A Verified Code Generator from Isabelle/HOL to CakeML

Lars Hupel

April 18, 2020
Contents

1 Terms 4
  1.1 Additional material over the Higher-Order-Terms AFP entry 4
  1.2 Reflecting HOL datatype definitions 11
  1.3 Constructor information 12
  1.4 Special constants 13
  1.5 Term algebra extended with wellformedness 14
  1.6 Terms with sequential pattern matching 17
  1.7 Terms with explicit pattern matching 25
  1.8 Irreducible terms (values) 35
  1.9 Viewing sterm as values 35
  1.10 A dedicated value type 38

2 A smaller version of CakeML: CupCakeML 55
  2.1 CupCake environments 55
  2.2 CupCake semantics 57

3 Term rewriting 77
  3.1 Higher-order term rewriting using de-Bruijn indices 78
    3.1.1 Matching and rewriting 78
    3.1.2 Wellformedness 78
  3.2 Higher-order term rewriting using explicit bound variable names 80
    3.2.1 Definitions 80
    3.2.2 Matching and rewriting 80
    3.2.3 Translation from Term-Class.term to nterm 81
    3.2.4 Correctness of translation 87
    3.2.5 Completeness of translation 88
    3.2.6 Splitting into constants 91
    3.2.7 Computability 96
  3.3 Higher-order term rewriting with explicit pattern matching 96
    3.3.1 Intermediate rule sets 96
    3.3.2 Pure pattern matching rule sets 133
  3.4 Sequential pattern matching 137
  3.5 Big-step semantics 151
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.1</td>
<td>Big-step semantics evaluating to irreducible \textit{sterms}</td>
<td>152</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Big-step semantics evaluating to \textit{value}</td>
<td>162</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Big-step semantics with conflation of constants and variables</td>
<td>179</td>
</tr>
<tr>
<td>4</td>
<td>Preprocessing of code equations</td>
<td>218</td>
</tr>
<tr>
<td>4.1</td>
<td>A type class for correspondence between HOL expressions and terms</td>
<td>218</td>
</tr>
<tr>
<td>4.2</td>
<td>Deep embedding of Pure terms into term-rewriting logic</td>
<td>221</td>
</tr>
<tr>
<td>4.3</td>
<td>Default instances</td>
<td>223</td>
</tr>
<tr>
<td>5</td>
<td>Final stage: Translation to CakeML</td>
<td>225</td>
</tr>
<tr>
<td>5.1</td>
<td>Basic CakeML setup</td>
<td>225</td>
</tr>
<tr>
<td>5.2</td>
<td>Constructors according to CakeML</td>
<td>227</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Running the generated type declarations through the semantics</td>
<td>228</td>
</tr>
<tr>
<td>5.3</td>
<td>CakeML backend</td>
<td>230</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Compilation</td>
<td>230</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Computability</td>
<td>234</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Correctness of semantic functions</td>
<td>234</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Correctness of compilation</td>
<td>244</td>
</tr>
<tr>
<td>5.4</td>
<td>Converting bytes to integers</td>
<td>270</td>
</tr>
<tr>
<td>6</td>
<td>Composition of phases and full compilation pipeline</td>
<td>272</td>
</tr>
<tr>
<td>6.1</td>
<td>Composition of correctness results</td>
<td>272</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Reflexive-transitive closure of \textit{irules.compile-correct}</td>
<td>272</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Reflexive-transitive closure of \textit{prules.compile-correct}</td>
<td>273</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Reflexive-transitive closure of \textit{rules.compile-correct}</td>
<td>274</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Reflexive-transitive closure of \textit{irules.transform-correct}</td>
<td>274</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Iterated application of \textit{transform-irule-set}</td>
<td>276</td>
</tr>
<tr>
<td>6.1.6</td>
<td>Big-step semantics</td>
<td>278</td>
</tr>
<tr>
<td>6.1.7</td>
<td>ML-style semantics</td>
<td>281</td>
</tr>
<tr>
<td>6.1.8</td>
<td>CakeML</td>
<td>284</td>
</tr>
<tr>
<td>6.1.9</td>
<td>Composition</td>
<td>290</td>
</tr>
<tr>
<td>6.2</td>
<td>Executable compilation chain</td>
<td>295</td>
</tr>
</tbody>
</table>
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 :: - $\Rightarrow$ bool where
  basic-rule (lhs, rhs) $\iff$
    linear lhs $\land$
    is-const (fst (strip-comb lhs)) $\land$
    $\lnot$ is-const lhs $\land$
    frees rhs $\subseteq$ frees lhs

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

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

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

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

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

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

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

declare \(\text{mk-pat.simps}[simp del]\)

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

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

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

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

hence \(\text{fset-of-list (map (patvars o mk-pat) args)} = \text{fset-of-list (map frees args)}\)
by metis
thus ?case
by (simp add: freess-def)
This definition might seem a little counter-intuitive. Assume we have two defining equations of a function, e.g. \( \text{map} \): \( \text{map} \ f \ [] = [] \) \( \text{map} \ f \ (x \# \ xs) = f \ x \# \text{map} \ f \ xs \) The pattern ”matrix” is compiled right-to-left. Equal patterns are grouped together. This definition is needed to avoid the following situation: \( \text{map} \ f \ [] = [] \) \( \text{map} \ g \ (x \# \ xs) = g \ x \# \text{map} \ g \ xs \) 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 :: \( \text{term} \Rightarrow \text{term} \Rightarrow \text{bool} \) where

\[
\text{pattern-compatible} \ (t_1 \$ t_2) \ (u_1 \$ u_2) \longleftrightarrow \text{pattern-compatible} \ t_1 \ u_1 \land (t_1 = u_1 \\
\longrightarrow \text{pattern-compatible} \ t_2 \ u_2) \]

\[
\text{pattern-compatible} \ t \ u \longleftrightarrow t = u \lor \text{non-overlapping} \ t \ u
\]

lemmas pattern-compatible-simps[simp] = 

\[
\text{pattern-compatible} \ . \ \text{simps} \ [\text{folded app-term-def}]
\]

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

lemma pattern-compatible-refl[intro?]: \( \text{pattern-compatible} \ t \ t \)

by (induct t) auto

corollary pattern-compatible-reflP[intro!]: \( \text{reflp} \ \text{pattern-compatible} \)

by (auto intro: pattern-compatible-refl reflpI)

lemma pattern-compatible-cases[consumes 1]:

\[
\text{assumes} \ \text{pattern-compatible} \ t \ u \\
\text{obtains} \ (eq) \ t = u \\
\lor \ (\text{non-overlapping}) \ \text{non-overlapping} \ t \ u
\]

using \( \text{assms} \ \text{proof} \) (induction arbitrary: thesis rule: \( \text{pattern-compatible-induct} \))

\[
\text{case} \ (\text{app-app} \ t_1 \ t_2 \ u_1 \ u_2) \\
\text{show} \ ?\text{case} \\
\text{proof} \ (\text{cases} \ t_1 = u_1 \land t_2 = u_2) \\
\text{case} \ True \\
\text{with} \ \text{app-app} \ \text{show} \ \text{thesis} \\
\text{by} \ \text{simp} \\
\text{next} \\
\text{case} \ False \\
\text{from} \ \text{app-app} \ \text{have} \ \text{pattern-compatible} \ t_1 \ t_1 = u_1 \Longrightarrow \text{pattern-compatible} \\
\text{t}_2 \ u_2 \\
\text{by} \ \text{auto} \\
\text{with} \ False \ \text{have} \ \text{non-overlapping} \ (t_1 \$ t_2) \ (u_1 \$ u_2) \\
\text{using} \ \text{app-app} \ \text{by} \ (\text{metis} \ \text{non-overlapping-app11} \ \text{non-overlapping-app12}) \\
\text{thus} \ \text{thesis}
\]
by (rule app-app.prems(2))

qed

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]) (ys @ [b]) ⇒ P a b

by (auto elim: rev-accum-relE)

end

abbreviation patterns-compatible :: term list ⇒ term list ⇒ bool where

patterns-compatible ≡ rev-accum-rel pattern-compatible

abbreviation patterns-compatibles :: (term list × ‘a) fset ⇒ bool where

patterns-compatibles ≡ fpairwise (λ(pats1, -) (pats2, -). patterns-compatible pats1 pats2)

lemma pattern-compatible-combD:

assumes length xs = length ys pattern-compatible (list-comb f xs) (list-comb f ys)

shows patterns-compatible xs ys

using assms by (induction xs ys rule: rev-induct2) (auto intro: rev-accum-rel.intros)

lemma pattern-compatible-combI[intro]:

assumes patterns-compatible xs ys pattern-compatible f g
shows \textit{pattern-compatible} \((\text{list-comb}\ f\ \text{xs})\ (\text{list-comb}\ g\ \text{ys})\)
using \textit{assms}
\textbf{proof} (induction rule: \textit{rev-accum-rel.induct})
\begin{itemize}
  \item \textbf{case} (\textit{snoc}\ \text{xs}\ \text{ys}\ \text{x}\ \text{y})
\end{itemize}
then have \textit{pattern-compatible} \((\text{list-comb}\ f\ \text{xs})\ (\text{list-comb}\ g\ \text{ys})\)
by \textit{auto}
moreover have \textit{pattern-compatible} \text{xy} if \text{list-comb}\ f\ \text{xs} = \text{list-comb}\ g\ \text{ys}
\textbf{proof} (rule \textit{snoc}, rule \textit{list-comb-semi-inj})
\begin{itemize}
  \item show \text{length}\ \text{xs} = \text{length}\ \text{ys}
    using \textit{snoc} by (auto dest: \textit{rev-accum-rel-length})
\end{itemize}
qed \textit{fact}
ultimately show \textit{case}
by \textit{auto}
qed (auto intro: \textit{rev-accum-rel.intros})
experiment begin
— The above example can be made concrete here. In general, the following identity
does not hold:
\textbf{lemma} \textit{pattern-compatible} \text{tu} \iff \text{tu} \lor \textit{non-overlapping} \text{tu}
\textbf{apply} \textit{rule}
\begin{itemize}
  \item \textbf{apply} (erule \textit{pattern-compatible-cases}; simp)
  \item \textbf{apply} (erule \textit{disjE})
  \item \textbf{apply} (metis \textit{pattern-compatible-refl})
\end{itemize}
\textit{oops}
— The counterexample:
\textbf{definition} \text{pats1} = [\text{Free} (\text{Name} 'f''), \text{Const} (\text{Name} 'nil'')]
\textbf{definition} \text{pats2} = [\text{Free} (\text{Name} 'g''), \text{Const} (\text{Name} 'cons'' \text{x}'') \$ \text{Free} (\text{Name} 'x''')]
\textbf{proposition} \textit{non-overlapping} \((\text{list-comb}\ c\ \text{pats1})\ (\text{list-comb}\ c\ \text{pats2})\)
\textbf{unfolding} \text{pats1-def pats2-def}
\textbf{apply} (simp add: \text{app-term-def})
\textbf{apply} (rule \textit{non-overlapping-appI2})
\textbf{apply} (rule \textit{non-overlapping-const-appI})
\textbf{done}
\textbf{proposition} \neg \textit{patterns-compatible} \text{pats1 pats2}
\textbf{unfolding} \text{pats1-def pats2-def}
\textbf{apply} \textit{rule}
\textbf{apply} (erule \textit{rev-accum-rel.cases})
\textbf{apply} \textit{simp}
\textbf{apply} \textit{auto}
apply (erule rev-accum-rel.cases)
apply auto
apply (erule rev-accum-rel.cases)
apply auto
apply (metis overlapping-varII)
done

