* Γ'_2 *rs*, $\Gamma_1 \vdash_v t \downarrow v$
by *blast*
moreover then obtain Γ'_1
where v = *Vrecabs css name* Γ'_1
and *fmrel-on-fset* (*frees* (*Sabs cs*)) *vrelated* Γ'_1 Γ'_2
and *fmrel-on-fset* (*consts* (*Sabs cs*)) *vrelated* (*fmap-of-list rs*) (Γ'_2 $++_f$ *mk-rec-env css* Γ'_2)
using ‹*fmlookup css name* = *Some cs*›
by *cases auto*
ultimately have *rs*, $\Gamma_1 \vdash_v t \downarrow$ *Vrecabs css name* Γ'_1
by *simp*

have $\exists\, u_1'.$ *rs*, $\Gamma_1 \vdash_v u \downarrow u_1' \wedge \vdash_v u_1' \approx u'_2$
using *rec-comb* **by** (*auto intro*: *fmrel-on-fsubset*)
then obtain u'_1 **where** $\vdash_v u'_1 \approx u'_2$ *rs*, $\Gamma_1 \vdash_v u \downarrow u'_1$
by *blast*

have *rel-option* (*rel-prod* (*fmrel vrelated*) (=)) (*vfind-match cs* u'_1) (*vfind-match cs* u'_2)
by (*rule vrelated.vfind-match-rel'*) *fact*
then obtain env_1 **where** *vfind-match cs* u'_1 = *Some* (env_1, *pat*, *rhs*) *fmrel vrelated* env_1 env_2
using ‹*vfind-match cs* u'_2 = -›
by *cases auto*

have (*pat*, *rhs*) $\in$ *set cs*

by (*rule vfind-match-elem*) *fact*

have *vwellformed* (*Vrecabs css name* Γ'_2)
apply (*rule veval′-wellformed*)
apply *fact*
using ‹*wellformed* (t $\$_s$ u)› **apply** *simp*
apply *fact+*
done
hence *wellformed-venv* Γ'_2 *vwellformed* (*Vabs cs* Γ'_2)
using *rec-comb* **by** *auto*

have *vwelldefined v*
apply (*rule veval-welldefined*)
apply *fact+*
using *rec-comb* **by** *simp*
hence *vwelldefined* (*Vrecabs css name* Γ'_1)
unfolding ‹v = -› **.**
hence *vwelldefined* (*Vabs cs* Γ'_1)
using *rec-comb* **by** *auto*

have *linear pat*
using ‹(*pat*, *rhs*) ∈ *set cs*› ‹*vwellformed* (*Vabs cs* Γ'_2)›
by (*auto simp*: *list-all-iff*)

have *fmdom* env_1 = *patvars* (*mk-pat pat*)
apply (*rule vmatch-dom*)
apply (*rule vfind-match-elem*)
apply *fact*
done
with ‹*linear pat*› **have** *fmdom* env_1 = *frees pat*
by (*simp add*: *mk-pat-frees*)

have *fmdom* env_2 = *patvars* (*mk-pat pat*)
apply (*rule vmatch-dom*)
apply (*rule vfind-match-elem*)
apply *fact*
done
with ‹*linear pat*› **have** *fmdom* env_2 = *frees pat*
by (*simp add*: *mk-pat-frees*)

note *fset-of-list-map*[*simp del*]
have $\exists\, val_1.\ rs, \Gamma'_1 +\!+_f env_1 \vdash_v rhs \downarrow val_1 \wedge \vdash_v val_1 \approx val_2$
proof (*rule rec-comb*)
have *not-shadows-vconsts* (*Vrecabs css name* Γ'_2)
apply (*rule veval′-shadows*)
apply *fact+*
using *rec-comb* **by** *auto*
thus ¬ *shadows-consts rhs*
using ‹(*pat*, *rhs*) ∈ *set cs*› *rec-comb*

by (*auto simp*: *list-all-iff*)
hence *fdisjnt all-consts* (*frees rhs*)
by (*rule shadows-consts-frees*)

have *not-shadows-vconsts-env* Γ'_2
using ‹*not-shadows-vconsts* (*Vrecabs css name* Γ'_2)›
by *simp*

moreover have *not-shadows-vconsts-env* (*mk-rec-env css* Γ'_2)
unfolding *mk-rec-env-def*
using ‹*not-shadows-vconsts* (*Vrecabs css name* Γ'_2)›
by (*auto intro*: *fmdomI*)

moreover have *not-shadows-vconsts-env* env_2
apply (*rule not-shadows-vconsts.vmatch-env*)
apply (*rule vfind-match-elem*)
apply *fact*
apply (*rule veval'-shadows*)
apply *fact*
apply *fact*
using *rec-comb* **by** *auto*

ultimately show *not-shadows-vconsts-env* (Γ'_2 $++_f$ *mk-rec-env css* Γ'_2 $++_f$ env_2)
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 $++_f$ env_1) (Γ'_2 $++_f$ *mk-rec-env css* Γ'_2 $++_f$ env_2)
proof
fix *name*
assume *name* $|\in|$ *frees rhs*
moreover have *fmdom css* $|\subseteq|$ *all-consts*
using ‹*vwelldefined* (*Vrecabs* - - -)› **unfolding** *all-consts-def*
by *auto*
ultimately have *name* $|\notin|$ *fmdom css*
using ‹*fdisjnt* - (*frees rhs*)›
unfolding *fdisjnt-alt-def*
by (*metis* (*full-types*) *fempty-iff finterI fset-rev-mp*)

show *rel-option vrelated* (*fmlookup* (Γ'_1 $++_f$ env_1) *name*) (*fmlookup* (Γ'_2 $++_f$ *mk-rec-env css* Γ'_2 $++_f$ env_2) *name*)
proof (*cases name* $|\in|$ *frees pat*)
case *True*
hence *name* $|\in|$ *fmdom* env_1 *name* $|\in|$ *fmdom* env_2
using ‹*fmdom* env_1 = *frees pat*› ‹*fmdom* env_2 = *frees pat*›

by *simp+*
hence
fmlookup $(\Gamma'_1 ++_f\ env_1)$ *name* = *fmlookup* env_1 *name*
fmlookup $(\Gamma'_2 ++_f$ *mk-rec-env css* $\Gamma'_2 ++_f\ env_2)$ *name* = *fmlookup* env_2 *name*
by *auto*
thus *?thesis*
using ‹*fmrel vrelated* env_1 env_2›
by *auto*
next
case *False*
hence *name* $|\notin|$ *fmdom* env_1 *name* $|\notin|$ *fmdom* env_2
using ‹*fmdom* env_1 = *frees pat*› ‹*fmdom* env_2 = *frees pat*›
by *simp+*
hence
fmlookup $(\Gamma'_1 ++_f\ env_1)$ *name* = *fmlookup* Γ'_1 *name*
fmlookup $(\Gamma'_2 ++_f$ *mk-rec-env css* $\Gamma'_2 ++_f\ env_2)$ *name* = *fmlookup* Γ'_2 *name*
unfolding *mk-rec-env-def* **using** ‹*name* $|\notin|$ *fmdom css*›
by *auto*

moreover have *name* $|\in|$ *frees* (*Sabs cs*)
using *False* ‹*name* $|\in|$ *frees rhs*› ‹(*pat*, *rhs*) $\in$ *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

show *fmrel-on-fset* (*consts rhs*) *vrelated* (*fmap-of-list rs*) $(\Gamma'_2 ++_f$ *mk-rec-env css* $\Gamma'_2 ++_f\ env_2)$
proof
fix *name*
assume *name* $|\in|$ *consts rhs*
hence *name* $|\in|$ *consts* (*Sabs cs*)
using ‹(*pat*, *rhs*) $\in$ *set cs*›
by (*auto intro*!: *fBexI simp*: *fset-of-list-elem ffUnion-alt-def*)
hence *rel-option vrelated* (*fmlookup* (*fmap-of-list rs*) *name*) (*fmlookup* $(\Gamma'_2 ++_f$ *mk-rec-env css* $\Gamma'_2)$ *name*)
using ‹*fmrel-on-fset* (*consts* (*Sabs cs*)) *vrelated* (*fmap-of-list rs*) $(\Gamma'_2 ++_f$ *mk-rec-env css* $\Gamma'_2)$›
by (*auto dest*: *fmrel-on-fsetD*)
moreover have *name* $|\notin|$ *fmdom* env_2
proof

assume *name* |∈| *fmdom* env_2
hence *fmlookup* env_2 *name* ≠ *None*
by (*meson fmdom-notI*)
then obtain *v* **where** *fmlookup* env_2 *name* = *Some v*
by *blast*
hence *name* |∈| *fmdom* env_2
by (*auto intro*: *fmdomI*)
hence *name* |∈| *frees pat*
using ‹*fmdom* env_2 = *frees pat*›
by *simp*

have *vwelldefined* (*Vabs cs* Γ'_1)
using ‹*vwelldefined* (*Vrecabs css* - Γ'_1)›
using *rec-comb*
by *auto*

hence *welldefined rhs*
using ‹(*pat*, *rhs*) ∈ *set cs*› *rec-comb*
by (*auto simp*: *list-all-iff*)
hence *name* |∈| *fst* |‘| *fset-of-list rs* |∪| *C*
using ‹*name* |∈| *consts rhs*› *all-consts-def*
by *blast*
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 $++_f$ *mk-rec-env css* Γ'_2 $++_f$ env_2) *name*)
by *simp*
qed
next
show *wellformed rhs*
using ‹(*pat*, *rhs*) ∈ *set cs*› ‹*vwellformed* (*Vabs cs* Γ'_2)›
by (*auto simp*: *list-all-iff*)
next
have *wellformed-venv* Γ'_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 ‹*wellformed* (*t* $\$_s$ *u*)› **apply** *simp*
apply *fact+*

done
moreover have *wellformed-venv* $(\mathit{mk\text{-}rec\text{-}env}\ \mathit{css}\ \Gamma'_2)$
unfolding *mk-rec-env-def*
using ‹*vwellformed* (*Vrecabs* - - -)›
by (*auto intro*: *fmdomI*)
ultimately show *wellformed-venv* $(\Gamma'_2 \mathbin{++_f} \mathit{mk\text{-}rec\text{-}env}\ \mathit{css}\ \Gamma'_2 \mathbin{++_f} \mathit{env}_2)$
by *blast*
next
show *welldefined rhs*
using ‹*vwelldefined* (*Vabs cs* Γ'_1)› ‹(*pat*, *rhs*) $\in$ *set cs*›
by (*auto simp*: *list-all-iff*)
next
have *fmpred* (λ-. *vwelldefined*) Γ'_1
using ‹*vwelldefined* (*Vabs cs* Γ'_1)› **by** *simp*

moreover have *fmpred* (λ-. *vwelldefined*) env_1
apply (*rule vwelldefined.vmatch-env*)
apply (*rule vfind-match-elem*)
apply *fact*
apply (*rule veval-welldefined*)
apply *fact*
apply *fact*
using *rec-comb* **apply** *simp*
done

ultimately show *fmpred* (λ-. *vwelldefined*) $(\Gamma'_1 \mathbin{++_f} \mathit{env}_1)$
by *blast*
qed

then obtain val_1 **where** $\mathit{rs}, \Gamma'_1 \mathbin{++_f} \mathit{env}_1 \vdash_v \mathit{rhs} \downarrow \mathit{val}_1 \vdash_v \mathit{val}_1 \approx \mathit{val}_2$
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* $(\lambda t\ u_2.\ (\exists\, u_1.\ \mathit{rs}, \Gamma_1 \vdash_v t \downarrow u_1 \wedge \vdash_v u_1 \approx u_2))\ \mathit{ts}\ \mathit{us}_2$
using ‹*list-all2* - *ts* us_2›
proof (*rule list.rel-mono-strong*, *elim conjE impE allE exE*)
fix $t\ u_2$
assume $t \in \mathit{set}\ \mathit{ts}\ u_2 \in \mathit{set}\ \mathit{us}_2$
assume $\Gamma_2 \vdash_v t \downarrow u_2$

show *wellformed t welldefined t* $\neg$ *shadows-consts t*
using *constr* ‹$t \in$ *set ts*›

```
      unfolding welldefined.list-comb wellformed.list-comb shadows.list-comb
      by (auto simp: list-all-iff list-ex-iff)

    show
      wellformed-venv Γ2
      not-shadows-vconsts-env Γ2
      fmpred (λ-. vwelldefined) Γ1
      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) Γ2
      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 fmrel-on-fset (frees t) vrelated Γ1 Γ2
      using constr by (blast intro: fmrel-on-fsubset)
  qed auto

  then obtain us1 where list-all2 (veval rs Γ1) ts us1 list-all2 vrelated us1 us2
    by induction auto

  thus ?case
    using constr
    by (auto intro: veval.constr vrelated.constr)
qed

lemma veval'-correct':
  assumes Γ2 ⊢v t ↓ v2 wellformed t wellformed-venv Γ2
  assumes ¬ shadows-consts t not-shadows-vconsts-env Γ2
  assumes welldefined t
  assumes closed t
  assumes fmrel-on-fset (consts t) vrelated (fmap-of-list rs) Γ2
  obtains v1 where rs, fmempty ⊢v t ↓ v1 ⊢v v1 ≈ v2
proof (rule veval'-correct[where Γ1 = fmempty])
  show fmpred (λ-. vwelldefined) fmempty by simp
next
  show fmrel-on-fset (frees t) vrelated fmempty Γ2
    using ‹closed t› unfolding closed-except-def by auto
qed (rule assms)+

end
```

Preservation of extensional equality

lemma (**in** *constants*) *veval′-agree-eq*:
  assumes $\Gamma \vdash_v t \downarrow v$ *fmrel-on-fset* (*ids t*) *erelated* Γ' Γ
  assumes *closed-venv* Γ *closed-except t* (*fmdom* Γ)
  assumes *wellformed t wellformed-venv* Γ *fdisjnt C* (*fmdom* Γ)
  assumes *consts t* $|\subseteq|$ *fmdom* Γ $|\cup|$ *C fmpred* (λ-. *vwelldefined′*) Γ
  assumes $\neg$ *shadows-consts t not-shadows-vconsts-env* Γ
  obtains v' **where** $\Gamma' \vdash_v t \downarrow v'$ $v' \approx_e v$
using *assms* **proof** (*induction arbitrary*: Γ' *thesis*)
  case (*const name* Γ *val*)
  hence *name* $|\in|$ *ids* (*Sconst name*)
    unfolding *ids-def* **by** *simp*
  with *const* **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′* $\approx_e$ *val*
    using ‹*fmlookup* Γ *name* = *Some val*›
    by *cases auto*
  thus *?case*
    using *const* **by** (*auto intro*: *veval′.const*)
next
  case (*var* Γ *name val*)
  hence *name* $|\in|$ *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′* $\approx_e$ *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* $\Gamma' \approx_e$ *Vabs cs* Γ
    by (*auto intro*: *erelated.abs*)
  thus *?case*
    using *abs* **by** (*auto intro*: *veval′.abs*)
next
  case (*comb* Γ *t cs* Γ_Λ *u* v_2 *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 v_1' **where** $\Gamma' \vdash_v t \downarrow v_1'$ $v_1' \approx_e$ *Vabs cs* Γ_Λ
    using *comb* **by** (*auto simp*: *closed-except-def*)
  then obtain Γ_Λ' **where** v_1' = *Vabs cs* Γ_Λ' *fmrel-on-fset* (*ids* (*Sabs cs*)) *erelated* Γ_Λ' Γ_Λ
    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 $v_2{}'$ **where** $\Gamma' \vdash_v u \downarrow v_2{}'$ $v_2{}' \approx_e v_2$
  apply −
  apply (*erule comb.IH*(*2*))
  using *comb* **by** (*auto simp*: *closed-except-def*)

have *rel-option* (*rel-prod* (*fmrel erelated*) (=)) (*vfind-match cs* $v_2{}'$) (*vfind-match cs* v_2)
  using ‹$v_2{}' \approx_e v_2$› **by** (*rule erelated.vfind-match-rel*$'$)
then obtain *env*$'$ **where** *fmrel erelated env*$'$ *env vfind-match cs* $v_2{}'$ = *Some* (*env*$'$, *pat*, *rhs*)
  using *comb* **by** *cases auto*

have *vclosed* (*Vabs cs* Γ_Λ)
  apply (*rule veval*$'$*-closed*)
  using *comb* **by** (*auto simp*: *closed-except-def*)
have *vclosed* v_2
  apply (*rule veval*$'$*-closed*)
  using *comb* **by** (*auto simp*: *closed-except-def*)

have *closed-except* (*Sabs cs*) (*fmdom* Γ_Λ)
  using ‹*vclosed* (*Vabs cs* Γ_Λ)› **by** (*auto simp*: *Sterm.closed-except-simps*)
hence *frees* (*Sabs cs*) $|\subseteq|$ *fmdom* Γ_Λ
  unfolding *closed-except-def* **.**

have *vwellformed* (*Vabs cs* Γ_Λ)
  apply (*rule veval*$'$*-wellformed*)
    apply *fact*
  using *comb* **by** *auto*

have (*pat*, *rhs*) ∈ *set cs*
  by (*rule vfind-match-elem*) *fact*
hence *linear pat*
  using ‹*vwellformed* (*Vabs cs* Γ_Λ)›
  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 comb*)
  done

have *vwelldefined*$'$ (*Vabs cs* Γ_Λ)
  apply (*rule veval*$'$*-welldefined*$'$)

```
      apply fact
  using comb by auto
hence consts rhs |⊆| fmdom Γ_Λ |∪| C fdisjnt C (fmdom Γ_Λ)
  using ‹(pat, rhs) ∈ set cs›
  by (auto simp: list-all-iff)

have not-shadows-vconsts (Vabs cs Γ_Λ)
  apply (rule veval'-shadows)
  using comb by auto

obtain val' where Γ_Λ' ++_f env' ⊢_v rhs ↓ val' val' ≈_e val
  proof (erule comb.IH)
    show closed-venv (Γ_Λ ++_f env)
      apply rule
      using ‹vclosed (Vabs cs Γ_Λ)› apply simp
      apply (rule vclosed.vmatch-env)
       apply (rule vfind-match-elem)
       apply fact
      apply fact
      done
  next
    show wellformed rhs
      using ‹(pat, rhs) ∈ set cs› ‹vwellformed (Vabs cs Γ_Λ)›
      by (auto simp: list-all-iff)
  next
    show wellformed-venv (Γ_Λ ++_f env)
      apply rule
      using ‹vwellformed (Vabs cs Γ_Λ)› 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 (fmdom (Γ_Λ ++_f env))
      using ‹(pat, rhs) ∈ set cs› ‹vclosed (Vabs cs Γ_Λ)› ‹fmdom env = frees pat›
      by (auto simp: list-all-iff)

    have fmdom env = fmdom env'
      using ‹fmrel erelated env' env›
      by (metis fmrel-fmdom-eq)

    show fmrel-on-fset (ids rhs) erelated (Γ_Λ' ++_f env') (Γ_Λ ++_f env)
      proof
        fix id
        assume id |∈| ids rhs

       thus rel-option erelated (fmlookup (Γ_Λ' ++_f env') id) (fmlookup (Γ_Λ ++_f
```

env) *id*)
unfolding *ids-def*
proof (*cases rule*: *funion-strictE*)
case *A*
hence *id* |∈| *fmdom env* |∪| *fmdom* Γ_Λ
using ‹*closed-except rhs* (*fmdom* (Γ_Λ $++_f$ *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 *ffUnion-alt-def*
apply *simp*
apply (*rule fBexI*[**where** *x* = (*pat*, *rhs*)])
using ‹*id* |∈| *frees rhs*› **apply** *simp*
unfolding *fset-of-list-elem*
apply (*rule* ‹(*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 ‹*id* |∉| *fmdom env*›
apply *simp*
apply (*rule fmrel-on-fsetD*)
apply (*rule* ‹*id* |∈| *ids* (*Sabs cs*)›)
apply (*rule* ‹*fmrel-on-fset* (*ids* (*Sabs cs*)) *erelated* $\Gamma_\Lambda{}'$ Γ_Λ›)
done
qed
next
case *B*
have *id* |∈| *consts* (*Sabs cs*)
apply *auto*
unfolding *ffUnion-alt-def*
apply *simp*

```
          apply (rule fBexI[where x = (pat, rhs)])
           apply simp
           apply fact
          unfolding fset-of-list-elem
          apply (rule ‹(pat, rhs) ∈ set cs›)
          done
        hence id |∈| ids (Sabs cs)
          unfolding ids-def by auto

        show ?thesis
          using ‹fmdom env = fmdom env'›
          apply auto
           apply (rule fmrelD)
           apply (rule ‹fmrel erelated env' env›)
          apply (rule fmrel-on-fsetD)
           apply (rule ‹id |∈| ids (Sabs cs)›)
          apply (rule ‹fmrel-on-fset (ids (Sabs cs)) erelated Γ_Λ' Γ_Λ›)
          done
      qed
    qed
next
  show fmpred (λ-. vwelldefined') (Γ_Λ ++_f env)
    proof
      have vwelldefined' (Vabs cs Γ_Λ)
        apply (rule veval'-welldefined')
            apply fact
        using comb by auto
      thus fmpred (λ-. vwelldefined') Γ_Λ
        by simp
    next
      have vwelldefined' v_2
        apply (rule veval'-welldefined')
            apply fact
        using 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 Γ_Λ)
    using ‹vwelldefined' (Vabs cs Γ_Λ)› by simp
  moreover have fdisjnt C (fmdom env)
    unfolding ‹fmdom env = -›
    using ‹(pat, rhs) ∈ set cs› ‹not-shadows-vconsts (Vabs cs Γ_Λ)›
    by (auto simp: list-all-iff all-consts-def fdisjnt-alt-def)
  ultimately show fdisjnt C (fmdom (Γ_Λ ++_f env))
```

unfolding *fdisjnt-alt-def* **by** *auto*
next
show $\neg$ *shadows-consts rhs*
using ‹(*pat*, *rhs*) $\in$ *set cs*› ‹*not-shadows-vconsts* (*Vabs cs* Γ_Λ)›
by (*auto simp*: *list-all-iff*)
next
have *not-shadows-vconsts-env* Γ_Λ
using ‹*not-shadows-vconsts* (*Vabs cs* Γ_Λ)› **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 *comb* **by** *auto*
ultimately show *not-shadows-vconsts-env* (Γ_Λ $++_f$ *env*)
by *blast*
next
show *consts rhs* $|\subseteq|$ *fmdom* (Γ_Λ $++_f$ *env*) $|\cup|$ C
using ‹*consts rhs* $|\subseteq|$ -›
by *auto*
qed

moreover have $\Gamma' \vdash_v t \ \$_s\ u \downarrow val'$
proof (*rule veval'.comb*)
show $\Gamma' \vdash_v t \downarrow$ *Vabs cs* Γ_Λ'
using ‹$\Gamma' \vdash_v t \downarrow v_1'$›
unfolding ‹$v_1' =$ -› **.**
qed *fact+*

ultimately show *?case*
using *comb* **by** *metis*
next
case (*rec-comb* Γ t *css name* Γ_Λ *cs u* v_2 *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 v_1' **where** $\Gamma' \vdash_v t \downarrow v_1'$ $v_1' \approx_e$ *Vrecabs css name* Γ_Λ
using *rec-comb* **by** (*auto simp*: *closed-except-def*)
then obtain Γ_Λ'
where $v_1' =$ *Vrecabs css name* Γ_Λ'
and *pred-fmap* (λ*cs*. *fmrel-on-fset* (*ids* (*Sabs cs*)) *erelated* Γ_Λ' Γ_Λ) *css*
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 $v_2{}'$ **where** $\Gamma' \vdash_v u \downarrow v_2{}'$ $v_2{}' \approx_e v_2$
apply –
apply (*erule rec-comb.IH(2)*)
using *rec-comb* **by** (*auto simp*: *closed-except-def*)

have *rel-option* (*rel-prod* (*fmrel erelated*) (=)) (*vfind-match cs* $v_2{}'$) (*vfind-match cs* v_2)
using ‹$v_2{}' \approx_e v_2$› **by** (*rule erelated.vfind-match-rel′*)
then obtain *env′* **where** *fmrel erelated env′ env vfind-match cs* $v_2{}'$ = *Some* (*env′*, *pat*, *rhs*)
using *rec-comb* **by** *cases auto*

have *vclosed* (*Vrecabs css name* Γ_Λ)
apply (*rule veval′-closed*)
using *rec-comb* **by** (*auto simp*: *closed-except-def*)
hence *vclosed* (*Vabs cs* Γ_Λ)
using *rec-comb* **by** (*auto simp*: *closed-except-def*)
have *vclosed* v_2
apply (*rule veval′-closed*)
using *rec-comb* **by** (*auto simp*: *closed-except-def*)

have *closed-except* (*Sabs cs*) (*fmdom* Γ_Λ)
using ‹*vclosed* (*Vabs cs* Γ_Λ)› **by** (*auto simp*: *Sterm.closed-except-simps*)
hence *frees* (*Sabs cs*) $|\subseteq|$ *fmdom* Γ_Λ
unfolding *closed-except-def* **.**

have *vwellformed* (*Vrecabs css name* Γ_Λ)
apply (*rule veval′-wellformed*)
apply *fact*
using *rec-comb* **by** *auto*
hence *vwellformed* (*Vabs cs* Γ_Λ)
using *rec-comb* **by** (*auto simp*: *closed-except-def*)

have (*pat*, *rhs*) ∈ *set cs*
by (*rule vfind-match-elem*) *fact*
hence *linear pat*
using ‹*vwellformed* (*Vabs cs* Γ_Λ)›
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 *vwelldefined′* (*Vrecabs css name* Γ_Λ)
apply (*rule veval′-welldefined′*)

apply *fact*
using *rec-comb* **by** *auto*
hence *consts rhs* $|\subseteq|$ *fmdom* Γ_Λ $|\cup|$ (C $|\cup|$ *fmdom css*) *fdisjnt* C (*fmdom* Γ_Λ)
using ‹(*pat*, *rhs*) ∈ *set cs*› ‹*fmlookup css name* = *Some cs*›
by (*auto simp*: *list-all-iff dest*!: *fmpredD*[**where** m = *css*])

have *not-shadows-vconsts* (*Vrecabs css name* Γ_Λ)
apply (*rule veval'-shadows*)
using *rec-comb* **by** *auto*
hence *not-shadows-vconsts* (*Vabs cs* Γ_Λ)
using *rec-comb* **by** *auto*

obtain *val'* **where** $\Gamma_\Lambda{}' \mathbin{++_f}$ *mk-rec-env css* $\Gamma_\Lambda{}' \mathbin{++_f}$ *env'* $\vdash_v$ *rhs* ↓ *val' val'* $\approx_e$ *val*
proof (*erule rec-comb.IH*)
show *closed-venv* ($\Gamma_\Lambda \mathbin{++_f}$ *mk-rec-env css* $\Gamma_\Lambda \mathbin{++_f}$ *env*)
apply *rule*
apply *rule*
using ‹*vclosed* (*Vabs cs* Γ_Λ)› **apply** *simp*
unfolding *mk-rec-env-def*
using ‹*vclosed* (*Vrecabs css name* Γ_Λ)› **apply** (*auto intro*: *fmdomI*)[]
apply (*rule vclosed.vmatch-env*)
apply (*rule vfind-match-elem*)
apply *fact*
apply *fact*
done
next
show *wellformed rhs*
using ‹(*pat*, *rhs*) ∈ *set cs*› ‹*vwellformed* (*Vabs cs* Γ_Λ)›
by (*auto simp*: *list-all-iff*)
next
show *wellformed-venv* ($\Gamma_\Lambda \mathbin{++_f}$ *mk-rec-env css* $\Gamma_\Lambda \mathbin{++_f}$ *env*)
apply *rule*
apply *rule*
using ‹*vwellformed* (*Vabs cs* Γ_Λ)› **apply** *simp*
unfolding *mk-rec-env-def*
using ‹*vwellformed* (*Vrecabs css name* Γ_Λ)› **apply** (*auto intro*: *fmdomI*)[]
apply (*rule vwellformed.vmatch-env*)
apply (*rule vfind-match-elem*)
apply *fact*
apply (*rule veval'-wellformed*)
apply *fact*
using *rec-comb* **by** *auto*
next
have *closed-except rhs* (*fmdom* ($\Gamma_\Lambda \mathbin{++_f}$ *env*))
using ‹(*pat*, *rhs*) ∈ *set cs*› ‹*vclosed* (*Vabs cs* Γ_Λ)› ‹*fmdom env* = *frees pat*›
by (*auto simp*: *list-all-iff closed-except-def*)
thus *closed-except rhs* (*fmdom* ($\Gamma_\Lambda \mathbin{++_f}$ *mk-rec-env css* $\Gamma_\Lambda \mathbin{++_f}$ *env*))
unfolding *closed-except-def*

by *auto*

have *fmdom env* = *fmdom env′*
using ‹*fmrel erelated env′ env*›
by (*metis fmrel-fmdom-eq*)

have *fmrel-on-fset* (*ids rhs*) *erelated* (*mk-rec-env css* $\Gamma_\Lambda{}'$) (*mk-rec-env css* Γ_Λ)
unfolding *mk-rec-env-def*
apply *rule*
apply *simp*
unfolding *option.rel-map*
apply (*rule option.rel-refl*)
apply (*rule erelated.intros*)
apply (*rule* ‹*pred-fmap* (λ*cs. fmrel-on-fset* (*ids* (*Sabs cs*)) *erelated* $\Gamma_\Lambda{}'$ Γ_Λ) *css*›)
done

have *fmrel-on-fset* (*ids* (*Sabs cs*)) *erelated* $\Gamma_\Lambda{}'$ Γ_Λ
using ‹*pred-fmap* (λ*cs. fmrel-on-fset* (*ids* (*Sabs cs*)) *erelated* $\Gamma_\Lambda{}'$ Γ_Λ) *css*› *rec-comb*
by *auto*

have *fmdom* (*mk-rec-env css* Γ_Λ) = *fmdom* (*mk-rec-env css* $\Gamma_\Lambda{}'$)
unfolding *mk-rec-env-def* **by** *auto*

show *fmrel-on-fset* (*ids rhs*) *erelated* ($\Gamma_\Lambda{}'$ $++_f$ *mk-rec-env css* $\Gamma_\Lambda{}'$ $++_f$ *env′*) (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ $++_f$ *env*)
proof
fix *id*
assume *id* |∈| *ids rhs*

thus *rel-option erelated* (*fmlookup* ($\Gamma_\Lambda{}'$ $++_f$ *mk-rec-env css* $\Gamma_\Lambda{}'$ $++_f$ *env′*) *id*) (*fmlookup* (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ $++_f$ *env*) *id*)
unfolding *ids-def*
proof (*cases rule*: *funion-strictE*)
case *A*
hence *id* |∈| *fmdom env* |∪| *fmdom* Γ_Λ
using ‹*closed-except rhs* (*fmdom* (Γ_Λ $++_f$ *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 *ffUnion-alt-def*
apply *simp*
apply (*rule fBexI*[**where** *x* = (*pat*, *rhs*)])
using ‹*id* |∈| *frees rhs*› **apply** *simp*
unfolding *fset-of-list-elem*
apply (*rule* ‹(*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 ‹*id* |∉| *fmdom env*› ‹*fmdom* (*mk-rec-env css* Γ_Λ) = *fmdom* (*mk-rec-env css* $\Gamma_\Lambda{}'$)›
apply *auto*
apply (*rule fmrel-on-fsetD*)
apply (*rule* ‹*id* |∈| *ids rhs*›)
apply (*rule* ‹*fmrel-on-fset* (*ids rhs*) *erelated* (*mk-rec-env css* $\Gamma_\Lambda{}'$) (*mk-rec-env css* Γ_Λ)›)
apply (*rule fmrel-on-fsetD*)
apply (*rule* ‹*id* |∈| *ids* (*Sabs cs*)›)
apply (*rule* ‹*fmrel-on-fset* (*ids* (*Sabs cs*)) *erelated* $\Gamma_\Lambda{}'$ Γ_Λ›)
done
qed
next
case *B*
have *id* |∈| *consts* (*Sabs cs*)
apply *auto*
unfolding *ffUnion-alt-def*
apply *simp*
apply (*rule fBexI*[**where** *x* = (*pat*, *rhs*)])
apply *simp*
apply *fact*
unfolding *fset-of-list-elem*
apply (*rule* ‹(*pat*, *rhs*) ∈ *set cs*›)
done
hence *id* |∈| *ids* (*Sabs cs*)
unfolding *ids-def* **by** *auto*