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 o fst) cs is-fmap (fset-of-list cs)
  assumes match pat t = Some env (pat, rhs) ∈ set cs
  shows find-match cs t = Some (env, pat, rhs)
using assms proof (induction cs arbitrary: pat rhs)
case (Cons c cs)
then obtain [simp]: c = (pat', rhs') by force
have find-match ((pat', rhs') # cs) t = Some (env, pat, rhs)
proof (cases match pat' t)
case None
have pattern-compatibles (fset-of-list cs)
  using Cons by (force simp: fpairwise-alt-def)
have list-all (linear o fst) cs
  using Cons by (auto simp: list-all-iff)
have is-fmap (fset-of-list cs)
  using Cons by (meson fset-of-list-subset is-fmap-subset set-subset-Cons)
show ?thesis
  apply (simp add: None)
  apply (rule Cons)
  apply fact
  using Cons None by force
next
case (Some env')
have linear pat linear pat'
  using Cons apply (metis Ball-set comp-app 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 :: \('a 'a \Rightarrow bool\) where
  arity-compatible \(t_1\ t_2 = \)
  (let
    (head_1, \(\text{pats}_1\)) = strip-comb \(t_1\);
    (head_2, \(\text{pats}_2\)) = strip-comb \(t_2\)
  in head_1 = head_2 \(\rightarrow\) length pats_1 = length pats_2)

abbreviation arity-compatibles :: \('(a 'b) fset \Rightarrow bool\) where
  arity-compatibles \(?= \text{fpairwise} (\lambda (\text{lhs}_1, \_). (\text{lhs}_2, \_). \text{arity-compatible} \text{lhs}_1 \text{lhs}_2)\)

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

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

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

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

end

1.2 Reflecting HOL datatype definitions

thory HOL-Datatype
imports
  Terms-Extras
  HOL-Library.Datatype-Records
  HOL-Library.Finite-Map
  Higher-Order-Terms.Name
begin
datatype \texttt{typ} = \\
\hspace{1em} \texttt{TVar name} | \\
\hspace{1em} \texttt{TApp name typ list}

datatype-compat \texttt{typ}

context begin

qualified definition \texttt{tapp-0} where \\
tapp-0 \texttt{tc} = \texttt{TApp tc []}

qualified definition \texttt{tapp-1} where \\
tapp-1 \texttt{tc t1} = \texttt{TApp tc [t1]}

qualified definition \texttt{tapp-2} where \\
tapp-2 \texttt{tc t1 t2} = \texttt{TApp tc [t1, t2]}

end

quickcheck-generator \texttt{typ} \\
\hspace{1em} constructors:
\hspace{3.5em} \texttt{TVar},
\hspace{3.5em} \texttt{HOL-Datatype.tapp-0},
\hspace{3.5em} \texttt{HOL-Datatype.tapp-1},
\hspace{3.5em} \texttt{HOL-Datatype.tapp-2}

datatype-record \texttt{dt-def} = \\
tparams :: \texttt{name list}
\hspace{1.5em} constructors :: \texttt{(name, typ list) fmap}

ML-file \texttt{hol-datatype.ML}

end

1.3 Constructor information

theory \textit{Constructors} \\
imports \\
\hspace{1em} \textit{Terms-Extras} \\
\hspace{1em} \textit{HOL-Datatype} \\
begin

\hspace{1em} \texttt{type-synonym C-info = (name, dt-def) fmap}

locale constructors = \\
\hspace{1.5em} \texttt{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))

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 |\u| C
end
abbreviation welldefined :: 'a::term ⇒ bool where
welldefined t ≡ consts t ⊆ 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 ⇒ bool
  fixes all-frees :: 'a ⇒ name fset
  assumes wellformed-const[simp]: wellformed (const name)
  assumes wellformed-free[simp]: wellformed (free name)
  assumes wellformed-app[simp]: wellformed (app u1 u2) ←→ wellformed u1 ∧ wellformed u2
  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 u1 u2) = all-frees u1 |∪| all-frees u2
begin

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

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

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

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

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

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

begin

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

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

end

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

instantiation term :: strong-term begin

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

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

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

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

instance proof (standard, goal-cases)
case 8

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

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

end

instantiation nterm :: strong-term begin

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

fun all-frees-nterm :: nterm \Rightarrow 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 (\Lambda n x. t) = finsert x (all-frees-nterm t) |
all-frees-nterm (Nconst -) = \{||\}

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

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

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

end

1.6 Terms with sequential pattern matching

theory Stern
imports Strong-Term
begin

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

datatype-compat sterm

derive linorder sterm

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

type-synonym sclauses = (term × sterm) list

lemma sterm-induct[case-names Sconst Svar Sabs Sapp]:
  assumes ∀x. P (Sconst x)
  assumes ∀x. P (Svar x)
  assumes ∀cs. (∀pat t. (pat, t) ∈ set cs ⇒ P t) ⇒ P (Sabs cs)
  assumes ∀t u. P t ⇒ P u ⇒ P (t $₧ 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 $₧ u

fun unapp-sterm where
unapp-sterm (t $₧ 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 (\(\lambda (pat, rhs). frees-sterm rhs - frees pat) cs)) |
frees-sterm (t $ s u) = frees-sterm t |\cup| 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 (t1 $ s t2) env = subst-sterm t1 env $ s subst-sterm t2 env |
subst-sterm (Sabs cs) env = Sabs (map (\(\lambda (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 (\(\lambda (:, rhs). consts-sterm rhs) cs)) |
consts-sterm (t $ s u) = consts-sterm t |\cup| 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. (pat, t) ∈ set cs → P t) → P t)

**lemma** abs-pred-stermI[intro]:
  assumes ∨ cs. (∀ pat. (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 (subt (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 (subt (2) prems(1)[symmetric, where pat = pat])
  subgoal by fact
  subgoal
  using prems
  unfolding fUnion-alt-def
  by (auto simp add: fmember.rep-eq fset-of-list.rep-eq elim!: fBallE)
  subgoal
  apply (rule arg-cong[where f = subst rhs])
  by (auto intro: fmap-ext)
  done
  done
next
  case (4 t)
  show ?case
  apply rule

19
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 \( \text{name} \mid \in \mid \text{frees} \ \text{rhs} \mid \neg \mid \text{fmdom} \ \text{(fmdrop-fset (frees pat) env)} \mid \neg \mid \text{frees pat} \)

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

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

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

using 1 by (simp add: fnpred-drop-fset)

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

apply (simp add: case-prod-twice comp-def)

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

apply (rule fBexI)

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

apply (subt fimage-iff)

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

apply simp

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

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

for \( m \ A f g \)

including \( \text{fset.lifting \ fmap.lifting} \)

by transfer' force

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

apply (simp add: funion-image-bind-eq)

apply (rule fbind-cong[OF refl])

apply (clarsimp split: prod.splits)

apply (subst 1)

apply (subt (asm) fset-of-list-elem, assumption)

apply simp

by (simp add: funion-image-bind-eq)

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

apply (subt fbind-fun-funion[symmetric])

apply (rule fbind-cong[OF refl])

by auto
also have ... = consts (Sabs cs) \( \cup \) fbind (fset-of-list cs) \((\lambda (pat, rhs). \text{ffUnion (consts } | \cdot \text{)| fnimage (fndrop-fset (frees pat) env) (frees rhs)))\)
apply (rule cong[OF cong, OF refl refl, where \( f1 = \text{funion} \)])
apply (subst funion-image-bind-eq[ symmetric])
unfolding consts-sterm.simps
apply (rule arg-cong[where \( f = \text{ffUnion} \)])
apply (subst fset-of-list-map)
apply (rule fset.map-cong[OF refl])
by auto
also have ... = consts (Sabs cs) \( \cup \) fbind (fnimage env (fbind (fset-of-list cs) \((\lambda (pat, rhs). \text{frees rhs } |\cdot| \text{frees pat}))\))
apply (subst funion-image-bind-eq)
apply (subst fnimage-drop-fset)
apply (rule cong[OF cong, OF refl refl, where \( f1 = \text{funion} \)])
apply (subst *)
apply (rule fbind-cong[OF refl])
by auto
also have ... = consts (Sabs cs) \( \cup \) \text{ffUnion (consts } | \cdot \text{)| fnimage env (frees (Sabs cs)))
by (simp only: frees-sterm.simps fset-of-list-map fnimage-Union funion-image-bind-eq)
finally show \?case.
qed
qed (auto simp: abs-pred-sterm-def)
end

lemma no-abs-abs[simp]: \( \neg \text{no-abs (Sabs cs)} \)
by (subst no-abs.simps) (auto simp: term-cases-def)

lemma closed-except-simps:
closed-except \((\text{Svar } x)\) \( S \rightsquigarrow x \in| S \)
closed-except \((t_1 \& t_2)\) \( S \rightsquigarrow \text{closed-except } t_1 \ S \land \text{closed-except } t_2 \ S \)
closed-except \((\text{Sabs cs})\) \( S \rightsquigarrow \text{list-all } (\lambda (pat, t). \text{closed-except t } (S \cup \text{frees pat}))\)
closed-except \((\text{Sconst name})\) \( S \rightsquigarrow \text{True} \)
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 minusI 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) ∈ set cs
  shows closed-except rhs (frees pat)

using assms unfolding closed-except-def

apply auto

by (metis bot.extremum-uniqueI fempty-iff ffUnion-subset-elem fimageI fminusI fset-of-list-elem old.prod.case)

instantiation sterm :: strong-term begin

fun wellformed-sterm :: sterm ⇒ bool where
  wellformed-sterm (t₁ $₄ t₂) ←→ wellformed-sterm t₁ ∧ wellformed-sterm t₂ |
  wellformed-sterm (Sabs cs) ←→ list-all (λ(pat, t). linear pat ∧ wellformed-sterm t) cs ∧ distinct (map fst cs) ∧ cs ≠ [] |
  wellformed-sterm - ←→ True

primrec all-frees-sterm :: sterm ⇒ name fset where
  all-frees-sterm (Svar x) = {x} |
  all-frees-sterm (t₁ $₄ t₂) = all-frees-sterm t₁ |∪| all-frees-sterm t₂ |
  all-frees-sterm (Sabs cs) = ffUnion (fset-of-list (map (λ(P, T) (map (map-prod frees all-frees-sterm) cs))) |)
  all-frees-sterm (Sconst -) = {||}

instance proof (standard, goal-cases)

  case (7 t)
  show ?case
    apply (intro abs-pred-stermI allI 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)

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

end

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

context pre-constants begin

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

lemma shadows-consts-sterm-simps[simp]:
  shadows-consts (t₁ $ₜ₂) =⇒ shadows-consts t₁ ∨ shadows-consts t₂
  shadows-consts (Svar name) =⇒ name |∈| all-consts
  shadows-consts (Sabs cs) =⇒ list-ex (λ(pat, t). ¬ fdisjnt all-consts (frees pat) ∨ shadows-consts t) cs
  shadows-consts (Sconst name) =⇒ False
proof (goal-cases)
  case 3
  unfolding shadows-consts-def list-ex-iff
  apply rule
  subgoal
    by (force simp: ffUnion-alt-def fset-of-list-elem fdisjnt-alt-def elim!: ballE)
  subgoal
    by (auto simp: fset-of-list-elem fdisjnt-alt-def)
    by (auto simp: fset-eq-empty-ifff Union-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::$t) not-shadows-consts-env Γ'

24
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-pterms :: sterm ⇒ pterm where
  sterm-to-pterms (Sconst name) = Pconst name |
  sterm-to-pterms (Svar name) = Pvar name |
  sterm-to-pterms (t $p u) = sterm-to-pterms t $p pterm u |
  sterm-to-pterms (Sabs cs) = Pabs (fset-of-list (map (map-prod id sterm-to-pterms) cs))

quickcheck-generator pterm
— will print some fishy “constructor” names, but at least it works
constructors: sterm-to-pterms
lemma sterm-to-pterm-total:
  obtains $t'$ where $t = \text{sterm-to-pterm } t'$
proof (induction $t$ arbitrary: thesis)
  case ($P\text{const } x$)
  then show $?case$
    by (metis sterm-to-pterm.simps)
next
  case ($P\text{var } x$)
  then show $?case$
    by (metis sterm-to-pterm.simps)
next
  case ($P\text{abs } cs$)
from $P\text{abs} \text{IH}$ obtain $cs'$ where $cs = \text{fset-of-list } (\text{map-prod } id \text{ sterm-to-pterm})
  cs'$
  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 $\exists cs'. \ cs = \text{fset-of-list } (\text{map-prod } id \text{ sterm-to-pterm})
      cs'$
      apply (rule insert)
      using insert.prems unfolding finset.rep-eq
      by blast
    then obtain $cs'$ where $cs = \text{fset-of-list } (\text{map-prod } id \text{ sterm-to-pterm})
      cs'$
      by blast
    obtain $rhs'$ where $rhs = \text{sterm-to-pterm } rhs'$
    apply (rule insert.prems[of (pat, rhs) rhs])
    unfolding $\langle c = \cdot \rangle$ by simp
    show $?case$
      apply (rule exI [where $x = (pat \ rhs') \# cs'$])
      unfolding $\langle c = \cdot \rangle \langle cs = \cdot \rangle \langle rhs = \cdot \rangle$
      by (simp add: id-def)
    qed
  hence $P\text{abs } cs = \text{sterm-to-pterm } (S\text{abs } cs')$
  by simp
  then show $?case$
    using $P\text{abs}$ by metis
next
  case ($P\text{app } t1 \ t2$)
  then obtain $t1' \ t2'$ where $t1 = \text{sterm-to-pterm } t1' \ t2 = \text{sterm-to-pterm } t2'$
  by metis
then have \( t1 \$p t2 = \text{stem-to-term} (t1' \$s t2') \)
  by simp
with \( P\text{app} \) show \(?case\)
  by metis
qed

lemma \( p\text{term-induct}[\text{case-names } P\text{const } P\text{var } P\text{abs } P\text{app}]:\)
assumes \( \forall x. \; P (P\text{const } x) \)
assumes \( \forall x. \; P (P\text{var } x) \)
assumes \( \forall cs. \; (\forall \text{pat } t. \; (\text{pat}, \; t) \triangleright\triangleright| cs \implies P \; t) \implies P (P\text{abs } cs) \)
assumes \( \forall t \; u. \; P \; t \implies P \; u \implies P (t \; \$p \; u) \)
shows \( P \; t \)
proof (rule \( p\text{term.induct}, \text{goal-cases} \))
  case (3 \( cs \))
  show \(?case\)
    apply (rule assms)
    using 3
    apply (subst (asm) \( \text{fnmember.rep-eq[symmetric]} \))
    by auto
qed \( \text{fact}+ \)

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

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

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

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

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

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

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

function (sequential) \( \text{subst-pterm} \) where
\( \text{subst-pterm} (P\text{var } s) \; \text{env} = (\text{case } \text{fnlookup } \text{env } s \; \text{of } \text{Some } t \Rightarrow t \; | \; \text{None } \Rightarrow P\text{var } s) \) |
\( \text{subst-pterm} (t1 \; \$p \; t2) \; \text{env} = \text{subst-pterm } t1 \; \text{env} \; \$p \; \text{subst-pterm } t2 \; \text{env} \) |
\( \text{subst-pterm} (P\text{abs } cs) \; \text{env} = P\text{abs } ((\lambda(\text{pat}, \; \text{rhs}). \; (\text{pat}, \; \text{subst-pterm } \text{rhs } (\text{fndrop-fset} \; \text{cs}))) \; \text{env}) \) |
(frees pat) env)) |\{ x | cs \} |
subst-pterm t - = t
by pat-completeness auto

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

primrec consts-pterm :: pterm \Rightarrow name fset where
consts-pterm (PConst x) = \{|x|\} |
consts-pterm (t1 \&| t2) = consts-pterm t1 |\cup| consts-pterm t2 |
consts-pterm (Pabs cs) = \text{ffUnion} (\text{snd} |\cdot| \text{map-prod id} \text{consts-pterm} |\cdot| \text{cs} |)
consts-pterm (Pvar -) = \{|\|\}

primrec frees-pterm :: pterm \Rightarrow name fset where
frees-pterm (PVar x) = \{|x|\} |
frees-pterm (t1 \&| t2) = frees-pterm t1 |\cup| frees-pterm t2 |
frees-pterm (Pabs cs) = \text{ffUnion} ((\lambda (pv, tv). tv - frees pv) |\cdot| \text{map-prod id} \text{frees-pterm} |\cdot| \text{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 \forall pat rhs. (pat, rhs) |\in| cs \Rightarrow 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 (subst (asm) fnmember.rep-eq[ symmetric ])
apply assumption
done
**corollary** frees-pabs-alt-def:
\[
frees (\text{Pabs cs}) = \text{ffUnion} ((\lambda (\text{pat}, \text{rhs}). \frees \text{rhs} - \frees \text{pat} \mid \text{\{\mid cs\}}) \mid \text{\text{\mid cs}})
\]

**apply** simp

**apply** (rule arg-cong [where \( f = \text{ffUnion} \])

**apply** (rule \text{fset.map-cong} [OF \text{refl}])

**by** auto

**lemma** sterm-to-pterm-frees[simp]: frees (sterm-to-pterm \( t \)) = frees \( t \)

**proof** (induction \( t \))

**case** (\text{Sabs cs})

**show** \(?case\)

**apply** simp

**apply** (rule arg-cong [where \( f = \text{ffUnion} \])

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

**apply** clarsimp

**apply** (subst Sabs)

**by** (auto simp: \text{fset-of-list-elem} \text{snds.simps})

**qed** auto

**lemma** sterm-to-pterm-consts[simp]: const (sterm-to-pterm \( t \)) = const \( t \)

**proof** (induction \( t \))

**case** (\text{Sabs cs})

**show** \(?case\)

**apply** simp

**apply** (rule arg-cong [where \( f = \text{ffUnion} \])

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

**apply** clarsimp

**apply** (subst Sabs)

**by** (auto simp: \text{fset-of-list-elem} \text{snds.simps})

**qed** auto

**lemma** subst-sterm-to-pterm:
subst (sterm-to-pterm \( t \)) (\text{fmmap sterm-to-pterm} \text{env}) = sterm-to-pterm (subst \( t \) \text{env})

**proof** (induction \( t \) arbitrary: \text{env} rule: sterm-induct)

**case** (\text{Sabs cs})

**show** \(?case\)

**apply** simp

**apply** (rule \text{fset.map-cong9})

**apply** (auto split: prod.splits)

**apply** (rule Sabs)

**by** (auto simp: \text{fset-of-list-rep-eq})

**qed** (auto split: option.splits)

**instantiation** pterm :: term begin

**definition** abs-pred-ptermm :: (pterm \( \Rightarrow \) bool) \( \Rightarrow \) pterm \( \Rightarrow \) bool where
\[
\text{[code del]}: \text{abs-pred} \ P \ t \longleftrightarrow (\forall \text{cs} \ t = \text{Pabs cs} \longrightarrow (\forall \text{pat} t. (\text{pat}, t) \mid \text{\text{\mid cs}}) \longrightarrow P \ t) \longrightarrow P \ t
\]
context begin

private lemma abs-pred-trivI0: P t \implies \text{abs-pred } P (t::pterm)

unfolding abs-pred-pterms-def by auto

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

  case (2 t)
  show ?case
    unfolding abs-pred-pterms-def
    apply clarify
    apply (rule subst-pabs-id)
    by blast
  next
  case (3 x t)
  show ?case
    unfolding abs-pred-pterms-def
    apply clarsimp
    apply (rule fset.map-cong0)
    apply (rename-tac c; hypsubst-thin)
    apply simp
    subgoal for cs env pat rhs
      apply (cases x |\in| frees pat)
      subgoal
        apply (rule arg-cong[where f = subst rhs])
        by (auto intro: fmap-ext)
      subgoal premises prems[rule-format]
        apply (subst (2) prems(1)[symmetric, where pat = pat])
      subgoal
        by (subst fnmember.rep-eq) fact
      subgoal
        using prems unfolding \fUnion-alt-def
        by (auto simp: fnmember.rep-eq fset-of-list.rep-eq elim!: fBallE)
      subgoal
        apply (rule arg-cong[where f = subst rhs])
        by (auto intro: fmap-ext)
      done
  done