show *?thesis*
using ‹*fmdom env* = *fmdom env'*› ‹*fmdom* (*mk-rec-env css* Γ_Λ) = *fmdom* (*mk-rec-env css* $\Gamma_\Lambda{}'$)›

```
              apply auto
                apply (rule fmrelD)
                apply (rule ‹fmrel erelated env' env›)
               apply (rule fmrel-on-fsetD)
                apply (rule ‹id |∈| ids rhs›)
                apply (rule ‹fmrel-on-fset (ids rhs) erelated (mk-rec-env css ΓΛ')
(mk-rec-env css ΓΛ)›)
               apply (rule fmrelD)
               apply (rule ‹fmrel erelated env' env›)
              apply (rule fmrel-on-fsetD)
               apply (rule ‹id |∈| ids (Sabs cs)›)
              apply (rule ‹fmrel-on-fset (ids (Sabs cs)) erelated ΓΛ' ΓΛ›)
              done
          qed
        qed
    next
      show fmpred (λ-. vwelldefined') (ΓΛ ++f mk-rec-env css ΓΛ ++f env)
      proof (intro fmpred-add)
        have vwelldefined' (Vrecabs css name ΓΛ)
          apply (rule veval'-welldefined')
                apply fact
          using rec-comb by auto
        thus fmpred (λ-. vwelldefined') ΓΛ fmpred (λ-. vwelldefined') (mk-rec-env
css ΓΛ)
          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 ΓΛ)›
        by (auto simp: list-all-iff all-consts-def fdisjnt-alt-def)
      moreover have fdisjnt C (fmdom css)
        using ‹vwelldefined' (Vrecabs css name ΓΛ)› by simp
      ultimately show fdisjnt C (fmdom (ΓΛ ++f mk-rec-env css ΓΛ ++f env))
        using ‹fdisjnt C (fmdom ΓΛ)›
        unfolding fdisjnt-alt-def mk-rec-env-def by auto
    next
```

show $\neg$ *shadows-consts rhs*
using ‹(*pat*, *rhs*) $\in$ *set cs*› ‹*not-shadows-vconsts* (*Vabs cs* Γ_Λ)›
by (*auto simp*: *list-all-iff*)
next
have *not-shadows-vconsts-env* Γ_Λ
using ‹*not-shadows-vconsts* (*Vabs cs* Γ_Λ)› **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_Λ)
unfolding *mk-rec-env-def*
using ‹*not-shadows-vconsts* (*Vrecabs css name* Γ_Λ)›
by (*auto intro*: *fmdomI*)
ultimately show *not-shadows-vconsts-env* (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ $++_f$ *env*)
by *blast*
next
show *consts rhs* |⊆| *fmdom* (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ $++_f$ *env*) |∪| *C*
using ‹*consts rhs* |⊆| -› **unfolding** *mk-rec-env-def*
by *auto*
qed

moreover have $\Gamma' \vdash_v t \$_s u \downarrow val'$
proof (*rule veval'.rec-comb*)
show $\Gamma' \vdash_v t \downarrow$ *Vrecabs css name* Γ_Λ'
using ‹$\Gamma' \vdash_v t \downarrow v_1'$›
unfolding ‹$v_1{}' =$ -› **.**
qed *fact+*

ultimately show *?case*
using *rec-comb* **by** *metis*
next
case (*constr name* Γ *ts us*)

have *list-all* ($\lambda t.$ *fmrel-on-fset* (*ids t*) *erelated* Γ' Γ $\wedge$ *closed-except t* (*fmdom* Γ) $\wedge$ *wellformed t* $\wedge$ *consts t* |⊆| *fmdom* Γ |∪| *C* $\wedge$ $\neg$ *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 $$ ts) (fmdom Γ)›
      unfolding closed.list-comb by (auto simp: list-all-iff)
    subgoal
      using ‹wellformed (name $$ ts)›
      unfolding wellformed.list-comb by (auto simp: list-all-iff)
    subgoal
      using ‹consts (name $$ ts) |⊆| fmdom Γ |∪| C›
      unfolding consts-list-comb
      by (metis Ball-set constr.prems(8) special-constants.sconsts-list-comb)
    subgoal
      using ‹¬ shadows-consts (name $$ ts)›
      unfolding shadows.list-comb by (auto simp: list-ex-iff)
    done
  done

obtain us' where list-all3 (λt u u'. Γ' ⊢v t ↓ u' ∧ u' ≈e u) ts us us'
  using ‹list-all2 - - -› ‹list-all - ts›
  proof (induction arbitrary: thesis rule: list.rel-induct)
    case (Cons t ts u us)
    then obtain us' where list-all3 (λt u u'. Γ' ⊢v t ↓ u' ∧ u' ≈e u) ts us us'
      by auto
    have
      fmrel-on-fset (ids t) erelated Γ' Γ closed-except t (fmdom Γ)
      wellformed t consts t |⊆| fmdom Γ |∪| C ¬ shadows-consts t
      using Cons by auto

    then obtain u' where Γ' ⊢v t ↓ u' u' ≈e u
       using ‹closed-venv Γ› ‹wellformed-venv Γ› ‹fdisjnt C (fmdom Γ)› ‹fmpred
(λ-. vwelldefined') Γ›
      using ‹not-shadows-vconsts-env Γ› Cons.hyps
      by blast

    show ?case
      apply (rule Cons.prems)
      apply (rule list-all3-cons)
       apply fact
      apply (rule conjI)
       apply fact+
      done
  qed auto

show ?case
  apply (rule constr.prems)
   apply (rule veval'.constr[where us = us'])
    apply fact
  using ‹list-all3 - ts us us'›
   apply (induct; auto)
  apply (rule erelated.intros)
  using ‹list-all3 - ts us us'›
```

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*: $rs \vdash t \approx a \Longrightarrow wellformed\ t$
begin

definition *eval'* :: *rule fset* ⇒ *term* ⇒ *'a* ⇒ *bool* (‹-/ ⊢/ (- ↓/ -)› [*50*,*0*,*50*] *50*)
where
$rs \vdash t \downarrow a \longleftrightarrow wellformed\ t \wedge (\exists t'.\ rs \vdash t \longrightarrow* t' \wedge rs \vdash t' \approx a)$

lemma *eval'I*[*intro*]:

assumes *wellformed t rs* $\vdash$ $t \longrightarrow\!* t'$ *rs* $\vdash$ $t' \approx a$
shows *rs* $\vdash$ $t \downarrow a$
using *assms* **unfolding** *eval'-def* **by** *auto*

lemma *eval'E*[*elim*]:
assumes *rs* $\vdash$ $t \downarrow a$
obtains t' **where** *wellformed t rs* $\vdash$ $t \longrightarrow\!* t'$ *rs* $\vdash$ $t' \approx a$
using *assms* **unfolding** *eval'-def* **by** *auto*

lemma *eval-trivI*: *rs* $\vdash$ $t \approx a \Longrightarrow$ *rs* $\vdash$ $t \downarrow a$
by (*auto dest*: *eval-wellformed*)

lemma *eval-compose*:
assumes *wellformed t rs* $\vdash$ $t \longrightarrow\!* t'$ *rs* $\vdash$ $t' \downarrow a$
shows *rs* $\vdash$ $t \downarrow a$
proof −
from ‹*rs* $\vdash$ $t' \downarrow a$› **obtain** t'' **where** *rs* $\vdash$ $t' \longrightarrow\!* t''$ *rs* $\vdash$ $t'' \approx a$
by *blast*
moreover hence *rs* $\vdash$ $t \longrightarrow\!* t''$
using *assms* **by** *auto*
ultimately show *rs* $\vdash$ $t \downarrow a$
using *assms* **by** *auto*
qed

end

instantiation *fun* :: (*evaluate*, *evaluate*) *evaluate* **begin**

definition *eval-fun* **where**
eval-fun rs t a $\longleftrightarrow$ *wellformed t* $\wedge$ ($\forall x\ t_x$. *rs* $\vdash$ $t_x \downarrow x \longrightarrow$ *rs* $\vdash$ $t\ \$\ t_x \downarrow a\ x$)

instance
by *standard* (*simp add*: *eval-fun-def*)

end

corollary *eval-funD*:
assumes *rs* $\vdash$ $t \approx f$ *rs* $\vdash$ $t_x \downarrow x$
shows *rs* $\vdash$ $t\ \$\ t_x \downarrow f\ x$
using *assms* **unfolding** *eval-fun-def* **by** *blast*

corollary *eval'-funD*:
assumes *rs* $\vdash$ $t \downarrow f$ *rs* $\vdash$ $t_x \downarrow x$
shows *rs* $\vdash$ $t\ \$\ t_x \downarrow f\ x$
proof −
from *assms* **obtain** t' **where** *rs* $\vdash$ $t \longrightarrow\!* t'$ *rs* $\vdash$ $t' \approx f$
by *auto*
have *wellformed* ($t\ \$\ t_x$)
using *assms* **by** *auto*

moreover have $rs \vdash t \ \$ \ t_x \longrightarrow\!* \ t' \ \$ \ t_x$
using ‹$rs \vdash t \longrightarrow\!* \ t'$› **by** (*rule rewrite.rt-fun*[*unfolded app-term-def*])
moreover have $rs \vdash t' \ \$ \ t_x \downarrow f\ x$
using ‹$rs \vdash t' \approx f$› *assms*(*2*) **by** (*rule eval-funD*)
ultimately show $rs \vdash t \ \$ \ t_x \downarrow f\ x$
by (*rule eval-compose*)
qed

lemma *eval-ext*:
assumes *wellformed* f $\bigwedge x\ t.\ rs \vdash t \downarrow x \Longrightarrow rs \vdash f \ \$ \ t \downarrow a\ x$
shows $rs \vdash f \approx a$
using *assms* **unfolding** *eval-fun-def* **by** *blast*

lemma *eval′-ext*:
assumes *wellformed* f $\bigwedge x\ t.\ rs \vdash t \downarrow x \Longrightarrow rs \vdash f \ \$ \ t \downarrow a\ x$
shows $rs \vdash f \downarrow a$
apply (*rule eval′I*[*OF* ‹*wellformed f*›])
apply (*rule rtranclp.rtrancl-refl*)
apply (*rule eval-ext*)
using *assms* **by** *auto*

lemma *eval′-ext-alt*:
fixes f :: *′a*::*evaluate* ⇒ *′b*::*evaluate*
assumes *wellformed′ 1 t* $\bigwedge u\ x.\ rs \vdash u \downarrow x \Longrightarrow rs \vdash t\ [u]_\beta \downarrow f\ x$
shows $rs \vdash \Lambda\ t \downarrow f$
proof (*rule eval′-ext*)
show *wellformed* (Λ *t*)
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* (Λ t \$ u)
using *assms* ‹$rs \vdash u \downarrow x$› **by** *auto*
next
show $rs \vdash \Lambda\ t \ \$ \ u \longrightarrow\!* \ t\ [u]_\beta$
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*]: $rs \vdash t \approx a \Longrightarrow$ *wellformed′ n t*
by (*auto dest*: *wellformed-inc eval-wellformed*[*unfolded wellformed-term-def*])

lemma *eval′-impl-wellformed*[*dest*]: $rs \vdash t \downarrow a \Longrightarrow$ *wellformed′ n t*
unfolding *eval′-def* **by** (*auto dest*: *wellformed-inc*)

lemma *wellformed-unpack*:
  $wellformed'\ n\ (t\ \$\ u) \Longrightarrow wellformed'\ n\ t$
  $wellformed'\ n\ (t\ \$\ u) \Longrightarrow wellformed'\ n\ u$
  $wellformed'\ n\ (\Lambda\ t) \Longrightarrow wellformed'\ (Suc\ n)\ t$
by *auto*

lemma *replace-bound-aux*:
  $n < 0 \longleftrightarrow False$
  $Suc\ n < Suc\ m \longleftrightarrow n < m$
  $0 < Suc\ n \longleftrightarrow True$
  $((0::nat) = 0) \longleftrightarrow True$
  $(0 = Suc\ m) \longleftrightarrow False$
  $(Suc\ m = Suc\ n) \longleftrightarrow n = m$
  $(Suc\ m = 0) \longleftrightarrow False$
  $(if\ True\ then\ P\ else\ Q) = P$
  $(if\ False\ then\ P\ else\ Q) = Q$
  $int\ (0::nat) = 0$
by *auto*

named-theorems *eval-data-intros*
named-theorems *eval-data-elims*

context begin

private definition $rewrite\text{-}step\text{-}term :: term \times term \Rightarrow term \Rightarrow term\ option$
where
$rewrite\text{-}step\text{-}term = rewrite\text{-}step$

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* = ‹*Scan.succeed* (*SIMPLE-METHOD′ o Tactics.wellformed-tac*)›

end

4.2 Deep embedding of Pure terms into term-rewriting logic

theory *Embed*
imports
  Constructor-Funs.Constructor-Funs
  ../Utils/Code-Utils

Eval-Class
keywords *embed* :: *thy-goal*
begin

fun *non-overlapping′* :: *term* ⇒ *term* ⇒ *bool* **where**
non-overlapping′ (*Const x*) (*Const y*) ⟷ $x \neq y$ |
non-overlapping′ (*Const* -) (- $ -) ⟷ *True* |
non-overlapping′ (- $ -) (*Const* -) ⟷ *True* |
non-overlapping′ (t_1 $ t_2) (u_1 $ u_2) ⟷ *non-overlapping′* t_1 u_1 ∨ *non-overlapping′* t_2 u_2 |
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+*

fun *pattern-compatible′* :: *term* ⇒ *term* ⇒ *bool* **where**
pattern-compatible′ (t_1 $ t_2) (u_1 $ u_2) ⟷ *pattern-compatible′* t_1 u_1 ∧ ($t_1 = u_1$ ⟶ *pattern-compatible′* t_2 u_2) |
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(lhs_1, -) (lhs_2, -). *pattern-compatible′* lhs_1 lhs_2)

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*) ∧

fBall rs (λ(-, *rhs*). *pre-constants.welldefined C-info* (*heads-of rs*) *rhs*) ∧
distinct (*constructors.all-constructors C-info*)

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-weaken*) (*blast intro*: *pattern-compatible-approx*)
qed

lemma *embed-ext*: $f \equiv g \Longrightarrow f\ x \equiv g\ x$
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*

end

Chapter 5

Final stage: Translation to CakeML

theory *Doc-Backend*
imports *Main*
begin

end

5.1 Basic CakeML setup

theory *CakeML-Setup*
imports
  ../CupCakeML/CupCake-Semantics
  CakeML.CakeML-Code
  ../Terms/Consts
begin

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 *fmlookup-ns*[*simp*]: *fmlookup* (*fmap-of-ns ns*) *k* = *cupcake-nsLookup ns* (*as-string k*)
by (*cases ns*) (*simp add*: *fmlookup-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*) = *fmupd 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* = *fmempty*
  unfolding *nsEmpty-def* **by** *simp*

context begin

private lemma *build-rec-env-fmap0*:
  fmap-of-ns (*foldr* (λ(*f*, *x*, *e*). *nsBind f* (*Recclosure* env_Λ *funs' f*)) *funs env*) =
    fmap-of-ns env $++_f$ *fmap-of-list* (*map* (λ(*f*, -). (*Name f*, *Recclosure* env_Λ *funs' f*)) *funs*)
apply (*induction funs arbitrary*: *env*)
 apply *auto*
by (*metis* (*no-types*, *lifting*) *fmap-of-nsBind name.sel*)

definition *cake-mk-rec-env* **where**
cake-mk-rec-env funs env = *fmap-of-list* (*map* (λ(*f*, -). (*Name f*, *Recclosure env funs f*)) *funs*)

lemma *build-rec-env-fmap*:
fmap-of-ns (*build-rec-env funs* env_Λ *env*) = *fmap-of-ns env* $++_f$ *cake-mk-rec-env funs* env_Λ
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*, *defined-mods* = {} ⦈

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* (λ*name*. *TypeId* (*Short* (*as-string name*))) (*sorted-list-of-fset* (*fmdom*
C-info)))
unfolding *cake-all-types-def 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-convoluted2*: *foldr* ($\lambda x\ y.\ f\ x$ @ y) *xs ys* = *concat* (*map f xs*) @ *ys*
by (*induction xs arbitrary*: *ys*) *auto*

private lemma *check-dup-ctors-alt-def*: *check-dup-ctors tds* $\longleftrightarrow$ *distinct* (*tds* $\gg=$ (λ(-, -, *cons*). *map fst cons*))
unfolding *check-dup-ctors-def*
apply *simp*
apply (*rule arg-cong*[**where** *f* = *distinct*])
apply (*subst foldr-cong*[*OF refl refl*, **where** $g = \lambda x\ a.$ *map fst* (*snd* (*snd x*)) @ *a*])
subgoal
  apply (*subst split-beta*)
  apply (*subst split-beta*)
  by (*rule cakeml-convoluted1*)
subgoal
  apply (*subst cakeml-convoluted2*)
  apply *auto*
  unfolding *List.bind-def*
  apply (*rule arg-cong*[**where** *f* = *concat*])
  by *auto*
done

lemma *evaluate-dec-prelude*:
  evaluate-dec t [] *env empty-state cake-dt-prelude* (*empty-state-with-types*, *Rval empty-sem-env*)
unfolding *cake-dt-prelude-def*
proof (*rule evaluate-type*, *intro conjI*)
  show *check-dup-ctors as-cake-type-def*
    using *distinct-ctr'*
   unfolding *check-dup-ctors-alt-def List.bind-def as-cake-type-def-def all-constructors-def*
    by (*auto simp*: *comp-def split-beta map-concat*)
next
  show *allDistinct* (*map* ($\lambda x.$ *case x of* (*tvs*, *tn*, *ctors*) $\Rightarrow$ *tn*) *as-cake-type-def*)
    unfolding *all-distinct-alt-def as-cake-type-def-def*
    apply (*auto simp*: *comp-def case-prod-twice*)
    apply (*rule name-as-string.fst-distinct*)
    unfolding *sorted-list-of-fmap-def*
    by (*auto simp*: *comp-def*)
next
  show *cake-all-types* = *type-defs-to-new-tdecs* [] *as-cake-type-def*
    unfolding *cake-all-types-def type-defs-to-new-tdecs all-tdefs-def*
    by *simp*
next
  show *disjnt cake-all-types* (*defined-types empty-state*)
    unfolding *empty-state-def disjnt-def* **by** *simp*
qed

end

end

Computability

declare *constructors.as-static-cenv-def*[*code*]
declare *constructors.as-cake-type-def-def*[*code*]
declare *constructors.cake-dt-prelude-def*[*code*]

export-code *constructors.as-static-cenv constructors.cake-dt-prelude*
 checking *Scala*

end

5.3 CakeML backend

theory *CakeML-Backend*
imports
 CakeML-Setup
 ../Terms/Value
 ../Rewriting/Rewriting-Sterm
begin

5.3.1 Compilation

fun *mk-ml-pat* :: *pat* ⇒ *Ast.pat* **where**
mk-ml-pat (*Patvar s*) = *Ast.Pvar* (*as-string s*) |
mk-ml-pat (*Patconstr s args*) = *Ast.Pcon* (*Some* (*Short* (*as-string s*))) (*map mk-ml-pat args*)

lemma *mk-pat-cupcake*[*intro*]: *is-cupcake-pat* (*mk-ml-pat pat*)
by (*induct pat*) (*auto simp*: *list-all-iff*)

context begin

private fun *frees'* :: *term* ⇒ *name list* **where**
frees' (*Free x*) = [*x*] |
frees' (t_1 \$ t_2) = *frees'* t_2 @ *frees'* t_1 |
frees' (Λ *t*) = *frees' t* |
frees' - = []

private lemma *frees'-eq*[*simp*]: *fset-of-list* (*frees' t*) = *frees t*
by (*induction t*) *auto*

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*)

private lemma *frees'-distinct*: *linear pat* ⟹ *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.Ptannot 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′* ∘ *mk-ml-pat* ∘ *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′* ∘ *mk-ml-pat* ∘ *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*
qed

lemma *mk-pat-distinct*: *linear pat* $\Longrightarrow$ *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* = *pre-constants*
begin

fun
mk-exp :: *name fset* $\Rightarrow$ *sterm* $\Rightarrow$ *exp* **and**
mk-clauses :: *name fset* $\Rightarrow$ (*term* $\times$ *sterm*) *list* $\Rightarrow$ (*Ast.pat* $\times$ *exp*) *list* **and**
mk-con :: *name fset* $\Rightarrow$ *sterm* $\Rightarrow$ *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 (t_1 $\$_s$ t_2) = *Ast.App Ast.Opapp* [*mk-con S* t_1, *mk-con S* t_2] |
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*) $\Rightarrow$
if c |∈| *C then Ast.Con* (*Some* (*Short* (*as-string c*))) (*map* (*mk-con S*) *args*) *else mk-exp S t*
| - $\Rightarrow$ *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 $\Longrightarrow$ *is-cupcake-exp* (*mk-exp S t*)
wellformed-clauses cs $\Longrightarrow$ *cupcake-clauses* (*mk-clauses S cs*) ∧ *cake-linear-clauses* (*mk-clauses S cs*)
wellformed t $\Longrightarrow$ *is-cupcake-exp* (*mk-con S t*)
proof (*induction rule*: *mk-exp-mk-clauses-mk-con.induct*)
case (*5 S t*)
show *?case*
apply (*simp split*!: *prod.splits sterm.splits if-splits*)
subgoal premises *prems* **for** *args c*
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 ⟹ *is-cupcake-exp* (*mk-exp S t*)
wellformed t ⟹ *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* ⇒ *srule list* ⇒ *Ast.dec* **where**
compile-group S rs = *Ast.Dletrec empty-locs* (*mk-letrec-body S rs*)

definition *compile* :: *srule list* ⇒ *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*]
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* **..**

5.3.2 Computability

export-code *cakeml.compile*
  checking *Scala*

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*) |
app: *related-exp* t_1 u_1 ⟹ *related-exp* t_2 u_2 ⟹ *related-exp* (t_1 $\$_s$ t_2) (*Ast.App Ast.Opapp* [u_1, u_2]) |
fun: *list-all2* (*rel-prod related-pat related-exp*) *cs ml-cs* ⟹

n |∉| *ids* (*Sabs cs*) $\Longrightarrow$ n |∉| *all-consts* $\Longrightarrow$
related-exp (*Sabs cs*) (*Ast.Fun* (*as-string n*) (*Ast.Mat* (*Ast.Var* (*Short* (*as-string n*))) *ml-cs*)) |
mat: *list-all2* (*rel-prod related-pat related-exp*) *cs ml-cs* $\Longrightarrow$
related-exp scr ml-scr $\Longrightarrow$
related-exp (*Sabs cs* $\$_s$ *scr*) (*Ast.Mat ml-scr ml-cs*)

lemma *related-exp-is-cupcake*:
assumes *related-exp t e wellformed t*
shows *is-cupcake-exp e*
using *assms* **proof** *induction*
case (*fun cs ml-cs n*)
hence *list-all* (λ(*pat*, *t*). *linear pat* ∧ *wellformed t*) *cs* **by** *simp*
moreover have *cupcake-clauses ml-cs* ∧ *cake-linear-clauses ml-cs*
using ‹*list-all2 - cs ml-cs*› ‹*list-all - cs*›
proof *induction*
case (*Cons c cs ml-c ml-cs*)
obtain *ml-p ml-e* **where** *ml-c* = (*ml-p*, *ml-e*) **by** *fastforce*
obtain *p t* **where** *c* = (*p*, *t*) **by** *fastforce*

have *ml-p* = *mk-ml-pat* (*mk-pat p*)
using *Cons* **unfolding** ‹*ml-c* = -› ‹*c* = -› **by** *simp*
thus *?case*
using *Cons* **unfolding** ‹*ml-c* = -› ‹*c* = -› **by** (*auto intro*: *mk-pat-distinct*)
qed *simp*

ultimately show *?case*
by *auto*
next
case (*mat cs ml-cs scr ml-scr*)
hence *list-all* (λ(*pat*, *t*). *linear pat* ∧ *wellformed t*) *cs* **by** *simp*
moreover have *cupcake-clauses ml-cs* ∧ *cake-linear-clauses ml-cs*
using ‹*list-all2 - cs ml-cs*› ‹*list-all - cs*›
proof *induction*
case (*Cons c cs ml-c ml-cs*)
obtain *ml-p ml-e* **where** *ml-c* = (*ml-p*, *ml-e*) **by** *fastforce*
obtain *p t* **where** *c* = (*p*, *t*) **by** *fastforce*

have *ml-p* = *mk-ml-pat* (*mk-pat p*)
using *Cons* **unfolding** ‹*ml-c* = -› ‹*c* = -› **by** *simp*
thus *?case*
using *Cons* **unfolding** ‹*ml-c* = -› ‹*c* = -› **by** (*auto intro*: *mk-pat-distinct*)
qed *simp*

ultimately show *?case*
using *mat* **by** *auto*
next
case (*constr name ts es*)
hence *list-all wellformed ts*

by (*simp add*: *wellformed.list-comb*)
with ‹*list-all2* - *ts es*› **have** *list-all is-cupcake-exp es*
by *induction auto*
thus *?case*
by *simp*
qed *auto*

definition *related-fun* :: (*term* × *sterm*) *list* ⇒ *name* ⇒ *exp* ⇒ *bool* **where**
related-fun cs n e ⟷
n |∉| *ids* (*Sabs cs*) ∧ *n* |∉| *all-consts* ∧ (*case e of*
(*Ast.Mat* (*Ast.Var* (*Short n′*)) *ml-cs*) ⇒
n = *Name n′* ∧ *list-all2* (*rel-prod related-pat related-exp*) *cs ml-cs*
| - ⇒ *False*)

lemma *related-fun-alt-def*:
related-fun cs n (*Ast.Mat* (*Ast.Var* (*Short* (*as-string n*))) *ml-cs*) ⟷
list-all2 (*rel-prod related-pat related-exp*) *cs ml-cs* ∧
n |∉| *ids* (*Sabs cs*) ∧ *n* |∉| *all-consts*
unfolding *related-fun-def*
by *auto*

lemma *related-funE*:
assumes *related-fun cs n e*
obtains *ml-cs*
where *e* = *Ast.Mat* (*Ast.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 *assms* **unfolding** *related-fun-def*
by (*simp split*: *exp0.splits id0.splits*)

lemma *related-exp-fun*:
related-fun cs n e ⟷ *related-exp* (*Sabs cs*) (*Ast.Fun* (*as-string n*) *e*) ∧ *n* |∉| *ids*
(*Sabs cs*) ∧ *n* |∉| *all-consts*
(**is** *?lhs* ⟷ *?rhs*)
proof
assume *?rhs*
hence *related-exp* (*Sabs cs*) (*Ast.Fun* (*as-string n*) *e*) **by** *simp*
thus *?lhs*
by *cases* (*auto simp*: *related-fun-def dest*: *name.expand*)
next
assume *?lhs*
thus *?rhs*
by (*auto intro*: *related-exp.fun elim*: *related-funE*)
qed

inductive *related-v* :: *value* ⇒ *v* ⇒ *bool* **where**
conv:
list-all2 related-v us vs ⟹
related-v (*Vconstr name us*) (*Conv* (*Some* (*as-string name*, -)) *vs*) |

closure:
  related-fun cs n e $\Longrightarrow$
   fmrel-on-fset (*ids* (*Sabs cs*)) *related-v* Γ (*fmap-of-ns* (*sem-env.v env*)) $\Longrightarrow$
    related-v (*Vabs cs* Γ) (*Closure env* (*as-string n*) *e*) |
rec-closure:
  fmrel-on-fset (*fbind* (*fmran css*) (*ids* $\circ$ *Sabs*)) *related-v* Γ (*fmap-of-ns* (*sem-env.v env*)) $\Longrightarrow$
   fmrel (λ*cs.* λ(*n*, *e*). *related-fun cs n e*) *css* (*fmap-of-list* (*map* (*map-prod Name* (*map-prod Name id*)) *exps*)) $\Longrightarrow$
    related-v (*Vrecabs css name* Γ) (*Recclosure env exps* (*as-string name*))

abbreviation *var-env* :: (*name*, *value*) *fmap* $\Rightarrow$ (*string* $\times$ *v*) *list* $\Rightarrow$ *bool* **where**
var-env Γ *ns* $\equiv$ *fmrel related-v* Γ (*fmap-of-list* (*map* (*map-prod Name id*) *ns*))

lemma *related-v-ext*:
  assumes *related-v v ml-v*
  assumes $v' \approx_e v$
  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 ‹$v' \approx_e$ *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 ‹$v' \approx_e$ -›
    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* (λ*cs. fmrel-on-fset* (*ids* (*Sabs cs*)) *erelated* Γ_1 Γ_2) *css*
using ‹$v' \approx_e$ -›
by *cases auto*

have *fmrel-on-fset* (*fbind* (*fmran css*) (*ids* $\circ$ *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* $\times$ *v*) *list* $\Rightarrow$ (*string* $\times$ *v*) *list match-result* $\Rightarrow$ (*name*, *value*) *fmap option* $\Rightarrow$ *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* $\Longrightarrow$ *match-result-related eenv* (*Match* (*eenv-m* @ *eenv*)) (*Some* Γ)

private corollary *match-result-related-empty*: *match-result-related eenv* (*Match eenv*) (*Some fmempty*)
proof −
have *match-result-related eenv* (*Match* ([] @ *eenv*)) (*Some fmempty*)

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*t 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 ∗: *match-result-related eenv*
(*Matching.fold2 f Match-type-error* (*map mk-ml-pat ps*) *vs* (*Match* (*eenv-m* @ *eenv*)))

```
        (map-option (foldl (++_f) Γ) (those (map2 vmatch ps us))) (is ?rel)
    if var-env Γ eenv-m
    for eenv-m Γ
    using Patconstr.IH ‹related-v v0 ml-v› that
    proof (induction us ps vs arbitrary: Γ 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-cenv
(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 (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 (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 Γ' eenv-m')

          have match-result-related eenv
                    (Matching.fold2 f Match-type-error (map mk-ml-pat ps) vs
(Match ((eenv-m' @ eenv-m) @ eenv)))
                  (map-option (foldl (++_f) (Γ ++_f Γ')) (those (map2 vmatch ps
us)))
            proof (rule Cons, rule Cons)
              show var-env (Γ ++_f Γ') (eenv-m' @ eenv-m)
                using ‹var-env Γ eenv-m› match
                by force
            qed (simp | fact)+

          thus ?thesis
            using match
            unfolding f-def
            by (auto simp: map-option.compositionality comp-def)
        qed
    qed (auto intro: match-result-related.match)

  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* (λ(*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* Γ **where**
ml-pat = *mk-ml-pat* (*mk-pat pat*)
related-exp rhs ml-rhs
vfind-match cs v = *Some* (Γ, *pat*, *rhs*)
var-env Γ *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** ‹*c* = -› **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** ‹*ml-c* = -›
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** ‹*ml-c* = -›
by (*simp split*: *if-splits*)
next
fix *pat rhs* Γ
assume *ml-pat* = *mk-ml-pat* (*mk-pat pat*)
assume *related-exp rhs ml-rhs*
assume *vfind-match cs v* = *Some* (Γ, *pat*, *rhs*)
assume *var-env* Γ *eenv*

from *rel* **have** *match-result-related* [] *No-match* (*vmatch* (*mk-pat pat0*) *v*)
using *No-match* **unfolding** ‹*ml-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 ‹ml-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 ‹ml-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
qed simp

end end

end
```

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* (λ*cs* (*n*, *e*). *related-fun cs n e*) *css* (*fmap-of-list* (*map* (*map-prod Name* (*map-prod Name id*)) *funs*))
  assumes *fmrel-on-fset* (*fbind* (*fmran css*) (*ids* $\circ$ *Sabs*)) *related-v* Γ_Λ (*fmap-of-ns* (*sem-env.v env*$_\Lambda$))
  shows *fmrel related-v* (*mk-rec-env css* Γ_Λ) (*cake-mk-rec-env funs env*$_\Lambda$)
proof (*rule fmrelI*)
  fix *name*
  have *rel-option* (λ*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*: *fmap-of-list.rep-eq*)

  then have *rel-option* (λ*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* Γ_Λ) (*Recclosure env*$_\Lambda$ *funs* (*as-string name*))
    using *assms* **by** (*auto intro*: *related-v.rec-closure*)

  show *rel-option related-v* (*fmlookup* (*mk-rec-env css* Γ_Λ) *name*) (*fmlookup* (*cake-mk-rec-env funs env*$_\Lambda$) *name*)
    unfolding *mk-rec-env-def cake-mk-rec-env-def fmap-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*)
  ids t $|\subseteq|$ *S* $\Longrightarrow$ *all-consts* $|\subseteq|$ *S* $\Longrightarrow$ $\neg$ *shadows-consts t* $\Longrightarrow$ *related-exp t* (*mk-con S t*)
proof (*induction rule*: *mk-exp-mk-clauses-mk-con.induct*)
  case (*2 S name*)
  show *?case*