30
done
next
case (4 t)
show ?case
unfolding abs-pred-pterm-def
apply clarsimp
apply (rule fset.map-cong0)
apply clarsimp
subgoal premises prems[rule-format] for cs env1 env2 a b
apply (rule prems(2)[unfolded fmemb.rep-eq, OF prems(5)])
using prems unfolding fdisj-all-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 = stern-to-pterm t'
  by (rule stern-to-pterm-total)
obtain env' where env = fmmap stern-to-pterm env'
  by (metis fmmap-total stern-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-stern-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 = stern-to-pterm t'
  by (rule stern-to-pterm-total)
obtain env' where env = fmmap stern-to-pterm env'
  by (metis fmmap-total stern-to-pterm-total)
show consts (subst t env) = consts t |∪| ffUnion (consts |'| fmimage env (frees t))
unfolding \( t = \cdot \) (env = \cdot)
apply simp
apply (subst comp-def)
apply simp
apply (subst subst-stem-to-term)
apply simp
apply (rule subst-consts')
done
qed
qed (rule abs-pred-trivI0)

end

end

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

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

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

lemma closed-except-simps:
closed-except (Pvar x) S \leftrightarrow x \in| S
closed-except (t_1 \\_p t_2) S \leftrightarrow closed-except t_1 S \land closed-except t_2 S
closed-except (Pabs cs) S \leftrightarrow fBall cs (\lambda(pat, t). closed-except t (S |\cup| frees pat))
closed-except \((P\text{const name}) S\) \(\leftrightarrow\) 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-pterms :: pterm \(\Rightarrow\) bool where
  wellformed-pterms \((t, p, t_2)\) \(\leftrightarrow\) wellformed-pterms \(t_1\) \(\land\) wellformed-pterms \(t_2\)
  wellformed-pterms \((Pabs cs)\) \(\leftrightarrow\) fBall \(cs\) \((\lambda\text{pat}, t).\text{linear pat} \land\) wellformed-pterms \(t\) \(\land\) is-fmap \(cs\) \(\land\) pattern-compatibles \(cs\) \(\land\) \(cs\) \(\neq\) {||}
  wellformed-pterms - \(\leftrightarrow\) True

by pat-completeness auto

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

primrec all-frees-pterms :: pterm \(\Rightarrow\) name fset where
  all-frees-pterms \((P\text{var} x)\) = \{|x|\} |
  all-frees-pterms \((t, p, t_2)\) = all-frees-pterms \(t_1\) \(\cup\) all-frees-pterms \(t_2\) |
  all-frees-pterms \((Pabs cs)\) = ffUnion \((\lambda(P, T). P \cup T)\) \map\ map-prod frees all-frees-pterms\[\map| cs\] |
  all-frees-pterms \((P\text{const})\) = {||}

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

end
lemma sterm-to-pterm-all-frees[simp]: all-frees (sterm-to-pterm t) = all-frees t
proof (induction t)
case (Subs cs)
  show ?case
  apply simp
  apply (rule arg-cong[where f = ffUnion])
  apply (rule fmap-cong[OF refl])
  apply clarsimp
  apply (rule subst Subs)
  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 ≠ {||}
  assumes ∪ pat t. (pat, t) ∈| cs ⇒ linear pat
  assumes ∪ pat t. (pat, t) ∈| cs ⇒ 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: fnest-alt-def closed-except-simps)

lemma (in constants) shadows-consts-pterms-simps[simp]:
  shadows-consts (t₁ $ₚ t₂) ↔ shadows-consts t₁ ∨ shadows-consts t₂
shadows-consts (Pvar name) \iff name \in \text{all-consts}
shadows-consts (Pabs cs) \iff fBex cs (\lambda (pat, t). \text{shadows-consts } pat \lor \text{shadows-consts } t)
shadows-consts (Pconst name) \iff \text{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 Stem
begin

1.9 Viewing \textit{sterm} as values

declare list.pred-mono[mono]

context constructors begin

inductive is-value :: \textit{sterm} \Rightarrow bool where
abs: is-value (Sabs cs) |
constr: list-all is-value vs \Rightarrow name \in C \Rightarrow 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 vs = 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 ⇒ 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 (++) fnempty) (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 (++) fnempty) (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-valueintros)
    apply (subst (2) const-sterm-def)
    apply (auto simp: matches-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) list 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-list [(name, v)]) |
vmatch (Patconstr name ps) (Vconstr name' vs) =
  (if name = name' ∧ length ps = length vs then
    map-option (foldl (++) fnempty) (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 (fmapmap pred Γ) ∧ P Γ cs |
pred (Vrecabs css name Γ) ←→
pred-fmap id (fmapmap pred Γ) ∧
pred-fmap (P Γ) css ∧
  name |∈| fmdom css ∧
  R (fmdom css)

declare pred.simps [simp def]

lemma pred-alt-def [simp, code]:
pred (Vconstr name vs) ←→ Q name ∧ list-all pred vs
pred (Vabs cs Γ) ←→ fnpred (λ-. pred) Γ ∧ P Γ cs
pred (Vrecabs css name Γ) ←→ fnpred (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 fnpred (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 \( \text{fnpred} (\lambda \cdot \text{pred}) \text{ env} \)

using \( \text{assms} \) proof (induction \( \text{pat} \) \( \text{v} \) arbitrary; env rule: \text{vmatch-induct})

\begin{align*}
\text{case} & (\text{constr} \text{ name} \text{ ps name' vs}) \\
\text{hence} & \text{map-option} (\text{foldl} (++f) \text{fnempty}) (\text{those} (\text{map2 vmatch ps vs})) = \text{Some env} \\
& \text{by} (\text{auto split: if-splits}) \\
\text{then obtain} & \text{ens where} \text{ env} = \text{foldl} (++f) \text{fnempty envs map2 vmatch ps vs} \\
& = \text{map} \text{ Some envs} \\
& \text{by} (\text{blast dest: those-someD})
\end{align*}

moreover have \( \text{fnpred} (\lambda \cdot \text{pred}) \text{ env} \) if \( \text{ env} \in \text{ set envs} \) for \( \text{ env} \)

proof –

\begin{align*}
& \text{from} \text{ that} \text{ have} \text{ Some env} \in \text{ set} (\text{map2 vmatch ps vs}) \\
& \text{unfolding} (\text{map2 - - - = -}) \text{ by simp} \\
& \text{then obtain} \text{ p v where} \text{ p} \in \text{ set} \text{ ps} \text{ v} \in \text{ set vs vmatch p v} = \text{Some env} \\
& \text{apply} (\text{rule map2-elemE}) \\
& \text{by auto} \\
& \text{hence} \text{ pred v} \\
& \text{using} \text{ constr by (simp add: list-all-iff)} \\
& \text{show} \ ?\text{thesis} \\
& \text{by} (\text{rule constr; safe?}) \text{ fact+} \\
& \text{qed}
\end{align*}

ultimately show \( ?\text{case} \)

by auto

qed auto

end

primrec value-to-sterm :: value \( \Rightarrow \) sterm where

\begin{align*}
\text{value-to-sterm} (\text{Vconstr} \text{ name} \text{ vs}) & = \text{name} \text{ \$\$ map value-to-sterm vs} | \\
\text{value-to-sterm} (\text{Vabs} \text{ cs} \text{ \( \Gamma \)}) & = \text{Sabs} (\text{map} (\lambda (\text{pat}, \text{t}). (\text{pat}, \text{subst t (fmdrop-fset (frees pat) (fnmap value-to-sterm \( \Gamma \)))) \text{ cs}) | \\
\text{value-to-sterm} (\text{Vrecabs} \text{ css name} \text{ \( \Gamma \)}) & = \\
& \text{Sabs} (\text{map} (\lambda (\text{pat}, \text{t}). (\text{pat}, \text{subst t (fmdrop-fset (frees pat) (fnmap value-to-sterm \( \Gamma \)))) \text{ (the (fnlookup css name))}) \text{ (the (fnlookup css name))}) \\
\end{align*}

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

locale pre-value-sterm-pred = value-pred +

\text{fixes} \text{ S} \\
\text{assumes} \text{ value-to-sterm: pred v \( \Rightarrow \) S (value-to-sterm v)}

begin

\text{corollary} \text{ value-to-sterm-env:}

\text{assumes} \text{ fnpred (\lambda \cdot \text{pred}) \( \Gamma \)}

\text{shows} \text{ fnpred (\lambda \cdot \text{S}) (fnmap value-to-sterm \( \Gamma \))}

unfolding \text{ fnpred-map proof}
fix name v
assume fnlookup Γ name = Some v
with assms have pred v by (metis fnpredD)
thus S (value-to-stern v) by (rule value-to-stern)
qed

end

locale value-stern-pred = value-pred + S: simple-syntactic-and S for S +
assumes const: \(\forall name. Q name \implies S (\text{const name})\)
assumes abs: \(\forall \Gamma cs. (\forall n. \text{fnlookup} \, \Gamma \, n = \text{Some} \, v \implies \text{pred} \, v \implies S (\text{value-to-stern} \, v)) \implies \text{fnpred} (\lambda-. \, \text{pred}) \, \Gamma \implies P \, \Gamma \, cs \implies S (\text{Sabs} (\text{map} (\lambda pat, t). (\text{pat}, \text{subst} \, t \, (\text{fnmap} \, \text{value-to-stern} \, (\text{fndrop-fset} \, (\text{frees} \, \text{pat}) \, \Gamma)))) \, cs))\)
begin
sublocale pre-value-stern-pred
proof
fix v
assume pred v
then show S (value-to-stern 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: fnran'I)
next
case (Vrecabs x1 x2 x3)
  show ?case
  apply auto
  apply (rule abs)
  using Vrecabs by (auto simp: fnlookup-dom-iff fnpred-iff intro: fnran'I)
qed
qed
end