```
    proof (cases name |∈| C)
      case True
      hence related-exp (name $$ []) (mk-exp S (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 fresh-fNext (S |∪| all-consts) |∉| S |∪| all-consts
    by (rule fNext-not-member)
  hence fresh-fNext S |∉| S |∪| all-consts
    using ‹all-consts |⊆| S› by (simp add: sup-absorb1)
  hence fresh-fNext S |∉| ids (Sabs cs) |∪| 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 |∉| ids (Sabs cs) |∪| 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 $$ args
          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
          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-ml-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› ‹(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 auto
        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* (*fmdom* Γ)
assumes *fmpred* (λ-. *vwelldefined'*) Γ *consts t* |⊆| *fmdom* Γ |∪| *C*
assumes *fdisjnt C* (*fmdom* Γ)
assumes ¬ *shadows-consts t not-shadows-vconsts-env* Γ
shows *if-rval* (λ*ml-v*. ∃ *v*. $\Gamma \vdash_v t \downarrow v$ ∧ *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-rval list-all2-rev1*,
folded ml-vs'-def]
moreover have
list-all wellformed ts list-all (λ*t*. ¬ *shadows-consts t*) *ts*
list-all (λ*t*. *consts t* |⊆| *fmdom* Γ |∪| *C*) *ts list-all* (λ*t*. *closed-except t* (*fmdom* 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* |⊆| *fmdom* Γ |∪| *C*›
unfolding *list-all-iff*
by (*metis Ball-set* ‹*t* = *name* $$ *ts*› *con1.prems*(*9*) *special-constants.sconsts-list-comb*)
subgoal

using ‹*closed-except t* (*fmdom* Γ)› **unfolding** ‹*t* = -› *closed.list-comb* **by** *simp*
done
moreover have
list-all (λ*t*′. *fmrel-on-fset* (*ids t*′) *related-v* Γ (*fmap-of-ns* (*sem-env.v env*))) *ts*
proof (*standard*, *rule fmrel-on-fsubset*)
fix *t*′
assume *t*′ ∈ *set ts*
thus *ids t*′ |⊆| *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-elem intro*!: *ffUnion-subset-elem*)
done
show *fmrel-on-fset* (*ids t*) *related-v* Γ (*fmap-of-ns* (*sem-env.v env*))
by *fact*
qed

ultimately obtain *us* **where** *list-all2* (*veval*′ Γ) *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 *t 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*′ Γ) *ts us list-all2 related-v us ml-vs*
unfolding ‹*ts0* = -›
by *auto*

from *Cons.hyps*[*simplified*, *THEN conjunct2*, *rule-format*, *of t* Γ]
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-venv Γ *closed-venv* Γ
fmpred (λ-. *vwelldefined*′) Γ *fdisjnt C* (*fmdom* Γ)
not-shadows-vconsts-env Γ
by *fact*+
next
show
wellformed t ¬ *shadows-consts t closed-except t* (*fmdom* Γ)
consts t |⊆| *fmdom* Γ |∪| *C fmrel-on-fset* (*ids t*) *related-v* Γ (*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 |∉| C
    unfolding ‹id = _›
    apply (cases t)
    using name.expand by blast+
  thus ?case
    proof cases
      case var
      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'.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 = -›
apply (*rule veval′.abs*)
unfolding ‹n = -›
apply (*rule related-v.closure*)
unfolding ‹u = -›
apply (*subst related-fun-alt-def*; *rule conjI*)
apply *fact*
apply (*rule conjI*; *fact*)
using ‹*fmrel-on-fset* (*ids t*) - - -›
unfolding ‹t = -› ‹*fmap-of-ns* (*sem-env.v env*) = -›
by *simp*
next
case (*app1 env exps ress ml-vs env′ exp′ bv*)
from ‹*related-exp t* -› **obtain** exp_1 exp_2 t_1 t_2
where *rev exps* = [exp_2, exp_1] *exps* = [exp_1, exp_2] $t = t_1 \$_s t_2$
and *related-exp* t_1 exp_1 *related-exp* t_2 exp_2
by *cases auto*
moreover from *app1* **have** *ress* = *map Rval ml-vs*
by *auto*
ultimately obtain *ml-*v_1 *ml-*v_2 **where** *ml-vs* = [*ml-*v_2, *ml-*v_1]
using *app1*(*1*)
by (*smt list-all2-shortcircuit-rval list-all2-Cons1 list-all2-Nil*)

have *is-cupcake-exp* exp_1 *is-cupcake-exp* exp_2
using *app1* **unfolding** ‹*exps* = -› **by** (*auto dest*: *related-exp-is-cupcake*)
moreover have *fmrel-on-fset* (*ids* t_1) *related-v* Γ (*fmap-of-ns* (*sem-env.v env*))
using *app1* **unfolding** *ids-def* ‹t = -›
by (*auto intro*: *fmrel-on-fsubset*)
moreover have *fmrel-on-fset* (*ids* t_2) *related-v* Γ (*fmap-of-ns* (*sem-env.v env*))
using *app1* **unfolding** *ids-def* ‹t = -›
by (*auto intro*: *fmrel-on-fsubset*)
ultimately have
cupcake-evaluate-single env exp_1 (*Rval ml-*v_1) *cupcake-evaluate-single env* exp_2
(*Rval ml-*v_2) **and**
$\exists\, t_1'.\ \Gamma \vdash_v t_1 \downarrow t_1' \land$ *related-v* t_1' *ml-*v_1 $\exists\, t_2'.\ \Gamma \vdash_v t_2 \downarrow t_2' \land$ *related-v* t_2' *ml-*v_2
using *app1* ‹*related-exp* t_1 exp_1› ‹*related-exp* t_2 exp_2›
unfolding ‹*ress* = -› ‹*exps* = -› ‹*ml-vs* = -› ‹t = -›
by (*auto simp*: *closed-except-def*)

then obtain v_1 v_2
where $\Gamma \vdash_v t_1 \downarrow v_1$ *related-v* v_1 *ml-*v_1
and $\Gamma \vdash_v t_2 \downarrow v_2$ *related-v* v_2 *ml-*v_2
by *blast*

have *is-cupcake-value ml-*v_1
  by (*rule cupcake-single-preserve*) *fact+*
moreover have *is-cupcake-value ml-*v_2
  by (*rule cupcake-single-preserve*) *fact+*
ultimately have *list-all is-cupcake-value* (*rev ml-vs*)
  unfolding ‹*ml-vs* = -› **by** *simp*

hence *is-cupcake-exp exp′ is-cupcake-all-env env′*
  using ‹*do-opapp* - = -› **by** (*metis cupcake-opapp-preserve*)+

have *vclosed* v_1
  proof (*rule veval′-closed*)
    show *closed-except* t_1 (*fmdom* Γ)
      using ‹*closed-except* - (*fmdom* Γ)›
      unfolding ‹*t* = -›
      by (*simp add*: *closed-except-def*)
  next
    show *wellformed* t_1
      using ‹*wellformed t*› **unfolding** ‹*t* = -›
      by *simp*
  qed *fact+*
have *vclosed* v_2
  apply (*rule veval′-closed*)
      apply *fact*
  using *app1* **unfolding** ‹*t* = -› **by** (*auto simp*: *closed-except-def*)

have *vwellformed* v_1
  apply (*rule veval′-wellformed*)
    apply *fact*
  using *app1* **unfolding** ‹*t* = -› **by** *auto*
have *vwellformed* v_2
  apply (*rule veval′-wellformed*)
    apply *fact*
  using *app1* **unfolding** ‹*t* = -› **by** *auto*

have *vwelldefined′* v_1
  apply (*rule veval′-welldefined′*)
        apply *fact*
  using *app1* **unfolding** ‹*t* = -› **by** *auto*
have *vwelldefined′* v_2
  apply (*rule veval′-welldefined′*)
        apply *fact*
  using *app1* **unfolding** ‹*t* = -› **by** *auto*

have *not-shadows-vconsts* v_1
  apply (*rule veval′-shadows*)
    apply *fact*
  using *app1* **unfolding** ‹*t* = -› **by** *auto*

have *not-shadows-vconsts* v_2
apply (*rule veval'-shadows*)
apply *fact*
using *app1* **unfolding** ‹t = -› **by** *auto*

show *?case*
proof (*rule if-rvalI*)
fix *ml-v*
assume *bv* = *Rval ml-v*
show $\exists\, v.\ \Gamma \vdash_v t \downarrow v \wedge$ *related-v v ml-v*
using ‹*do-opapp* - = -›
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* (λ-. *nsBind* (*as-string* (*Name n*)) *ml-v*$_2$ (*sem-env.v env*$_\Lambda$)) *env*$_\Lambda$
unfolding ‹*ml-vs* = -› **by** *auto*
obtain Γ_Λ *cs*
where v_1 = *Vabs cs* Γ_Λ *related-fun cs* (*Name n*) *exp'*
and *fmrel-on-fset* (*ids* (*Sabs cs*)) *related-v* Γ_Λ (*fmap-of-ns* (*sem-env.v env*$_\Lambda$))
using ‹*related-v* v_1 *ml-v*$_1$› **unfolding** ‹*ml-v*$_1$ = -›
by *cases auto*

then obtain *ml-cs*
where *exp'* = *Mat* (*Var* (*Short* (*as-string* (*Name n*)))) *ml-cs Name n* |∉| *ids* (*Sabs cs*) *Name n* |∉| *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*) (*Rval ml-v*)
using ‹*cupcake-evaluate-single env' exp' bv*›
unfolding ‹*bv* = -›
by *simp*

then obtain *m-env v' ml-rhs ml-pat*
where *cupcake-evaluate-single env'* (*Var* (*Short* (*as-string* (*Name n*)))) (*Rval v'*)
and *cupcake-match-result* (*sem-env.c env'*) *v' ml-cs Bindv* = *Rval* (*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* (*Rval ml-v*)
by *cases auto*

have
closed-venv (*fmupd* (*Name n*) v_2 Γ_Λ) *wellformed-venv* (*fmupd* (*Name n*) v_2 Γ_Λ)

not-shadows-vconsts-env (*fmupd* (*Name n*) v_2 Γ_Λ) *fmpred* (λ-. *vwelldefined′*) (*fmupd* (*Name n*) v_2 Γ_Λ)
using ‹*vclosed* v_1› ‹*vclosed* v_2›
using ‹*vwellformed* v_1› ‹*vwellformed* v_2›
using ‹*not-shadows-vconsts* v_1› ‹*not-shadows-vconsts* v_2›
using ‹*vwelldefined′* v_1› ‹*vwelldefined′* v_2›
unfolding ‹v_1 = -›
by *auto*

have *closed-except* (*Sabs cs*) (*fmdom* (*fmupd* (*Name n*) v_2 Γ_Λ))
using ‹*vclosed* v_1› **unfolding** ‹v_1 = -›
apply (*auto simp*: *Sterm.closed-except-simps list-all-iff*)
apply (*auto simp*: *closed-except-def*)
done

have *consts* (*Sabs cs*) |⊆| *fmdom* (*fmupd* (*Name n*) v_2 Γ_Λ) |∪| *C*
using ‹*vwelldefined′* v_1› **unfolding** ‹v_1 = -›
unfolding *sconsts-sabs*
by (*auto simp*: *list-all-iff*)

have ¬ *shadows-consts* (*Sabs cs*)
using ‹*not-shadows-vconsts* v_1› **unfolding** ‹v_1 = -›
by (*auto simp*: *list-all-iff list-ex-iff*)

have *fdisjnt C* (*fmdom* Γ_Λ)
using ‹*vwelldefined′* v_1› **unfolding** ‹v_1 = -›
by *simp*

have *if-rval* (λ*ml-v*. ∃ *v*. *fmupd* (*Name n*) v_2 Γ_Λ $\vdash_v$ *Sabs cs* $\$_s$ *Svar* (*Name n*) ↓ *v* ∧ *related-v v ml-v*) *bv*
proof (*rule app1*(*2*))
show *fmrel-on-fset* (*ids* (*Sabs cs* $\$_s$ *Svar* (*Name n*))) *related-v* (*fmupd* (*Name n*) v_2 Γ_Λ) (*fmap-of-ns* (*sem-env.v env′*))
unfolding *closure′*
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*)
apply (*fold ids-def*)
using ‹*fmrel-on-fset* (*ids* (*Sabs cs*)) *related-v* Γ_Λ -› **apply** *simp*
apply (*rule* ‹*related-v* v_2 *ml-*v_2›)
done
next
show *wellformed* (*Sabs cs* $\$_s$ *Svar* (*Name n*))
using ‹*vwellformed* v_1› **unfolding** ‹v_1 = -›
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 Γ_Λ))
using ‹*closed-except* (*Sabs cs*) (*fmdom* (*fmupd* (*Name n*) v_2 Γ_Λ))› **by** (*simp add*: *closed-except-def*)
next
show ¬ *shadows-consts* (*Sabs cs* $\$_s$ *Svar* (*Name n*))
using ‹¬ *shadows-consts* (*Sabs cs*)› ‹*Name n* |∉| *all-consts*› **by** *simp*
next
show *consts* (*Sabs cs* $\$_s$ *Svar* (*Name n*)) |⊆| *fmdom* (*fmupd* (*Name n*) v_2 Γ_Λ) |∪| *C*
using ‹*consts* (*Sabs cs*) |⊆| *fmdom* (*fmupd* (*Name n*) v_2 Γ_Λ) |∪| *C*› **by** *simp*
next
show *fdisjnt C* (*fmdom* (*fmupd* (*Name n*) v_2 Γ_Λ))
using ‹*Name n* |∉| *all-consts*› ‹*fdisjnt C* (*fmdom* Γ_Λ)›
unfolding *fdisjnt-alt-def all-consts-def* **by** *auto*
qed *fact*+
then obtain *v* **where** *fmupd* (*Name n*) v_2 Γ_Λ $\vdash_v$ *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* v_2 = *Some* (*env*, *pat*, *rhs*)
and *fmupd* (*Name n*) v_2 Γ_Λ $++_f$ *env* $\vdash_v$ *rhs* ↓ *v*
by (*auto elim*: *veval′-sabs-svarE*)
hence (*pat*, *rhs*) ∈ *set cs vmatch* (*mk-pat pat*) v_2 = *Some env*
by (*metis vfind-match-elem*)+
hence *linear pat wellformed rhs*
using ‹*vwellformed* v_1› **unfolding** ‹v_1 = -›
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*) v_2 = *Some env*›)
done

obtain *v′* **where** Γ_Λ $++_f$ *env* $\vdash_v$ *rhs* ↓ *v′* *v′* $\approx_e$ *v*
proof (*rule veval′-agree-eq*)
show *fmupd* (*Name n*) v_2 Γ_Λ $++_f$ *env* $\vdash_v$ *rhs* ↓ *v* **by** *fact*
next
have ∗: *Name n* |∉| *ids rhs* **if** *Name n* |∉| *fmdom env*
proof
assume *Name n* |∈| *ids rhs*
thus *False*

```
          unfolding ids-def
          proof (cases rule: funion-strictE)
            case A
            moreover have Name n |∉| frees pat
              using that unfolding ‹fmdom env = frees pat› .
            ultimately have Name n |∈| frees (Sabs cs)
              apply auto
              unfolding ffUnion-alt-def
              apply simp
              apply (rule fBexI[where x = (pat, rhs)])
               apply (auto simp: fset-of-list-elem)
              apply (rule ‹(pat, rhs) ∈ set cs›)
              done
            thus ?thesis
              using ‹Name n |∉| ids (Sabs cs)› unfolding ids-def
              by blast
          next
            case B
            hence Name n |∈| consts (Sabs cs)
              apply auto
              unfolding ffUnion-alt-def
              apply simp
              apply (rule fBexI[where x = (pat, rhs)])
               apply (auto simp: fset-of-list-elem)
              apply (rule ‹(pat, rhs) ∈ set cs›)
              done
            thus ?thesis
              using ‹Name n |∉| ids (Sabs cs)› unfolding ids-def
              by blast
          qed
        qed

      show fmrel-on-fset (ids rhs) erelated (Γ_Λ ++_f env) (fmupd (Name n)
v_2 Γ_Λ ++_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 Γ_Λ ++_f env)
        apply rule
         apply fact
        apply (rule vclosed.vmatch-env)
         apply fact
        apply fact
        done
    next
```

```
          show wellformed-venv (fmupd (Name n) v₂ Γ_Λ ++_f env)
            apply rule
             apply fact
            apply (rule vwellformed.vmatch-env)
             apply fact
            apply fact
            done
        next
          show closed-except rhs (fmdom (fmupd (Name n) v₂ Γ_Λ ++_f env))
            using ‹fmdom env = frees pat› ‹(pat, rhs) ∈ set cs›
            using ‹closed-except (Sabs cs) (fmdom (fmupd (Name n) v₂ Γ_Λ))›
            by (auto simp: Sterm.closed-except-simps list-all-iff)
        next
          show wellformed rhs by fact
        next
          show consts rhs |⊆| fmdom (fmupd (Name n) v₂ Γ_Λ ++_f env) |∪| C
             using ‹consts (Sabs cs) |⊆| fmdom (fmupd (Name n) v₂ Γ_Λ) |∪| C›
‹(pat, rhs) ∈ set cs›
            unfolding sconsts-sabs
            by (auto simp: list-all-iff)
        next
          have fdisjnt C (fmdom env)
            using ‹(pat, rhs) ∈ set cs› ‹¬ shadows-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) v₂ Γ_Λ ++_f env))
            using ‹Name n |∉| all-consts› ‹fdisjnt C (fmdom Γ_Λ)›
            unfolding fdisjnt-alt-def
            by (auto simp: all-consts-def)
        next
          show ¬ shadows-consts rhs
            using ‹(pat, rhs) ∈ set cs› ‹¬ shadows-consts (Sabs cs)›
            by (auto simp: list-ex-iff)
        next
          have not-shadows-vconsts-env env
            by (rule not-shadows-vconsts.vmatch-env) fact+
          thus not-shadows-vconsts-env (fmupd (Name n) v₂ Γ_Λ ++_f env)
            using ‹not-shadows-vconsts-env (fmupd (Name n) v₂ Γ_Λ)› by blast
        next
          have fmpred (λ-. vwelldefined') env
            by (rule vmatch-welldefined') fact+
          thus fmpred (λ-. vwelldefined') (fmupd (Name n) v₂ Γ_Λ ++_f env)
            using ‹fmpred (λ-. vwelldefined') (fmupd (Name n) v₂ Γ_Λ)› by blast
        qed blast

      show ?thesis
        apply (intro exI conjI)
        unfolding ‹t = -›
         apply (rule veval'.comb)
```

using ‹$\Gamma \vdash_v t_1 \downarrow v_1$› **unfolding** ‹$v_1$ = -›
apply *blast*
apply *fact*
apply *fact+*
apply (*rule related-v-ext*)
apply *fact+*
done
next
case (*recclosure env*$_\Lambda$ *funs name n*)
with *recclosure* **have** *recclosure'*:
ml-v$_1$ = *Recclosure env*$_\Lambda$ *funs name*
env' = *update-v* (λ-. *nsBind* (*as-string* (*Name n*)) *ml-v*$_2$ (*build-rec-env funs env*$_\Lambda$ (*sem-env.v env*$_\Lambda$))) *env*$_\Lambda$
unfolding ‹*ml-vs* = -› **by** *auto*
obtain Γ_Λ *css*
where v_1 = *Vrecabs css* (*Name name*) Γ_Λ
and *fmrel-on-fset* (*fbind* (*fmran css*) (*ids* ∘ *Sabs*)) *related-v* Γ_Λ (*fmap-of-ns* (*sem-env.v env*$_\Lambda$))
and *fmrel* (λ*cs* (*n*, *e*). *related-fun cs n e*) *css* (*fmap-of-list* (*map* (*map-prod Name* (*map-prod Name id*)) *funs*))
using ‹*related-v* v_1 *ml-v*$_1$› **unfolding** ‹*ml-v*$_1$ = -›
by *cases auto*
from ‹*fmrel* - - -› **have** *rel-option* (λ*cs* (*n*, *e*). *related-fun cs* (*Name n*) *e*) (*fmlookup css* (*Name name*)) (*find-recfun name funs*)
apply −
apply (*subst option.rel-sel*)
apply *auto*
apply (*drule fmrel-fmdom-eq*)
apply (*drule fmdom-notI*)
using ‹v_1 = *Vrecabs css* (*Name name*) Γ_Λ› ‹*vwellformed* v_1› **apply** *auto*[*1*]
using *recclosure*(*3*) **apply** *auto*[*1*]
apply (*erule fmrel-cases*[**where** *x* = *Name name*])
apply *simp*
apply (*subst* (*asm*) *fmlookup-of-list*)
apply (*simp add*: *name.map-of-rekey'*)
by *blast*

then obtain *cs* **where** *fmlookup css* (*Name name*) = *Some cs related-fun cs* (*Name n*) *exp'*
using ‹*find-recfun* - - = -›
by *cases auto*

then obtain *ml-cs*
where *exp'* = *Mat* (*Var* (*Short* (*as-string* (*Name n*)))) *ml-cs Name n* |∉| *ids* (*Sabs cs*) *Name n* |∉| *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 n*)) *ml-cs*) (*Rval ml-v*)
using ‹*cupcake-evaluate-single env' exp' bv*›
unfolding ‹*bv* = -›
by *simp*

then obtain *m-env v' ml-rhs ml-pat*
where *cupcake-evaluate-single env'* (*Var* (*Short n*)) (*Rval v'*)
and *cupcake-match-result* (*sem-env.c env'*) *v' ml-cs Bindv* = *Rval* (*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* (*Rval ml-v*)
by *cases auto*

have *closed-venv* (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ))
using ‹*vclosed* v_1› ‹*vclosed* v_2›
using ‹*fmlookup css* (*Name name*) = *Some cs*›
unfolding ‹v_1 = -› *mk-rec-env-def*
apply *auto*
apply *rule*
apply *rule*
apply (*auto intro*: *fmdomI*)
done
have *wellformed-venv* (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ))
using ‹*vwellformed* v_1› ‹*vwellformed* v_2›
using ‹*fmlookup css* (*Name name*) = *Some cs*›
unfolding ‹v_1 = -› *mk-rec-env-def*
apply *auto*
apply *rule*
apply *rule*
apply (*auto intro*: *fmdomI*)
done
have *not-shadows-vconsts-env* (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ))
using ‹*not-shadows-vconsts* v_1› ‹*not-shadows-vconsts* v_2›
using ‹*fmlookup css* (*Name name*) = *Some cs*›
unfolding ‹v_1 = -› *mk-rec-env-def*
apply *auto*
apply *rule*
apply *rule*
apply (*auto intro*: *fmdomI*)
done
have *fmpred* (λ-. *vwelldefined'*) (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ))
using ‹*vwelldefined'* v_1› ‹*vwelldefined'* v_2›
using ‹*fmlookup css* (*Name name*) = *Some cs*›
unfolding ‹v_1 = -› *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 Γ_Λ))
using ‹*vclosed* v_1› **unfolding** ‹v_1 = -›
apply (*auto simp*: *Sterm.closed-except-simps list-all-iff*)
using ‹*fmlookup css* (*Name name*) = *Some cs*›
apply (*auto simp*: *closed-except-def dest*!: *fmpredD*[**where** *m* = *css*])
done

have *consts* (*Sabs cs*) |⊆| *fmdom* (*fmupd* (*Name n*) v_2 Γ_Λ) |∪| (*C* |∪| *fmdom css*)
using ‹*vwelldefined′* v_1› **unfolding** ‹v_1 = -›
unfolding *sconsts-sabs*
using ‹*fmlookup css* (*Name name*) = *Some cs*›
by (*auto simp*: *list-all-iff dest*!: *fmpredD*[**where** *m* = *css*])

have ¬ *shadows-consts* (*Sabs cs*)
using ‹*not-shadows-vconsts* v_1› **unfolding** ‹v_1 = -›
using ‹*fmlookup css* (*Name name*) = *Some cs*›
by (*auto simp*: *list-all-iff list-ex-iff*)

have *fdisjnt C* (*fmdom* Γ_Λ)
using ‹*vwelldefined′* v_1› **unfolding** ‹v_1 = -›
using ‹*fmlookup css* (*Name name*) = *Some cs*›
by *auto*

have *if-rval* (*λml-v*. ∃ *v*. *fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ) $\vdash_v$ *Sabs cs* $\$_s$ *Svar* (*Name n*) ↓ *v* ∧ *related-v v ml-v*) *bv*
proof (*rule app1*(*2*))
have *fmrel-on-fset* (*ids* (*Sabs cs*)) *related-v* Γ_Λ (*fmap-of-ns* (*sem-env.v* env_Λ))
apply (*rule fmrel-on-fsubset*)
apply *fact*
apply (*subst funion-image-bind-eq*[*symmetric*])
apply (*rule ffUnion-subset-elem*)
apply (*subst fimage-iff*)
apply (*rule fBexI*)
apply *simp*
apply (*rule fmranI*)
apply *fact*
done

have *fmrel-on-fset* (*ids* (*Sabs cs*)) *related-v* (*mk-rec-env css* Γ_Λ) (*cake-mk-rec-env funs* env_Λ)
apply *rule*
apply (*rule mk-rec-env-related*[*THEN fmrelD*])
apply (*rule* ‹*fmrel - css -*›)

apply (*rule* ‹*fmrel-on-fset* (*fbind* - -) *related-v* Γ_Λ -›)
done

show *fmrel-on-fset* (*ids* (*Sabs cs* $\$_s$ *Svar* (*Name n*))) *related-v* (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ)) (*fmap-of-ns* (*sem-env.v env'*))
unfolding *recclosure'*
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_Λ -› **by** *simp*
subgoal
using ‹*fmrel-on-fset* (*ids* (*Sabs cs*)) *related-v* (*mk-rec-env css* Γ_Λ) -› **by** *simp*
apply (*rule* ‹*related-v* v_2 *ml-*v_2›)
done
next
show *wellformed* (*Sabs cs* $\$_s$ *Svar* (*Name n*))
using ‹*vwellformed* v_1› **unfolding** ‹v_1 = -›
using ‹*fmlookup css* (*Name name*) = *Some cs*›
by *auto*
next
show *related-exp* (*Sabs cs* $\$_s$ *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* $\$_s$ *Svar* (*Name n*)) (*fmdom* (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ)))
using ‹*closed-except* (*Sabs cs*) (*fmdom* (*fmupd* (*Name n*) v_2 Γ_Λ))›
by (*auto simp*: *list-all-iff closed-except-def*)
next
show ¬ *shadows-consts* (*Sabs cs* $\$_s$ *Svar* (*Name n*))
using ‹¬ *shadows-consts* (*Sabs cs*)› ‹*Name n* |∉| *all-consts*› **by** *simp*
next
show *consts* (*Sabs cs* $\$_s$ *Svar* (*Name n*)) |⊆| *fmdom* (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ)) |∪| *C*
using ‹*consts* (*Sabs cs*) |⊆| -› **unfolding** *mk-rec-env-def*
by *auto*
next
show *fdisjnt C* (*fmdom* (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ)))
using ‹*Name n* |∉| *all-consts*› ‹*fdisjnt C* (*fmdom* Γ_Λ)› ‹*vwelldefined'* v_1›

unfolding *mk-rec-env-def* ‹v_1 = -›
by (*auto simp*: *fdisjnt-alt-def all-consts-def*)
qed *fact+*
then obtain *v*
where *fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ) $\vdash_v$ *Sabs cs* $\$_s$ *Svar* (*Name n*) $\downarrow$ *v related-v v ml-v*
unfolding ‹*bv* = -›
by *auto*

then obtain *env pat rhs*
where *vfind-match cs* v_2 = *Some* (*env*, *pat*, *rhs*)
and *fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ) $++_f$ *env* $\vdash_v$ *rhs* $\downarrow$ *v*
by (*auto elim*: *veval′-sabs-svarE*)
hence (*pat*, *rhs*) $\in$ *set cs vmatch* (*mk-pat pat*) v_2 = *Some env*
by (*metis vfind-match-elem*)+
hence *linear pat wellformed rhs*
using ‹*vwellformed* v_1› **unfolding** ‹v_1 = -›
using ‹*fmlookup css* (*Name name*) = *Some 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* ‹*vmatch* (*mk-pat pat*) v_2 = *Some env*›)
done

obtain v' **where** Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ $++_f$ *env* $\vdash_v$ *rhs* $\downarrow$ v' v' $\approx_e$ *v*
proof (*rule veval′-agree-eq*)
show *fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ) $++_f$ *env* $\vdash_v$ *rhs* $\downarrow$ *v* **by** *fact*
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*[**where** *x* = (*pat*, *rhs*)])
apply (*auto simp*: *fset-of-list-elem*)
apply (*rule* ‹(*pat*, *rhs*) $\in$ *set cs*›)

done
thus *?thesis*
using ‹*Name n* |∉| *ids* (*Sabs cs*)› **unfolding** *ids-def*
by *blast*
next
case *B*
hence *Name n* |∈| *consts* (*Sabs cs*)
apply *auto*
unfolding *ffUnion-alt-def*
apply *simp*
apply (*rule fBexI*[**where** x = (*pat*, *rhs*)])
apply (*auto simp*: *fset-of-list-elem*)
apply (*rule* ‹(*pat*, *rhs*) ∈ *set cs*›)
done
thus *?thesis*
using ‹*Name n* |∉| *ids* (*Sabs cs*)› **unfolding** *ids-def*
by *blast*
qed
qed

show *fmrel-on-fset* (*ids rhs*) *erelated* (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ $++_f$ *env*) (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ) $++_f$ *env*)
apply *rule*
apply *auto*
apply (*rule option.rel-refl*; *rule erelated-refl*)
using ∗ **apply** *auto*[]
apply (*rule option.rel-refl*; *rule erelated-refl*)+
using ∗ **apply** *auto*[]
apply (*rule option.rel-refl*; *rule erelated-refl*)+
done
next
show *closed-venv* (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ) $++_f$ *env*)
apply *rule*
apply *fact*
apply (*rule vclosed.vmatch-env*)
apply *fact*
apply *fact*
done
next
show *wellformed-venv* (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ) $++_f$ *env*)
apply *rule*
apply *fact*
apply (*rule vwellformed.vmatch-env*)
apply *fact*
apply *fact*
done
next

show *closed-except rhs* (*fmdom* (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ) $++_f$ *env*))
using ‹*fmdom env* = *frees pat*› ‹(*pat*, *rhs*) ∈ *set cs*›
using ‹*closed-except* (*Sabs cs*) (*fmdom* (*fmupd* (*Name n*) v_2 Γ_Λ))›
apply (*auto simp*: *Sterm.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*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ) $++_f$ *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 ‹(*pat*, *rhs*) ∈ *set cs*› ‹¬ *shadows-consts* (*Sabs cs*)›
unfolding ‹*fmdom env* = *frees pat*›
by (*auto simp*: *list-ex-iff all-consts-def fdisjnt-alt-def*)
moreover have *fdisjnt C* (*fmdom css*)
using ‹*vwelldefined′* v_1› **unfolding** ‹v_1 = -›
by *simp*
ultimately show *fdisjnt C* (*fmdom* (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ) $++_f$ *env*))
using ‹*Name n* |∉| *all-consts*› ‹*fdisjnt C* (*fmdom* Γ_Λ)›
unfolding *fdisjnt-alt-def mk-rec-env-def*
by (*auto simp*: *all-consts-def*)
next
show ¬ *shadows-consts rhs*
using ‹(*pat*, *rhs*) ∈ *set cs*› ‹¬ *shadows-consts* (*Sabs cs*)›
by (*auto simp*: *list-ex-iff*)
next
have *not-shadows-vconsts-env env*
by (*rule not-shadows-vconsts.vmatch-env*) *fact*+
thus *not-shadows-vconsts-env* (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ) $++_f$ *env*)
using ‹*not-shadows-vconsts-env* (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ))› **by** *blast*
next
have *fmpred* (λ-. *vwelldefined′*) *env*
by (*rule vmatch-welldefined′*) *fact*+
thus *fmpred* (λ-. *vwelldefined′*) (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ) $++_f$ *env*)
using ‹*fmpred* (λ-. *vwelldefined′*) (*fmupd* (*Name n*) v_2 (Γ_Λ $++_f$ *mk-rec-env css* Γ_Λ))› **by** *blast*
qed *simp*

```
        show ?thesis
          apply (intro exI conjI)
          unfolding ‹t = -›
           apply (rule veval'.rec-comb)
          using ‹Γ ⊢v t1 ↓ v1› unfolding ‹v1 = -› apply blast
              apply fact+
          apply (rule related-v-ext)
           apply fact+
          done
      qed
    qed
next
  case (mat1 env ml-scr ml-scr' ml-cs ml-rhs ml-pat env' ml-res)

  obtain scr cs
    where t = Sabs cs $s scr related-exp scr ml-scr
      and list-all2 (rel-prod related-pat related-exp) cs ml-cs
    using ‹related-exp t (Mat ml-scr ml-cs)›
    by cases

  have sem-env.c env = as-static-cenv
    using ‹is-cupcake-all-env env›
    by (auto elim: is-cupcake-all-envE)

  obtain scr' where Γ ⊢v scr ↓ scr' related-v scr' ml-scr'
    using mat1(4) unfolding if-rval.simps
    proof
      show
        is-cupcake-all-env env related-exp scr ml-scr wellformed-venv Γ
        closed-venv Γ fmpred (λ-. vwelldefined') Γ fdisjnt C (fmdom Γ)
        not-shadows-vconsts-env Γ
        by fact+
    next
      show fmrel-on-fset (ids scr) related-v Γ (fmap-of-ns (sem-env.v env))
        apply (rule fmrel-on-fsubset)
         apply fact
        unfolding ‹t = -› ids-def
        apply auto
        done
    next
      show
        wellformed scr consts scr |⊆| fmdom Γ |∪| C
        closed-except scr (fmdom Γ) ¬ shadows-consts scr
        using mat1 unfolding ‹t = -› by (auto simp: closed-except-def)
    qed blast

  have list-all (λ(pat, -). linear pat) cs
    using mat1 unfolding ‹t = -› by (auto simp: list-all-iff)
```

```
  obtain rhs pat Γ'
    where ml-pat = mk-ml-pat (mk-pat pat) related-exp rhs ml-rhs
      and vfind-match cs scr' = Some (Γ', pat, rhs)
      and var-env Γ' env'
    using ‹list-all2 - cs ml-cs› ‹list-all - cs› ‹related-v scr' ml-scr'› mat1(2)
    unfolding ‹sem-env.c env = as-static-cenv›
    by (auto elim: match-all-related)

  hence vmatch (mk-pat pat) scr' = Some Γ'
    by (auto dest: vfind-match-elem)
  hence patvars (mk-pat pat) = fmdom Γ'
    by (auto simp: vmatch-dom)

  have (pat, rhs) ∈ set cs
    by (rule vfind-match-elem) fact

  have linear pat
    using ‹(pat, rhs) ∈ set cs› ‹wellformed t› unfolding ‹t = -›
    by (auto simp: list-all-iff)
  hence fmdom Γ' = frees pat
    using ‹patvars (mk-pat pat) = fmdom Γ'›
    by (simp add: mk-pat-frees)

  show ?case
    proof (rule if-rvalI)
      fix ml-rhs'
      assume ml-res = Rval ml-rhs'

      obtain rhs' where Γ ++f Γ' ⊢v rhs ↓ rhs' related-v rhs' ml-rhs'
        using mat1(5) unfolding ‹ml-res = -› if-rval.simps
        proof
          show is-cupcake-all-env (env (| sem-env.v := nsAppend (alist-to-ns env')
(sem-env.v env) |))
            proof (rule cupcake-v-update-preserve)
              have list-all (is-cupcake-value ∘ snd) env'
                proof (rule match-all-preserve)
                  show cupcake-c-ns (sem-env.c env)
                    unfolding ‹sem-env.c env = -› 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
                    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 fmrel-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*
apply (*rule fBexI*[**where** *x* = (*pat*, *rhs*)])
apply (*auto simp*: *fset-of-list-elem*)
done
thus *id* |∈| *frees t* |∪| *consts t* **by** *simp*
qed