41
global-interpretation $v_wellformed$:
  value-sterm-pred
  $\lambda$. wellformed-clauses
  $\lambda$. True
  $\lambda$. True
  wellformed
  defines $v_wellformed = v_wellformed.prd$
proof (standard, goal-cases)
case (2 $\Gamma$ $cs$)
hence $cs \neq []$
  by simp
moreover have wellformed (subst rhs (fmdrop-fset (frees pat) (fmmap value-to-sterm $\Gamma$)))
  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 $\circ$ ($\lambda$(pat, t). (pat, subst t (fmmap value-to-sterm (fmdrop-fset (frees pat) $\Gamma$))))) $cs$)
  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 ($\lambda$. $v_wellformed$)

global-interpretation $v_{closed}$:
  value-sterm-pred
  $\lambda\Gamma$ $cs$. list-all ($\lambda$(pat, t). closed-except t (fdom $\Gamma$ $|$ $\cup$ frees pat)) $cs$
  $\lambda$. True
  $\lambda$. True
  closed
  defines $v_{closed} = v_{closed.prd}$
proof (standard, goal-cases)
case (2 $\Gamma$ $cs$)
  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-term-pred
    λ- cs. list-all (λ(-, t). welldefined t) cs
    λname. name ∈ C
    λdom. dom ⊆ heads
  welldefined

defines vwelldefined = vwelldefined.pred

proof (standard, goal-cases)
  case (2 Γ cs)
  note fset-of-list-map[simp del]

show ?case
  apply simp
  apply (rule ffUnion-least)
  apply (rule fBallI)
  apply (subst (asm) fset-of-list-elem)
  apply simp
  apply (erule imageE)
  apply (simp add: case-prod-twice)

subgoal for - x
  apply (cases x)
  apply simp
  apply (rule substconsts)

subgoal
  using 2 by (fastforce simp: list-all-iff)

subgoal
  apply simp
  apply (rule fmpred-drop-fset)
  unfolding fmpred-map
  apply (rule fmpredI)
  using 2 by auto

done

done

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

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

context constructors begin

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

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

abbreviation vconstructor-value-env ≡ 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-stern-pred
    \lambda\cdot cs. list-all (\lambda(pat, t). fdissjnt all-consts (frees pat) \land \neg shadows-consts t) cs
  \lambda\cdot True
  \lambda\cdot True
  \lambda t. \neg shadows-consts t
defines not-shadows-vconsts = not-shadows-consts.pred
proof (standard, goal-cases)
case (2 \Gamma 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 fnpred-drop-fset)
          apply (rule fnpredI)
          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 ≡ fnpred (\lambda s. not-shadows-vconsts s)

end

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

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

lemma (in constructors) term-to-value-to-sterm:
  assumes is-value t
  shows value-to-sterm (term-to-value t) = t
  using assms proof induction
  case (constr vs name)

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

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

unfolding \( (\text{map2 } \text{vmatch } ps vs = \cdot) \) by auto
moreover hence \( \text{fdom } \text{env} = \text{patvars } p \)
using constr \( (\text{name } = \text{name}') \) \( (\text{length } ps = \text{length } vs) \) \( (p \in \text{set } ps) \) \( (v \in \text{set } vs) \)
by fastforce
ultimately have \( \text{names } \in \text{set } (\text{map } \text{fdom } \text{envs}) \)
unfolding \( (\text{names } = \cdot) \) by auto
thus \( \text{names } \in \text{fset-of-list } (\text{map } \text{fdom } \text{envs}) \)
unfolding fset-of-list-elem .
qed

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

fun \text{vfind-match} :: \text{sclauses } \Rightarrow \text{value } \Rightarrow ((\text{name }, \text{value }) \text{ fmap } \times \text{term } \times \text{stern}) \text{ option where}
\text{vfind-match} [] = \text{None} |
\text{vfind-match} ((\text{pat }, \text{rhs }) \# \text{ cs }) \text{ t } =
(case \text{vmatch } (\text{mk-pat pat }) \text{ t } \text{ of}
  \text{Some } \text{env } \Rightarrow \text{Some } (\text{env }, \text{pat }, \text{rhs })
  | \text{None } \Rightarrow \text{vfind-match } \text{cs } \text{ t })

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

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

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

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

global-interpretation \text{vno-abs}: \text{value-pred } \lambda - - . \text{ False } \lambda -. \text{ True } \lambda -. \text{ 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 (++) 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 =
  defines Q :: value ⇒ value ⇒ bool
assumes Q-impl-struct: Q t1 t2 → venq-structure t1 t2
assumes Q-def[simp]: Q (Vconstr name ts) (Vconstr name' us) ↔ name = name' ∧ list-all2 Q ts us

begin

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

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

custom begin

private lemma vmatch′-rel:
  assumes Q t1 t2
  shows rel-option (fmrel Q) (vmatch′ p t1) (vmatch′ p t2)
  using assms(1) proof (induction p arbitrary: t1 t2)
  case (Patconstr name ps)
  have veq-structure t1 t2 by blast
  thus ?case proof (cases rule: veq-structure.cases)
    case (constr-constr ts us name′)
    { assume length ps = length ts
      have list-all2 (rel-option (fmrel Q)) (map2 vmatch′ ps ts) (map2 vmatch′ ps us)
        using (list-all2 veq-structure ts us) Patconstr (length ps = length ts)
        unfolding (t1 = ᾱ) (t2 = ᾱ)
        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 = ᾱ by simp
          moreover have list-all2 (rel-option (fmrel Q)) (map2 vmatch′ ps ts)
    }
  }

49
(map2 vmatch' ps us)

\textbf{apply} (rule Cons)

\textbf{subgoal}

\textbf{apply} (rule Cons)

\textbf{unfolding} \langle ps0 = \cdot \rangle \textbf{apply simp}

\textbf{by assumption}

\textbf{subgoal}

\textbf{using} \langle Q (Vconstr name' (t \# ts)) (Vconstr name' (u \# us)) \rangle \langle \text{length}

\textbf{ts} = \text{length us} \rangle

\textbf{by} (simp add: list-all-iff)

\textbf{subgoal}

\textbf{unfolding} \langle ps0 = \cdot \rangle \textbf{by simp}

\textbf{done}

\textbf{ultimately show} \ ?case

\textbf{unfolding} \langle ps0 = \cdot \rangle

\textbf{by} auto

\textbf{qed} auto

\textbf{hence rel-option} (list-all2 (fmrel Q)) (those (map2 vmatch' ps ts)) (those

(map2 vmatch' ps us))

\textbf{by} (rule rel-funD[OF those-transfer])

\textbf{have}

rel-option (fmrel Q)

(map-option (foldl (++) fnempty) (those (map2 vmatch' ps ts)))

(map-option (foldl (++) fnempty) (those (map2 vmatch' ps us)))

\textbf{apply} (rule rel-funD[OF rel-funD[OF option.map-transfer]])

\textbf{apply} transfer-prover

\textbf{by fact}

\}

\textbf{note \ast = this}

\textbf{have length ts = length us}

\textbf{using constr-constr by} (auto dest: list-all2-lengthD)

\textbf{thus} ?thesis

\textbf{unfolding} \langle t_1 = \cdot \rangle \langle t_2 = \cdot \rangle

\textbf{apply auto}

\textbf{apply} \ (rule \ast)

\textbf{by simp}

\textbf{qed auto}

\textbf{qed auto}

\textbf{lemma} vmatch-rel: \ Q t_1 t_2 \Rightarrow \text{rel-option} (fmrel Q) (vmatch p t_1) (vmatch p t_2)

\textbf{unfolding} vmatch-vmatch'-eq \textbf{by} (rule vmatch'-rel)

\textbf{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-match \( cs_2 \) \( t_2 \))
using \( \text{assms}(1) \) proof induction
  case (Cons \( c_1 \) \( cs_1 \) \( c_2 \) \( cs_2 \))
  moreover obtain \( \text{pat}_1 \) \( \text{rhs}_1 \) where \( c_1 = (\text{pat}_1, \text{rhs}_1) \) by fastforce
  moreover obtain \( \text{pat}_2 \) \( \text{rhs}_2 \) where \( c_2 = (\text{pat}_2, \text{rhs}_2) \) by fastforce
ultimately have \( \text{pat}_1 = \text{pat}_2 \ R \ \text{rhs}_1 \ \text{rhs}_2 \)
  by auto

have rel-option (fmrel \( Q \)) (vmatch (mk-pat \( \text{pat}_1 \)) \( t_1 \)) (vmatch (mk-pat \( \text{pat}_1 \)) \( t_2 \))
  by (rule vmatch-rel) fact
thus ?case
  proof cases
    case None
    thus ?thesis
      unfolding \( \langle c_1 = \cdot \rangle \ \langle c_2 = \cdot \rangle \ \langle \text{pat}_1 = \cdot \rangle \)
      using Cons by auto
next
    case (Some \( \Gamma_1 \) \( \Gamma_2 \))
    thus ?thesis
      unfolding \( \langle c_1 = \cdot \rangle \ \langle c_2 = \cdot \rangle \ \langle \text{pat}_1 = \cdot \rangle \)
      using \( \langle R \ \text{rhs}_1 \ \text{rhs}_2 \rangle \)
      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
dest-fact vmatch-vmatch'-eq
dest-const vmatch'
global-interpretation veq-structure: value-struct-rel veq-structure
  by standard auto
abbreviation env-eq where
  env-eq \( \equiv \) fmrel (\( \lambda v \ t. \ t = \text{value-to-sterm} \ v \) )
lemma env-eq-eq:
  assumes env-eq venv senv
  shows senv = fmmap value-to-sterm venv
proof (rule fmap-ext, unfold fmlookup-map)

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

context constructors begin

context begin

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

thus ?case
  apply (auto simp: strip-list-comb-const)
  apply (subst const-sterm-def, simp)+
done
qed auto

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

end

end

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

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

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

code-pred erelated .

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

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

thus \texttt{?case}

\textbf{apply} (auto intro: \texttt{erelated.intros \texttt{fmmpredI fmrel-on-fset-refl-strong}})

\textbf{apply} (auto intro: \texttt{fmran'I})

\textbf{done}

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

\textbf{export-code}

\texttt{value-to-term \texttt{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}

\textbf{end}
Chapter 2

A smaller version of CakeML: *CupCakeML*

theory Doc-CupCake
imports Main
begin

end

2.1 CupCake environments

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

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

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

fun is-cupcake-exp :: exp ⇒ bool where
is-cupcake-exp (Ast.Var (Short -)) ‐‐> True |
is-cupcake-exp (Ast.App oper es) ‐‐> oper = Ast.Opapp ∧ list-all is-cupcake-exp es |
is-cupcake-exp (Ast.Con (Some (Short -)) es) ‐‐> list-all is-cupcake-exp es |
is-cupcake-exp (Ast.Fun - e) ‐‐> is-cupcake-exp e |
is-cupcake-exp (Ast.Mut e 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 :: \((\text{Ast} \ast \text{exp}) \text{\ list} \Rightarrow \text{bool}\) where

cupcake-clauses \equiv \text{list-all } (\lambda(p, e). \text{is-cupcake-pat } p \land \text{is-cupcake-exp } e)\)

fun cupcake-c-ns :: c-ns \Rightarrow \text{bool} where
cupcake-c-ns (Bind cs mods) \iff mods = [] \land \text{list-all } (\lambda(-, -, \text{tid}). \text{case tid of } \text{TypeId } (\text{Short }-) \Rightarrow \text{True} | - \Rightarrow \text{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 = (| \text{sem-env.v} = \text{nsEmpty}, \text{sem-env.c} = \text{static-cenv} |)