```
have fmrel-on-fset (ids rhs) related-v (Γ ++_f Γ′) (fmap-of-ns (sem-env.v
env) ++_f fmap-of-list (map (map-prod Name id) env′))
    apply rule
    apply simp
    apply safe
    subgoal
      apply (rule fmrelD)
      apply (rule ‹var-env Γ′ env′›)
      done
    subgoal
      unfolding *[symmetric]
  using fmdom-of-list fset-of-list-map fset.set-map image-image fst-map-prod
      by simp
    subgoal using *
    by (metis (no-types, opaque-lifting) comp-def fimageI fmdom-fmap-of-list
fset-of-list-map fst-comp-map-prod)
    subgoal using **
      by (metis fmlookup-ns fmrel-on-fsetD mat1.prems(2))
    done

  thus fmrel-on-fset (ids rhs) related-v (Γ ++_f Γ′) (fmap-of-ns (sem-env.v
(env (| sem-env.v := nsAppend (alist-to-ns env′) (sem-env.v env) |))))
    by (auto split: sem-env.splits)
next
  show wellformed-venv (Γ ++_f Γ′)
    apply rule
     apply fact
    apply (rule vwellformed.vmatch-env)
     apply fact
    apply (rule veval′-wellformed)
      apply fact
    using mat1 unfolding ‹t = -› by auto
next
  show closed-venv (Γ ++_f Γ′)
    apply rule
     apply fact
    apply (rule vclosed.vmatch-env)
     apply fact
    apply (rule veval′-closed)
        apply fact
    using mat1 unfolding ‹t = -› by (auto simp: closed-except-def)
next
  show fmpred (λ-. vwelldefined′) (Γ ++_f Γ′)
    apply rule
     apply fact
    apply (rule vmatch-welldefined′)
     apply fact
    apply (rule veval′-welldefined′)
          apply fact
```

using *mat1* **unfolding** ‹t = -› **by** *auto*
next
show *not-shadows-vconsts-env* $(\Gamma \mathbin{++_f} \Gamma')$
apply *rule*
apply *fact*
apply (*rule not-shadows-vconsts.vmatch-env*)
apply *fact*
apply (*rule veval′-shadows*)
apply *fact*
using *mat1* **unfolding** ‹t = -› **by** *auto*
next
show *wellformed rhs*
using ‹(*pat*, *rhs*) $\in$ *set cs*› ‹*wellformed t*› **unfolding** ‹t = -›
by (*auto simp*: *list-all-iff*)
next
show *closed-except rhs* (*fmdom* $(\Gamma \mathbin{++_f} \Gamma')$)
apply *simp*
unfolding ‹*fmdom* Γ' = *frees pat*›
using ‹(*pat*, *rhs*) $\in$ *set cs*› ‹*closed-except t* (*fmdom* Γ)› **unfolding** ‹t =
-›
by (*auto simp*: *Sterm.closed-except-simps list-all-iff*)
next
have *consts* (*Sabs cs*) $|\subseteq|$ *fmdom* Γ $|\cup|$ C
using *mat1* **unfolding** ‹t = -› **by** *auto*
show *consts rhs* $|\subseteq|$ *fmdom* $(\Gamma \mathbin{++_f} \Gamma')$ $|\cup|$ C
apply *simp*
unfolding ‹*fmdom* Γ' = *frees pat*›
using ‹(*pat*, *rhs*) $\in$ *set cs*› ‹*consts* (*Sabs cs*) $|\subseteq|$ -›
unfolding *sconsts-sabs*
by (*auto simp*: *list-all-iff*)
next
have *fdisjnt* C (*fmdom* Γ')
unfolding ‹*fmdom* Γ' = *frees pat*›
using ‹$\neg$ *shadows-consts t*› ‹(*pat*, *rhs*) $\in$ *set cs*›
unfolding ‹t = -›
by (*auto simp*: *list-ex-iff fdisjnt-alt-def all-consts-def*)

thus *fdisjnt* C (*fmdom* $(\Gamma \mathbin{++_f} \Gamma')$)
using ‹*fdisjnt* C (*fmdom* Γ)›
unfolding *fdisjnt-alt-def* **by** *auto*
next
show $\neg$ *shadows-consts rhs*
using ‹(*pat*, *rhs*) $\in$ *set cs*› ‹$\neg$ *shadows-consts t*› **unfolding** ‹t = -›
by (*auto simp*: *list-ex-iff*)
qed *blast*

show $\exists\, t'.\ \Gamma \vdash_v t \downarrow t' \wedge$ *related-v* t' *ml-rhs′*
unfolding ‹t = -›
apply (*intro exI conjI seval.intros*)

apply (*rule veval′.intros*)
apply (*rule veval′.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-venv* Γ
assumes *closed-venv* Γ *closed-except t* (*fmdom* Γ)
assumes *fmpred* (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** $\Gamma \vdash_v t \downarrow v$ *related-v v ml-v*
using *semantic-correctness0*[*OF assms*]
by *auto*

end 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* (λ*n*. *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 = *Ast.App* (*Ast.Opn Ast.Times*) [*t*, *u*]

lemma *cake-times-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-times t u*) = (*s*′′, *Rval* (*Litv* (*IntLit* (*x* ∗ *y*))))
unfolding *cake-times-def* **using** *assms* **by** *simp*

definition *cake-int-of-bool* :: *exp* ⇒ *exp* **where**
cake-int-of-bool e = *Ast.Mat e*
  [(*Ast.Pcon* (*Some* (*Short* ′′*HOL-False*′′)) [], *Lit* (*IntLit 0*)),
   (*Ast.Pcon* (*Some* (*Short* ′′*HOL-True*′′)) [], *Lit* (*IntLit 1*))]

definition *summands* :: *exp list* **where**
summands = *map* (λ*n*. *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 =
  Ast.Fun ′′*x*′′ (*Ast.Mat* (*Ast.Var* (*Short* ′′*x*′′)) [(*pat*, *foldl cake-plus* (*Lit* (*IntLit* *0*)) *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*
begin

hide-const (**open**) *sem-env.v*

Term-Class.term $\longrightarrow$ *nterm* $\longrightarrow$ *pterm* $\longrightarrow$ *sterm*

6.1.1 Reflexive-transitive closure of *irules.compile-correct*.

lemma (**in** *prules*) *prewrite-closed*:
  assumes $rs \vdash_p t \longrightarrow t'$ *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*)› *beta* **by** (*auto simp*: *closed-except-def*)
qed (*auto simp*: *closed-except-def*)

corollary (**in** *prules*) *prewrite-rt-closed*:
assumes $rs \vdash_p t \longrightarrow* t'$ *closed t*
shows *closed* t'
using *assms*
by *induction* (*auto intro*: *prewrite-closed*)

corollary (**in** *irules*) *compile-correct-rt*:
assumes *Rewriting-Pterm.compile* $rs \vdash_p t \longrightarrow* t'$ *finished rs*
shows $rs \vdash_i t \longrightarrow* t'$
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* $rs \vdash_s u \longrightarrow* u'$
shows $rs \vdash_p$ *sterm-to-pterm* $u \longrightarrow*$ *sterm-to-pterm* u'
using *assms* **proof** *induction*
case *step*
thus *?case*
by (*meson compile-correct rtranclp.simps*)
qed *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 $rs \vdash_s t \longrightarrow t'$ *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*) $\in$ *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* $\vdash_s t \longrightarrow * t'$ *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*) $\longleftrightarrow$ *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* $\vdash_n u \longrightarrow * u'$ *closed u*
shows *rs* $\vdash$ *nterm-to-term'* $u \longrightarrow *$ *nterm-to-term'* u'
using *assms*
proof *induction*
case (*step* u' u'')
hence *rs* $\vdash$ *nterm-to-term'* $u \longrightarrow *$ *nterm-to-term'* u'
by *auto*
also have *rs* $\vdash$ *nterm-to-term'* $u' \longrightarrow$ *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* $\vdash_i u \longrightarrow * u''$ $t \approx_p u$ *closed t*
obtains t'' **where** *rs* $\vdash_i t \longrightarrow * t''$ $t'' \approx_p u''$
using *assms* **proof** (*induction arbitrary*: *thesis t*)
case (*step* u' u'')

obtain t' **where** *rs* $\vdash_i t \longrightarrow * t'$ $t' \approx_p u'$
using *step* **by** *blast*

obtain t'' **where** *rs* $\vdash_i t' \longrightarrow * t''$ $t'' \approx_p u''$
apply (*rule transform-correct*)
apply (*rule* ‹*transform-irule-set rs* $\vdash_i u' \longrightarrow u''$›)

apply (*rule* ‹$t' \approx_p u'$›)
apply (*rule irewrite-rt-closed*)
apply (*rule* ‹*rs* $\vdash_i$ $t \longrightarrow* t'$›)
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* $\vdash_i$ $t \longrightarrow* u$ *closed t no-abs u*
shows *rs* $\vdash_i$ $t \longrightarrow* u$
proof −
have $t \approx_p t$ **by** (*rule prelated-refl*)
obtain t' **where** *rs* $\vdash_i$ $t \longrightarrow* t'$ $t' \approx_p u$
apply (*rule transform-correct-rt*)
apply (*rule assms*)
apply (*rule* ‹$t \approx_p t$›)
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* $\vdash_i$ $t \longrightarrow* u$ *closed t no-abs u*
shows *rs* $\vdash_i$ $t \longrightarrow* u$
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* $\vdash_i$ $t \longrightarrow* u$ *closed t no-abs u*
shows *rs* $\vdash_i$ $t \longrightarrow* u$
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 *transform-irule-set*.

definition *max-arity* :: *irule-set* ⇒ *nat* **where**
max-arity rs = *fMax* ((*arity* ∘ *snd*) |'| *rs*)

lemma *rules-transform-iter0*:
  assumes *irules C-info rs*
  shows *irules C-info* ((*transform-irule-set* ^^ *n*) *rs*)
using *assms*
by (*induction n*) (*auto intro*: *irules.rules-transform del*: *irulesI*)

lemma (**in** *irules*) *rules-transform-iter*: *irules C-info* ((*transform-irule-set* ^^ *n*) *rs*)
by (*rule rules-transform-iter0*) (*rule irules-axioms*)

lemma *transform-irule-set-n-heads*: *fst* |'| ((*transform-irule-set* ^^ *n*) *rs*) = *fst* |'| *rs*
by (*induction n*) (*auto simp*: *transform-irule-set-heads*)

hide-fact *rules-transform-iter0*

definition *transform-irule-set-iter* :: *irule-set* ⇒ *irule-set* **where**
transform-irule-set-iter rs = (*transform-irule-set* ^^ *max-arity rs*) *rs*

lemma *transform-irule-set-iter-heads*: *fst* |'| *transform-irule-set-iter rs* = *fst* |'| *rs*
unfolding *transform-irule-set-iter-def* **by** (*simp add*: *transform-irule-set-n-heads*)

lemma (**in** *irules*) *finished-alt-def*: *finished rs* ⟷ *max-arity rs* = *0*
proof
  assume *max-arity rs* = *0*
  hence ¬ *fBex* ((*arity* ∘ *snd*) |'| *rs*) (λ*x*. *0* < *x*)
    using *nonempty*
    unfolding *max-arity-def*
    by (*metis fBex-fempty fmax-ex-gr not-less0*)
  thus *finished rs*
    unfolding *finished-def*
    by *force*
next
  assume *finished rs*
  have *fMax* ((*arity* ∘ *snd*) |'| *rs*) ≤ *0*
    proof (*rule fMax-le*)
      show *fBall* ((*arity* ∘ *snd*) |'| *rs*) (λ*x*. *x* ≤ *0*)
        using ‹*finished rs*› **unfolding** *finished-def* **by** *force*
    next
      show (*arity* ∘ *snd*) |'| *rs* ≠ {||}
        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* $\Longrightarrow$ *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 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* ∘ *snd*) |‘| *transform-irule-set rs* = (λ*x*. *x* − *1*) |‘| (*arity* ∘ *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*) |∈| *rs*
hence *arity-compatibles irs irs* ≠ {||}
using *nonempty inner*
unfolding *atomize-conj*
by (*smt* (*verit*, *del-insts*) ‹(*name*, *irs*) |∈| *rs*› *case-prodD fBallE inner*)
thus *arity* (*transform-irules irs*) = *arity irs* − *1*
by (*simp add*: *arity-transform-irules*)
qed
hence *max-arity* (*transform-irule-set rs*) = *fMax* ((λ*x*. *x* − *1*) |‘| (*arity* ∘ *snd*) |‘| *rs*)
unfolding *max-arity-def* **by** *simp*
also have ... = *fMax* ((*arity* ∘ *snd*) |‘| *rs*) − *1*
proof (*rule fmax-decr*)
show *fBex* ((*arity* ∘ *snd*) |‘| *rs*) ((≤) *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 (*subst irules.finished-alt-def*)
(*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'* = *irules*

sublocale *irules'* ⊆ *irules'-as-irules*: *irules C-info transform-irule-set-iter rs*
unfolding *transform-irule-set-iter-def* **by** (*rule rules-transform-iter*)

sublocale *crules* ⊆ *crules-as-irules'*: *irules' C-info Rewriting-Pterm-Elim.compile rs*
unfolding *irules'-def* **by** (*fact compile-rules*)

sublocale *irules'* ⊆ *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* :: (*name*, *sclauses*) *fmap* **where**
global-css = *fmap-of-list* (*map* (*map-prod id clauses*) *rs*)

lemma *fmdom-global-css*: *fmdom global-css* = *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 fmempty*)) *rs*

lemma *as-vrules-fst*[*simp*]: *fst* |'| *fset-of-list as-vrules* = *fst* |'| *fset-of-list rs*
unfolding *as-vrules-def*
apply *simp*
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 = Vrecabs 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
    using ‹(name, rhs) ∈ set as-vrules› unfolding as-vrules-def
    including fset.lifting apply transfer'
    by force

  moreover have R (fmdom global-css)
    using assms unfolding global-css-def
    by auto

  ultimately show value-pred.pred P Q R rhs
    by (simp add: value-pred.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
    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 (λ(-, t). ¬ 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 |`| fset-of-list rs) C
    using disjnt by simp
next
  show list-all (λ(-, rhs). 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 (λ(-, rhs). 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 auto
    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*, *fmempty* $\vdash_v$ *t* $\downarrow$ *v wellformed t closed t*
  shows *rs*, *fmempty* $\vdash_s$ *t* $\downarrow$ *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 fmempty*
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* (‹$\vdash_v$/ - ≈ -› [*0*, *50*] *50*)

lemma *vrecabs-global-css-refl*:
  assumes *name* |∈| *fmdom global-css*
  shows $\vdash_v$ *Vrecabs global-css name fmempty* ≈ *Vrecabs global-css name fmempty*
using *assms*
proof (*coinduction arbitrary*: *name*)
  case *vrelated*
  have *rel-option* (λv_1 v_2. (∃ *name*. v_1 = *Vrecabs global-css name fmempty* ∧ v_2 = *Vrecabs global-css name fmempty* ∧ *name* |∈| *fmdom global-css*) ∨ $\vdash_v$ v_1 ≈ v_2) (*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*)
    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 (*drule map-of-SomeD*)
apply *auto*
apply (*rule vrecabs-global-css-refl*)
unfolding *global-css-def*
by (*auto simp*: *fset-of-list-elem intro*: *rev-image-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* |∈| *C*
  hence *c* |∉| *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* $|\notin|$ *fmdom* (*fmap-of-list as-vrules*)
by *simp*
hence *fmlookup* (*fmap-of-list as-vrules*) *c* = *None*
by (*metis fmdom-notD*)
thus *rel-option vrelated* (*fmlookup* (*fmap-of-list as-vrules*) *c*) (*fmlookup* (*fmap-of-list as-vrules*) *c*)
by *simp*
qed

lemma *veval′-correct″*:
fixes *t v*
assumes *fmap-of-list as-vrules* $\vdash_v$ $t \downarrow v$
assumes *wellformed t*
assumes $\neg$ *shadows-consts t*
assumes *welldefined t*
assumes *closed t*
assumes *vno-abs v*
shows *as-vrules*, *fmempty* $\vdash_v$ $t \downarrow v$
proof −
obtain v_1 **where** *as-vrules*, *fmempty* $\vdash_v$ $t \downarrow v_1$ $\vdash_v$ $v_1 \approx v$
using ‹*fmap-of-list as-vrules* $\vdash_v$ $t \downarrow v$›
proof (*rule srules-as-vrules.veval′-correct′*, *unfold as-vrules-fst*)
show *wellformed t* $\neg$ *shadows-consts t closed t consts t* $|\subseteq|$ *all-consts*
by *fact+*
next
show *wellformed-venv* (*fmap-of-list as-vrules*)
apply *rule*
using *srules-as-vrules.all-rules*
apply (*auto simp*: *list-all-iff*)
done
next
show *not-shadows-vconsts-env* (*fmap-of-list as-vrules*)
apply *rule*
using *srules-as-vrules.not-shadows*
apply (*auto simp*: *list-all-iff*)
done
next
have *fmrel-on-fset* (*fst* |‘| *fset-of-list as-vrules* $|\cup|$ *C*) *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

show *fmrel-on-fset* (*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 sem-env* ⇒ *v sem-env* **where**
as-sem-env env = ⦇ *sem-env.v* = *build-rec-env* (*mk-letrec-body all-consts rs*) *env nsEmpty*, *sem-env.c* = *nsEmpty* ⦈

lemma *compile-sem-env*:
evaluate-dec ck mn env state (*compile-group all-consts rs*) (*state*, *Rval* (*as-sem-env 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 state env [(*compile-group all-consts rs*)] = (*state*, *Rval* (*as-sem-env 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 (*compile rs*) (*state*, *combine-dec-result* (*as-sem-env env*) (*Rval* ⦇ *sem-env.v* = *nsEmpty*, *sem-env.c* = *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*

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 (*rule cupcake-build-rec-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.rep-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.rec-closure*)
     apply *auto*[]
    apply (*simp add*:
          fmmap-of-list[*symmetric*, *unfolded apsnd-def map-prod-def id-def*]
fmap.rel-map
        global-css-def Let-def map-prod-def comp-def case-prod-twice)
    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 cs*
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*)
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 1*

```
          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
    done
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 t ↓ 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
    apply (subst (2) global-css-def)
    apply (rule fmpred-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 fmpred (λ-. vwelldefined′) (fmap-of-list as-vrules)
  apply (rule fmpred-of-list)
  unfolding as-vrules-def
  apply simp
  apply (erule imageE)
  apply (auto split: 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 image-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 fmpred-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)
  apply (rule sem-env-refl)
  done
next
 show consts t |⊆| fmdom (fmap-of-list as-vrules) |∪| C
  apply (subst fmdom-fmap-of-list)
  apply (subst as-vrules-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* v_1 *ml-v related-v* v_2 *ml-v cake-no-abs ml-v*
  shows $v_1 = v_2$
using *assms* **proof** (*induction arbitrary*: v_2)
  case (*conv* vs_1 *ml-vs name*)
  then obtain vs_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 -› ‹*list-all2* - vs_2 -› ‹*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* ¬ *shadows-consts t wellformed t*
  assumes *cake-no-abs ml-v*
  shows *fmap-of-list as-vrules* $\vdash_v$ *t* ↓ *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 ≡ *fresh-frun* (*Term-to-Nterm.term-to-nterm* [] *t*) *all-consts*

abbreviation *sterm-to-cake* **where**
sterm-to-cake ≡ *rules-as-nrules.crules-as-irules'.irules'-as-prules.prules-as-srules.mk-con all-consts*

abbreviation *term-to-cake t* ≡ *sterm-to-cake* (*pterm-to-sterm* (*nterm-to-pterm* (*term-to-nterm t*)))
abbreviation *cake-to-term t* ≡ (*convert-term* (*value-to-sterm* (*cake-to-value t*)) :: *term*)

abbreviation *cake-sem-env* ≡ *rules-as-nrules.crules-as-irules'.irules'-as-prules.prules-as-srules.sem-env*

definition *compiled* ≡ *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* (*sterm-to-cake t*) (*Rval ml-v*)
  assumes *welldefined t closed t* ¬ *shadows-consts t wellformed t*
  assumes *cake-no-abs ml-v*
  shows *fmap-of-list compiled* $\vdash_v$ *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 ∃ *k*. *Evaluate-Single.evaluate cake-sem-env* (*s* ⦇ *clock* := *k* ⦈) (*term-to-cake t*) = (*s′*, *Rval 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* ⊢ *t* ⟶∗ *cake-to-term ml-v*
proof −
  let *?heads* = *fst* |‘| *fset-of-list rules-as-nrules.crules-as-irules′.irules′-as-prules.prules-as-srules.as-vrules*
  have *?heads* = *heads-of rs*
    using *fmdom-compiled* **unfolding** *compiled-def* **by** *simp*

  have *wellformed* (*nterm-to-pterm* (*term-to-nterm t*))
    by *auto*
  hence *wellformed* (*pterm-to-sterm* (*nterm-to-pterm* (*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* (*term-to-cake t*)
    by (*rule rules-as-nrules.nrules-as-crules.crules-as-irules′.irules′-as-prules.prules-as-srules.srules-as-cake.mk-fact*

obtain *k* **where** *Evaluate-Single.evaluate cake-sem-env* (*s* ⦇ *clock* := *k* ⦈) (*term-to-cake* *t*) = (*s'*, *Rval ml-v*)
using *assms* **by** *blast*
then have *Big-Step-Unclocked-Single.evaluate cake-sem-env* (*s* ⦇ *clock* := (*clock* *s'*) ⦈) (*term-to-cake t*) (*s'*, *Rval ml-v*)
using *unclocked-single-fun-eq* **by** *fastforce*
have *cupcake-evaluate-single cake-sem-env* (*sterm-to-cake* (*pterm-to-sterm* (*nterm-to-pterm* (*term-to-nterm t*)))) (*Rval ml-v*)
apply (*rule cupcake-single-complete*)
apply *fact+*
done
hence *is-cupcake-value ml-v*
apply (*rule rules-as-nrules.crules-as-irules'.irules'-as-prules.prules-as-srules.cupcake-sem-env-preserve*)
by (*auto intro*: *pterm-to-sterm-wellformed*)
hence *vno-abs* (*cake-to-value ml-v*)
using ‹*cake-no-abs* -›
by (*metis rules-as-nrules.nrules-as-crules.crules-as-irules'.irules'-as-prules.prules-as-srules.srules-as-cake.ca*

hence *no-abs* (*value-to-sterm* (*cake-to-value ml-v*))
by (*metis vno-abs-value-to-sterm*)

hence *no-abs* (*sterm-to-pterm* (*value-to-sterm* (*cake-to-value ml-v*)))
by (*metis sterm-to-pterm convert-term-no-abs*)

have *welldefined* (*term-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* (*pterm-to-sterm* (*nterm-to-pterm* (*term-to-nterm t*)))
apply (*subst pterm-to-sterm-consts*)
apply *fact*
apply (*subst consts-nterm-to-pterm*)
apply *fact+*
done

have ¬ *shadows-consts t*
using *assms* **unfolding** *shadows-consts-def fdisjnt-alt-def*
by *auto*
hence ¬ *shadows-consts* (*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 ¬ *shadows-consts* (*pterm-to-sterm* (*nterm-to-pterm* (*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 = *fempty*, *simplified*, *THEN fdisjnt-swap*])
apply (*fold wellformed-term-def*)
apply *fact*
using ‹*closed t*› **unfolding** *closed-except-def* **by** (*auto simp*: *fdisjnt-alt-def*)

have *closed* (*term-to-nterm t*)
using *assms* **unfolding** *closed-except-def*
using *term-to-nterm-vars* **unfolding** *wellformed-term-def* **by** *blast*
hence *closed* (*nterm-to-pterm* (*term-to-nterm t*))
using *closed-nterm-to-pterm* **unfolding** *closed-except-def*
by *auto*
have *closed* (*pterm-to-sterm* (*nterm-to-pterm* (*term-to-nterm t*)))
unfolding *closed-except-def*
apply (*subst pterm-to-sterm-frees*)
apply *fact*
using ‹*closed* (*term-to-nterm t*)› *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-vrules* $\vdash_v$ *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-vrules*, *fmempty* $\vdash_v$ *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 ¬ *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 ‹¬ *shadows-consts* (-::*sterm*)› ‹*?heads* = *heads-of rs*› **by** *auto*
next
show *consts* (*pterm-to-sterm* (*nterm-to-pterm* (*term-to-nterm t*))) |⊆| *rules-as-nrules.crules-as-irules'.irule*
using ‹*welldefined* (*pterm-to-sterm* -)› ‹*?heads* = -› **by** *auto*
qed *fact+*
hence *Rewriting-Sterm.compile* (*Rewriting-Pterm.compile* (*transform-irule-set-iter* (*Rewriting-Pterm-Elim.compile* (*consts-of compile*)))), *fmempty* $\vdash_s$ *pterm-to-sterm* (*nterm-to-pterm* (*term-to-nterm t*)) ↓ *value-to-sterm* (*cake-to-value ml-v*)
by (*rule rules-as-nrules.crules-as-irules'.irules'-as-prules.prules-as-srules.veval-correct*) *fact+*
hence *Rewriting-Sterm.compile* (*Rewriting-Pterm.compile* (*transform-irule-set-iter* (*Rewriting-Pterm-Elim.compile* (*consts-of compile*)))) $\vdash_s$ *pterm-to-sterm* (*nterm-to-pterm* (*term-to-nterm t*)) ⟶∗ *value-to-sterm* (*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*))) $\vdash_p$ *sterm-to-pterm* (*pterm-to-sterm* (*nterm-to-pterm* (*term-to-nterm* *t*))) $\longrightarrow*$ *sterm-to-pterm* (*value-to-sterm* (*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*))) $\vdash_p$ *nterm-to-pterm* (*term-to-nterm t*) $\longrightarrow*$ *sterm-to-pterm* (*value-to-sterm* (*cake-to-value ml-v*))
by (*subst* (*asm*) *pterm-to-sterm-sterm-to-pterm*) *fact*
hence *transform-irule-set-iter* (*Rewriting-Pterm-Elim.compile* (*consts-of compile*)) $\vdash_i$ *nterm-to-pterm* (*term-to-nterm t*) $\longrightarrow*$ *sterm-to-pterm* (*value-to-sterm* (*cake-to-value ml-v*))
by (*rule rules-as-nrules.crules-as-irules'.irules'-as-irules.compile-correct-rt*)
(*rule rules-as-nrules.crules-as-irules.transform-finished*)
have *Rewriting-Pterm-Elim.compile* (*consts-of compile*) $\vdash_i$ *nterm-to-pterm* (*term-to-nterm* *t*) $\longrightarrow*$ *sterm-to-pterm* (*value-to-sterm* (*cake-to-value ml-v*))
apply (*rule rules-as-nrules.crules-as-irules.transform-correct-rt-n-no-abs*)
using ‹*transform-irule-set-iter* - $\vdash_i$ - $\longrightarrow*$ -› **unfolding** *transform-irule-set-iter-def*
apply *simp*
apply *fact+*
done

then obtain t' **where** *compile* $\vdash_n$ *term-to-nterm t* $\longrightarrow*$ t' $t' \approx_i$ *sterm-to-pterm* (*value-to-sterm* (*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* (*sterm-to-pterm* -)›
by (*metis irelated-no-abs*)

have *rs* $\vdash$ *nterm-to-term'* (*term-to-nterm t*) $\longrightarrow*$ *nterm-to-term'* t'
by (*rule compile-correct-rt*) *fact+*
hence *rs* $\vdash$ *t* $\longrightarrow*$ *nterm-to-term'* t'
apply (*subst* (*asm*) *fresh-frun-def*)
apply (*subst* (*asm*) *term-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' = *sterm-to-pterm* (*value-to-sterm* (*cake-to-value ml-v*))
using ‹$t' \approx_i$ -›
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*) *sterm-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 (*subst 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* |∪| *constructors.C C-info*)
(*pterm-to-sterm* (*nterm-to-pterm* (*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*: *all-consts-def*)

primrec *compress-pterm* :: *pterm* ⇒ *pterm* **where**
compress-pterm (*Pabs cs*) = *Pabs* (*fcompress* (*map-prod id compress-pterm* |‘| *cs*))
|
compress-pterm (*Pconst name*) = *Pconst name* |

compress-pterm (*Pvar name*) = *Pvar name* |
compress-pterm (*t* $\$_p$ *u*) = *compress-pterm t* $\$_p$ *compress-pterm u*

lemma *compress-pterm-eq*[*simp*]: *compress-pterm t* = *t*
by (*induction t*) (*auto simp*: *subst-pabs-id fset-map-snd-id map-prod-def*)

definition *compress-crule-set* :: *crule-set* ⇒ *crule-set* **where**
compress-crule-set = *fcompress* ∘ *fimage* (*map-prod id fcompress*)

definition *compress-irule-set* :: *irule-set* ⇒ *irule-set* **where**
compress-irule-set = *fcompress* ∘ *fimage* (*map-prod id* (*fcompress* ∘ *fimage* (*map-prod id compress-pterm*)))

definition *compress-prule-set* :: *prule fset* ⇒ *prule fset* **where**
compress-prule-set = *fcompress* ∘ *fimage* (*map-prod id compress-pterm*)

lemma *compress-crule-set-eq*[*simp*]: *compress-crule-set rs* = *rs*
unfolding *compress-crule-set-def* **by** *force*

lemma *compress-irule-set-eq*[*simp*]: *compress-irule-set rs* = *rs*
unfolding *compress-irule-set-def map-prod-def* **by** *simp*

lemma *compress-prule-set*[*simp*]: *compress-prule-set rs* = *rs*
unfolding *compress-prule-set-def* **by** *force*

definition *transform-irule-set-iter* :: *irule-set* ⇒ *irule-set* **where**
transform-irule-set-iter rs = ((*transform-irule-set* ∘ *compress-irule-set*) ^^ *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 =
  ⦇ *sem-env.v* =
      build-rec-env (*cakeml.mk-letrec-body C-info* (*fset-of-list* (*map fst rs*) |∪| *constructors.C C-info*) *rs*) *env nsEmpty*,
    sem-env.c =
      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-cenv 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*)) (*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-nrules.crules-as-irules′.irules′-as-prules.prules-as-srules.sem-env-def*
unfolding *rules-as-nrules.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