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

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

fun is-cupcake-value :: SemanticPrimitives.v \Rightarrow \text{bool}
and is-cupcake-all-env :: all-env \Rightarrow \text{bool} where
is-cupcake-value (Conv (Some (\text{-}, \text{TypeId } (\text{Short }-))) \text{vs}) \iff \text{list-all is-cupcake-value} \text{vs} |
is-cupcake-value (\text{Closure env } - e) \iff is-cupcake-exp e \land \text{is-cupcake-all-env env} |
is-cupcake-value (\text{Reclosure env es } -) \iff \text{list-all } (\lambda(\text{-}, \text{-}, e). \text{is-cupcake-exp } e) \text{es} \land \text{is-cupcake-all-env env} |
is-cupcake-value - \iff \text{False} | is-cupcake-all-env (| \text{sem-env.v} = \text{Bind v0} [], \text{sem-env.c} = c0 |) \iff c0 = \text{static-cenv} \land \text{list-all } (\text{is-cupcake-value } \circ \text{snd} v0 |)
is-cupcake-all-env - \iff \text{False}

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

fun is-cupcake-ns :: v-ns \Rightarrow \text{bool} where
is-cupcake-ns (Bind v0 []) \iff \text{list-all } (\text{is-cupcake-value } \circ \text{snd} v0 |)
is-cupcake-ns - \iff \text{False}

lemma is-cupcake-nsE:
  assumes is-cupcake-ns ns
  obtains v where
  ns = \text{Bind v} [] \text{list-all } (\text{is-cupcake-value } \circ \text{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
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*pat)list ⇒((string*pat)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
            ...
          else...
        else...
      No _ => ...
    )
end
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 \( \text{cupcake-pmatch} \ cenv \ p \ v0 \ [] = \text{Match} \ env' \)

moreover have \( \text{pe} \# \text{pes} = ((p0, \ e0) \# \text{init}) @ (p, \ e) \# \text{rest} \)
unfolding \( \langle \text{pe} = \cdot \rangle \langle \text{pe} = \cdot \rangle \) by simp

moreover have \( \text{list-all} \ (\lambda (p, \ e). \ \text{cupcake-pmatch} \ cenv \ p \ v0 \ []) = \)
\( \text{No-match} \wedge \text{distinct} \ (\text{pat-bindings} \ p \ []) \) \((p0, \ e0) \# \text{init}\)
apply auto
subgoal by fact
subgoal using True by simp
subgoal using \( \langle \text{list-all} \ - \cdot \rangle \) by simp
done

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

ultimately show thesis
using Cons by blast

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

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

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

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

qed simp

lemma \( \text{cupcake-pmatch-eq} \):
\( \text{is-cupcake-pat} \ pat \ \Longrightarrow \ \text{pmatch-single} \ envC \ s \ pat \ v0 \ env = \text{cupcake-pmatch} \ envC \ pat \ v0 \ env \)

proof (induct rule: pmatch-single.induct)
case 4
from \( \text{is-cupcake-pat.elims}(2)[\text{OF 4}(2)] \) 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
done
qed auto

lemma cupcake-match-result-eq:
cupcake-clauses pes ⇒
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 =⇒ is-cupcake-value v0 =⇒ 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 ⇒ is-cupcake-ns ns2 ⇒
  is-cupcake-ns (nsAppend ns1 ns2)
by (auto clin!: is-cupcake-ns.elims)

lemma cupcake-alist-to-ns-preserve: list-all (is-cupcake-value o snd) env ⇒ 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]⟩ reclosure
        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]⟩ reclosure
        unfolding ⟨env = ⟩
        using cupcake-build-rec-preserve cupcake-nsBind-preserve cupcake-v-update-preserve
        is-cupcake-all-envD(1)
        by auto
      qed

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

61
by simp+
qed

context begin

lemma cup-pmatch-list-length-neq:
length vs ≠ length ps ⇒ Matching.fold2(λp v m. case m of
  | m ⇒ 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 ⇒ Matching.fold2(λp v m. case m of
  | m ⇒ 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 ⇒ Matching.fold2(λp v m. case m of
  | m ⇒ 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 ∀ p v env. p ∈ set ps ∧ v ∈ set vs → list-all (is-cupcake-value ◦ snd) env → if-match (list-all (is-cupcake-value ◦ snd)) (cupcake-pmatch cenv p v env) list-all (is-cupcake-value ◦ snd) env
shows if-match (list-all (λa. is-cupcake-value (snd a))) (Matching.fold2 (λp v m. case m of
  | m ⇒ 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 ◦ snd) env'
    using 4 by fastforce
  then show ?thesis
  apply (cases length ps = length vs)
using 4 Match by fastforce+

qed

qed (auto simp: comp-def)

private lemma cupcake-pmatch-preserve0:
  is-cupcake-pat pat \implies
  is-cupcake-value v0 \implies
  list-all (is-cupcake-value \circ snd) env \implies
  cupcake-c-ns envC \implies
  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 \implies is-cupcake-pat p for p
    using 2 by (metis Ball-set is-cupcake-pat.simps(2))
  have v: v \in set vs \implies is-cupcake-value v for v
    using 2 by (metis Ball-set is-cupcake-value.elims(2) v.distinct(11) v.distinct(13) v.inject(2))
  show ?case
    by (auto intro!: cupcake-pmatch-list-preserve split:if-splits option.splits) (metis 2 p v)+

end

lemma cupcake-pmatch-pmatch-preserve:
  is-cupcake-pat pat \implies
  is-cupcake-value v0 \implies
  list-all (is-cupcake-value \circ snd) env \implies
  cupcake-c-ns envC \implies
  cupcake-pmatch envC pat v0 env = Match env' \implies
  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 \implies
  cupcake-clauses pes \implies
  is-cupcake-value v \implies
  if-rval (\lambda (e, p, env'). is-cupcake-pat p \land is-cupcake-exp e \land list-all (is-cupcake-value \circ snd) env') env'
  (cupcake-match-result envC v pes err-v)
  apply (induction pes)
  apply (auto split: match-result.splits)
  apply (rule cupcake-pmatch-preserve)
  apply auto
  done

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

lemma cupcake-build-conv-preserve:  
  fixes v  
  assumes list-all is-cupcake-value vs build-conv static-cenv (Some (Short i)) vs = Some v  
  shows is-cupcake-value v  
using assms  
by (auto simp: build-conv simp splits 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  
  qed
\(\text{env}'\)
- \(\text{by} \ (\text{elim cupcake-match-resultE})\)
- \(\text{hence} \ (p, e) \in \text{set pes}\)
  - \(\text{by simp}\)
- \(\text{thus} \ \text{is-cupcake-exp e is-cupcake-pat p}\)
  - \(\text{using assms by} \ (\text{auto simp: list-all-iff})\)

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

\text{qed}

\text{end}

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

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

\text{lemma list-all2-shortcircuit-mono[mono]:}
- \(\text{assumes R } \leq Q\)
- \(\text{shows list-all2-shortcircuit R } \leq \text{list-all2-shortcircuit Q}\)
\text{proof}
- \(\text{fix } xs ys\)
- \(\text{assume list-all2-shortcircuit R xs ys}\)
- \(\text{thus list-all2-shortcircuit Q xs ys}\)
  - \(\text{using assms by (induction xs ys rule: list-all2-shortcircuit-induct) auto}\)
\text{qed}\)

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

\text{lemma list-all2-shortcircuit-rval[simp]:}
- \(\text{list-all2-shortcircuit P xs (map Rval ys) } \iff \text{list-all2 (}\lambda x y. P x (Rval y)) xs ys\)
proof
  assume ?lhs thus \(\text{?rhs}\)
  by (induction map \(\text{Rval ys::'b, 'c}\) result arbitrary: \(\text{ys rule: list-all2-shortcircuit-induct}\)) auto
next
  assume \(\text{?rhs}\) thus \(\text{?lhs}\)
  by (induction rule: \(\text{list-all2-induct}\)) auto
qed

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

\(\text{con1}:\)
\(\text{do-con-check (c env) cn (length es)} \Rightarrow \)
\(\text{list-all2-shortcircuit (cupcake-evaluate-single env) (rev es) rs} \Rightarrow \)
\(\text{sequence-result rs = Real vs} \Rightarrow \)
\(\text{build-conv (c env) cn (rev vs) = Some v0} \Rightarrow \)
\(\text{cupcake-evaluate-single env (Con cn es) (Real v0)} \mid \)
\(\text{con2}:\)
\(\neg \text{do-con-check (c env) cn (List.length es)} \Rightarrow \)
\(\text{cupcake-evaluate-single env (Con cn es) (Rerr (Rabort Rtype-error))} \mid \)
\(\text{con3}:\)
\(\text{do-con-check (c env) cn (List.length es)} \Rightarrow \)
\(\text{list-all2-shortcircuit (cupcake-evaluate-single env) (rev es) rs} \Rightarrow \)
\(\text{sequence-result rs = Rerr err} \Rightarrow \)
\(\text{cupcake-evaluate-single env (Con cn es) (Rerr err)} \mid \)
\(\text{var1}:\)
\(\text{nsLookup (sem-env.v env) n = Some v0} \Rightarrow \text{cupcake-evaluate-single env (Var n) (Real v0)} \mid \)
\(\text{var2}:\)
\(\text{nsLookup (sem-env.v env) n = None} \Rightarrow \text{cupcake-evaluate-single env (Var n) (Rerr (Rabort Rtype-error))} \mid \)
\(\text{fn}:\)
\(\text{cupcake-evaluate-single env (Fun n e) (Real (Closure env n e))} \mid \)
\(\text{app1}:\)
\(\text{list-all2-shortcircuit (cupcake-evaluate-single env) (rev es) rs} \Rightarrow \)
\(\text{sequence-result rs = Real vs} \Rightarrow \)
\(\text{do-opapp (rev vs) = Some (env', e)} \Rightarrow \)
\(\text{cupcake-evaluate-single env' e bv} \Rightarrow \)
\(\text{cupcake-evaluate-single env (App Opapp es) bv} \mid \)
\(\text{app3}:\)
\(\text{list-all2-shortcircuit (cupcake-evaluate-single env) (rev es) rs} \Rightarrow \)
\(\text{sequence-result rs = Rerr err} \Rightarrow \)
\(\text{cupcake-evaluate-single env (App op0 es) (Rerr err)} \mid \)
\(\text{mat1}:\)
cupcake-evaluate-single env e (Rval v0) \implies 
cupcake-match-result (c env) v0 pes Bindv = Rval (e', -, env') \implies 
cupcake-evaluate-single (env [] sem-env.v := nsAppend (alist-to-ns env') (sem-env.v env) |) e' bv \implies 
cupcake-evaluate-single env (Mat e pes) bv |

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

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

context cakeml-static-env begin

close context begin

private lemma cupcake-list-preserve0:
list-all2-shortcircuit
(\lambda e r. cupcake-evaluate-single env e r \land (is-cupcake-all-env env \implies is-cupcake-exp e \implies if-rval is-cupcake-value r)) es rs \implies 
is-cupcake-all-env env \implies list-all is-cupcake-exp es \implies sequence-result rs = Rval
vs \implies list-all is-cupcake-value vs

proof (induction es rs arbitrary; vs rule:list-all2-shortcircuit-induct)
case (cons-val - - - rs)
then obtain tid where cn:
cn = Some (Short tid)
and list-all is-cupcake-exp (rev es)
by (cases sequence-result rs) auto

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

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

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

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

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

show ?case
  unfolding vs
  by (rule; rule evaluate-list.cons1) fact+
qed (auto intro:evaluate-list.intros)

private lemma cupcake-list-correct-rerr:
  assumes list-all2-shortcircuit
  (λe r.
    cupcake-evaluate-single env e r ∧
    (is-cupcake-all-env env --> is-cupcake-exp e --> (∀ (s::'a state). ∃ s'. evaluate env s e (s', r))))
  es rs is-cupcake-all-env env list-all is-cupcake-exp es sequence-result rs = Rerr err

  shows ∃ 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
  (λe r.
    cupcake-evaluate-single env e r ∧
    (is-cupcake-all-env env --> is-cupcake-exp e --> (∀ (s::'a state). ∃ s'. evaluate env s e (s', r))))
  es rs is-cupcake-all-env env list-all is-cupcake-exp es

  shows ∃ 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 o snd) env' and e': is-cupcake-exp e'

using cupcake-match-result-preserve[where envC = c env and v = v0 and
pes = pes and err-v = Bindv, unfolded mat1, simplified]

by blast +

have env': is-cupcake-all-env (update-v (λ-. nsAppend (alist-to-ns env')) (sem-env.v env)) env)

apply (rule cupcake-v-update-preserve)
apply fact
apply (rule cupcake-nsAppend-preserve)
apply fact
apply (rule is-cupcake-all-envD)
apply fact
done

from e obtain s' where evaluate env s e (s', Rval v0)

using mat1 by blast

have match-result env s' v0 pes Bindv = Rval (e', env')

using mat1 cupcake-match-result-eq[OF pes, where env = env and v = v0
and err-v = Bindv and s = s']

by fastforce

from e' env' obtain s'' where evaluate (update-v (λ-. nsAppend (alist-to-ns env')) (sem-env.v env)) env) s' e' (s'', bv)

using mat1 by blast

show ?case

by rule + fact +

next
case (mat1error env e v0 pes err)

hence is-cupcake-exp e and pes: cupcake-clauses pes

by (auto dest: is-cupcake-all-envD)

then obtain s' where Big-Step-Unclocked-Single.evaluate env s e (s', Rval v0)

using mat1error by blast

hence match-result env s' v0 pes Bindv = Rerr err

using cupcake-match-result-eq[OF pes, where env = env and s = s' and v = v0 and err-v = Bindv] unfolding mat1error

by (simp add: error-result.map-id)

show ?case

by (rule; rule evaluate.mat1b) fact +

next
case (mat2 - e)

hence is-cupcake-exp e

by simp
then show ?case

using mat2 evaluate.mat2 by blast
qed (blast intro:evaluate.intros)+

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

private lemma cupcake-list-complete0:
evaluate-list
(\lambda s e r. evaluate env s e r \land (is-cupcake-all-env env \implies is-cupcake-exp e \implies
cupcake-evaluate-single env e (snd r))) s1 es res \implies
is-cupcake-all-env env \implies list-all is-cupcake-exp es \implies \exists rs. list-all2-shortcircuit
(cupcake-evaluate-single env) es rs \land sequence-result rs = (snd res)
proof (induction rule:evaluate-list.induct)
case empty
have list-all2-shortcircuit (cupcake-evaluate-single env) [] []
by fastforce
then show ?case
by fastforce
next
case (cons1 s1 e s2 v es s3 vs)
then obtain rs where list-all2-shortcircuit (cupcake-evaluate-single env) es rs
and sequence-result rs = Rval vs
and list-all2-shortcircuit (cupcake-evaluate-single env) (e # es) (Rval v # rs)
by fastforce+
then show ?case
by fastforce
next
case (cons2 s1 e s2 err es)
hence list-all2-shortcircuit (cupcake-evaluate-single env) (e # es) [Rerr err]
by simp
then show ?case
by fastforce
next
case (cons3 s1 e s2 v es s3 err)
then obtain rs where list-all2-shortcircuit (cupcake-evaluate-single env) es rs
and err:sequence-result rs = Rerr err
and list-all2-shortcircuit (cupcake-evaluate-single env) (e # es) (Rval v # rs)
by fastforce
moreover have sequence-result (Rval v # rs) = Rerr err
by (auto simp: error-result.map-id err)
ultimately show ?case
by fastforce
qed

private lemma cupcake-single-complete0:
evaluate env s e res \implies is-cupcake-all-env env \implies is-cupcake-exp e \implies 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 $x = \text{Con\ cn\ es}$]) 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 $x = \text{Con\ cn\ es}$]) 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)+

73
next
\text{case} (mat1 env s1 e s2 v1 pes e' env' bv)
\text{hence} \text{is-cupcake-exp} \; e \; \text{and} \; \text{cupcake-c-ns} \; (c \; \text{env}) \; \text{and} \; pes:\text{cupcake-clauses} \; pes
\text{and} \; \text{is-cupcake-value} \; v1
\text{by} \; \text{(fastforce dest: is-cupcake-all-envD cupcake-single-preserve)+}

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

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

ultimately have \; \text{cupcake-evaluate-single} \; \text{env} \; e \; (\text{Real v1})
\text{and} \; \text{cupcake-evaluate-single} \; (\text{update-v (λ-. nsAppend (alist-to-ns env')} \; (\text{sem-env.v env'}) \; \text{env}) \; e' \; (\text{snd bv})
\text{using} \; \text{mat1} \; \text{by} \; \text{fastforce+}

show \; ?case
\text{by} \; \text{(rule cupcake-evaluate-single.mat1) fact+}

next
\text{case} (mat1b env s1 e s2 v1 pes err)
\text{hence} \text{is-cupcake-exp} \; e \; \text{and} \; pes: \text{cupcake-clauses} \; pes
\text{by} \; \text{(auto dest: is-cupcake-all-envD)}

have \; \text{cupcake-evaluate-single} \; \text{env} \; e \; (\text{Real v1})
\text{using} \; \text{mat1b by} \; \text{fastforce}
\text{have} \; \text{cupcake-match-result} \; (c \; \text{env}) \; v1 \; pes \; \text{Bindv} = \text{Rerr} \; err
\text{using} \; \text{cupcake-match-result-eq[OF pes, where} \; \text{env} = \text{env and} \; s=s2 \; \text{and} \; v = v1 \; \text{and} \; err-v = \text{Bindv, unfolding mat1b]
\text{by} \; \text{(cases (cupcake-match-result (c env) v1 pes Bindv)) (auto simp:error-result.map-id)
}\text{show} \; ?case
\text{by} \; \text{(simp; rule cupcake-evaluate-single.mat1error) fact+}
\text{qed} \; \text{(fastforce intro: cupcake-evaluate-single.intros)+

74
lemma cupcake-single-complete:
  evaluate env s e (s', res) ⇒ is-cupcake-all-env env ⇒ is-cupcake-exp e ⇒
cupcake-evaluate-single env e res
  by (fastforce dest:cupcake-single-complete0)

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

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

lemma cupcake-state-preserve:
  assumes Big-Step-Unclocked-Single.evaluate env s e res is-cupcake-all-env env
  is-cupcake-exp e
  shows s = (fst res)
  using assms proof (induction arbitrary: rule: evaluate.induct)
  case (con1 env cn es vs v s1 s2)
  hence list-all is-cupcake-exp es
    by (cases rule: is-cupcake-exp.cases[where x = Con cn es]) auto
  then show ?case
    using con1 by (fastforce dest:cupcake-list-state-preserve0)
next
case (con3 env cn es s1 s2 err)
  hence list-all is-cupcake-exp es
    by (cases rule: is-cupcake-exp.cases[where x = Con cn es]) auto
  then show ?case
    using con3 by (fastforce dest:cupcake-list-state-preserve0)
next
case (app1 env s1 es s2 vs env' e bv)
  have list-all is-cupcake-exp (rev es)
    using app1 by fastforce
  then obtain rs where 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 \( \text{is-cupcake-exp} \ e \ \text{and} \ \text{cupcake-c-ns} \ (c \ env) \ \text{and} \ \text{pes:cupcake-clauses} \ \text{pes} \)
and \( \text{is-cupcake-value} \ v1 \)
by \( \text{fastforce dest: is-cupcake-all-envD} \ \text{cupcake-single-complete} \ \text{cupcake-single-preserve} \)

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

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

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

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

end end

hide-const (\text{open}) \ c

end
Chapter 3

Term rewriting

theory Doc-Rewriting
imports Main
begin

end

theory General-Rewriting
imports Terms-Extras
begin

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

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

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

lemma rt-comb:
R** t_1 u_1 ⇒ R** t_2 u_2 ⇒ 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
3.1 Higher-order term rewriting using de-Bruijn indices

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

3.1.1 Matching and rewriting

type-synonym rule = term × term

inductive rewrite :: rule fset ⇒ term ⇒ term ⇒ bool (-/- -→/ - [50,0,50] 50) for rs where
  step: r ∈ rs ⇒ r ⊢ t → u ⇒ rs ⊢ t → u |
  beta: rs ⊢ t → t'⇒ rs ⊢ t [t'/β] |
  fun: rs ⊢ t → t'⇒ rs ⊢ t $ u → t' $ u |
  arg: rs ⊢ u → 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'/β] = 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 ≠ {||}
assumes not-shadows: fBall rs (λ(lhs, -). ¬ shadows-consts lhs)
assumes welldefined-rs: fBall rs (λ(-, rhs). welldefined rhs)

begin

lemma rewrite-wellformed:
  assumes rs ⊢ t → 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 ⊢ t →∗ t' ⇒ wellformed t ⇒ wellformed t'
by (induction rule: rtranclp.induct) (auto intro: rewrite-wellformed simp del: wellformed-term-def)

lemma rewrite-closed: rs ⊢ t → t' ⇒ closed t ⇒ 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 |⊆| frees lhs
  by simp
  moreover have (lhs, rhs) ⊢ t → u
  using step unfolding ⟨r = -] by simp
  show ?case
  apply (rule rewrite-step-closed)
  apply fact+
  done
3.2 Higher-order term rewriting using explicit bound variable names

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

3.2.1 Definitions

type-synonym nrule = term × nterm

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

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

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

3.2.2 Matching and rewriting

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

fun: \( rs \vdash_n \ t \rightarrow t' \rightarrow rs \vdash_n \ t \ p n \ u \rightarrow \ t' p n \ u \ |
\)

arg: \( rs \vdash_n u \rightarrow u' \rightarrow rs \vdash_n \ t \ p n \ u \rightarrow t' p n \ u' \)

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/- \rightarrow*/
- [50,0,50] 50) where
nrewrite-rt rs ≡ (nrewrite rs)**

lemma (in nrules) nrewrite-closed:
- assumes \( rs \vdash_n \ t \rightarrow t' \) 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 \( |\subseteq| \) frees lhs
  by simp
  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
  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 \( rs \vdash_n \ t \rightarrow* t' \) 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)) \subseteq fset-of-list Γ |∪| frees t |∪| T ∧ fBall T (λy. y > x)
using assms proof (induction Γ t arbitrary; x rule: term-to-nterm-induct)
case (bound Γ t)
hence Γ ! i \in 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))) \subseteq |
      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 abs[where x = finsert (next x) T])
    apply (intro conjI)
    subgoal by auto
    subgoal using (all-frees (fst (run-state - (next x)))) \subseteq - by simp
    subgoal
      apply simp
      apply (rule conjI)
      apply (rule next-ge)
      using fBall ((<) (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)) \subseteq 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' \subseteq fset-of-list Γ |∪| frees t1 |∪| T1
  by simp
obtain T2
  where all-frees (fst (run-state (term-to-nterm Γ t2) x2)) \subseteq fset-of-list Γ |∪|

82
frees \( t_2 \upharpoonright \bigcup T_2 \)
and \( f\text{Ball} T_2 ((<) x_2) \)
apply atomize-clm
apply (rule app(2))
using app by auto
moreover obtain \( t_2' x' \) where run-state \((\text{term-to-nterm } \Gamma t_2) x_2 = (t_2', x')\)
by fastforce
ultimately have all-frees \( t_2' \subseteq f\text{set-of-list } \Gamma \upharpoonright \bigcup \frees t_2 \upharpoonright \bigcup T_2 \)
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 \((\text{term-to-nterm } \Gamma t_1) x_1 = \cdot\)
apply simp
unfolding :run-state \((\text{term-to-nterm } \Gamma t_2) x_2 = \cdot\)
apply simp
apply (rule ext[where \( x = T_1 \upharpoonright \bigcup T_2 \)])
apply (intro conjI)
subgoal using \( \text{all-frees } t_1' \subseteq \cdot \) by auto
subgoal using \( \text{all-frees } t_2' \subseteq \cdot \) by auto
subgoal
apply auto
using \( f\text{Ball} T_1 ((<) x_1) \) apply auto]
using \( f\text{Ball} T_2 ((<) x_2) \) \( x_1 \leq x_2 \)
by (meson fBallE less-le-not-le order-trans)
done
qed auto

lemma term-to-nterm-all-vars:
assumes wellformed \( t \ f\text{disjnt} \ (\frees t) S \)
shows \( f\text{disjnt} \ (\frees\ f\text{run} (\text{term-to-nterm } [] t) (T \upharpoonright \bigcup S))) S \)
proof

let \(?\Gamma = []\)
let \(?x = \text{fresh-fNext} (T \upharpoonright \bigcup S)\)
from assms have wellformed' \((\text{length } ?\Gamma) t \)
by simp
then obtain \( F \)
where \( \frees t \upharpoonright \bigcup \frees t S \)
and \( f\text{Ball} F (\lambda y. y > ?x) \)
by (metis term-to-nterm-all-vars0)

have \( f\text{disjnt} F (T \upharpoonright \bigcup S) \) if \( S \neq \{||\} \)
apply (rule fdisjnt-ge-max)
apply \((\text{rule } \text{Ball-pred-weaken}) \text{OF } - \langle \text{Ball } F (\lambda y. y > ?x) \rangle)\)
apply \((\text{rule } \text{less-trans})\)
apply \((\text{rule } \text{fNext-ge-max})\)
using that by auto
show \(?\text{thesis}\)
apply \((\text{rule fdisjnt-subset-left})\)
apply \((\text{subst fresh-frun-def})\)
apply \(\text{fact}\)
apply simp
apply \((\text{rule fdisjnt-union-left})\)
apply \(\text{fact}\)
using \((- = \Rightarrow \text{fdisjnt } F (T \cup S))\) by \(\text{(auto simp: fdisjnt-alt-def)}\)
qed
end

fun translate-rule :: name \(\Rightarrow\) rule \(\Rightarrow\) nrule where
translate-rule \(S\) \((\text{lhs}, \text{rhs})\) = \((\text{lhs}, \text{fresh-frun (term-to-nterm } [] \text{ rhs}) \text{ (frees lhs } \cup S))\)

lemma translate-rule-alt-def:
\(\text{translate-rule } S = (\lambda(\text{lhs}, \text{rhs}). (\text{lhs}, \text{fresh-frun (term-to-nterm } [] \text{ rhs}) \text{ (frees lhs } \cup S)))\)
by auto

definition compile' where
\(\text{compile'} \ C\text{-info } rs = \text{translate-rule (pre-constants.allconsts } C\text{-info } (\text{heads-of } rs)) \mid \mid \text{rs}\)

context rules begin

definition compile :: nrule \(\Rightarrow\) fset \text{where}
\(\text{compile} = \text{translate-rule all-consts } \mid \mid \text{rs}\)

lemma compile'-compile-eq[simp]: compile' \(\text{C-info } rs = \text{compile}\)
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 \(\text{C-info } compile\)
proof
  have \(\text{Ball } compile \ (\lambda(\text{lhs}, \text{rhs}). \text{nrule } (\text{lhs}, \text{rhs}))\)
    proof safe
    fix \(\text{lhs } \text{rhs}\)
    assume \((\text{lhs}, \text{rhs}) \in compile\)
    then obtain \(\text{rhs}'\)
where (lhs, rhs') \in rs

and rhs: rhs = fresh-frun (term-to-nterm [] rhs') (frees lhs \cup 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)) \sim is-const lhs by auto

have frees rhs \subseteq frees rhs'

unfolding rhs using rule

by (metis rule.simps term-to-nterm-vars)

also have frees rhs' \subseteq frees lhs

using rule by auto

finally show frees rhs \subseteq frees lhs.

qed

qed

thus fBall compile nrule

by fast

next

show arity-compatibles compile

proof safe

fix lhs_1 rhs_1 lhs_2 rhs_2

assume (lhs_1, rhs_1) \in compile (lhs_2, rhs_2) \in compile

then obtain rhs_1', rhs_2' where (lhs_1', rhs_1') \in rs (lhs_2, rhs_2') \in rs

unfolding compile-def by force

thus arity-compatible lhs_1 lhs_2

using arity by (blast dest: fpairwiseD)

qed

next

have is-fmap rs

using fmap by simp

thus is-fmap compile

unfolding compile-def translate-rule-alt-def

by (rule is-fmap-image)

next

have pattern-compatibles rs

using patterns by simp

thus pattern-compatibles compile

unfolding compile-def translate-rule-alt-def

by (auto dest: fpairwiseD)

next

show fdisjnt (heads-of compile) C

using disjnt by (simp add: compile-heads)

next

have fBall compile not-shadowing

85
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 \isasymGamma \isasymGamma u \isasymrightarrow u' closed u
  shows rs \isasymturnstile nterm-to-term' u \isasymrightarrow 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) \isasymin rs
      unfolding ⟨r = \isasymrightarrow⟩ compile-def by auto
    with assms have rule (pat, rhs)
      using all-rules by blast
    hence rhs = nterm-to-term [] rhs'
      unfolding ⟨rhs' = \isasymrightarrow⟩
      by (simp add: term-to-nterm-nterm-to-term fresh-frun-def)
  have compile \isasymturnstile \isasymGamma u \isasymrightarrow 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) \isasymturnstile nterm-to-term [] u \isasymrightarrow nterm-to-term [] u'
      apply (subst nterm-to-term-eq-closed)
      apply fact
      apply simp
apply \((\text{rule exI[where } x = \text{env}])\)
apply (rule conjI)
apply fact
unfolding \((\text{rhs} = \omega)\)
apply (subst nrelated-subst)
apply fact
unfolding \(\text{fdisjnt-alt-def} \) apply simp
unfolding \(u' = \text{subst rhs'}\text{env}\) 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[\text{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 \([\text{simp}] = \text{closed-except-def fdisjnt-alt-def}\)
begin

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

obtain pat rhs' where \(r = (\text{pat}, \text{rhs}')\)
by fastforce
with step obtain \( \text{env}' \) where match pat \( t = \text{Some} \text{env}' \) \( t' = \text{subst} \) \( \text{rhs}' \) \( \text{env}' \)

by fastforce

with \( (- = t) \) have rel-option \((\text{nrelated}.P.\text{env} [])\) \((\text{match} \) \( \text{pat} \) \( t \) \) \((\text{match} \) \( \text{pat} \) \( ?t_n \))\)

by \((\text{metis} \) \( \text{nrelated}.\text{match}-rel)\)

with \( \langle \) closed \( ?t_n \rangle \) have closed-env \( \text{env} \)

by \((\text{blast intro: closed.match})\)

from step obtain \( \text{rhs} \)

where \( \text{rhs} = \text{fresh-frun} \) \((\text{term-to-nterm} []) \) \( \text{rhs}' \) \((\text{frees} \) \( \text{pat} \) \( \cup \) \( \text{all-consts} \) \) \((\text{pat}, \) \( \text{rhs} \)) \( \in \) \( \text{compile} \)

unfolding \( \langle r = - \rangle \) \text{compile-def}

by force

with step have rule \((\text{pat}, \) \( \text{rhs}' \))

unfolding \( \langle r = - \rangle \)

using all-rules by fast

hence nterm-to-term' \( \text{rhs} = \text{rhs}' \)

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

by \((\text{simp add: fresh-frun-def term-to-nterm-nterm-to-term})\)

obtain \( u' \) where subst \( \text{rhs} \) \( \text{env} = u' \)

by simp

have \( t' = \text{nterm-to-term'} \) \( u' \)

unfolding \( \langle t' = - \rangle \)

unfolding \( \langle - = \text{rhs}' \rangle \) \([\text{symmetric}]\)

apply (\( \text{subst} \) \( \text{nrelated-subst} \))

apply fact+ 

using \( \langle - = u' \rangle \)

by simp+

have compile \( \vdash \) \( ?t_n \) \( \rightarrow \) \( u' \)

apply (rule nrewrite.step)

apply fact

apply simp

apply (intro conjI exI)

apply fact+

done

with \( \langle \text{closed} \) \( ?t_n \rangle \) have closed \( u' \)

by \((\text{blast intro: rules-as-nrules.nrewrite-closed})\)

with \( \langle t' = \text{nterm-to-term'} \rangle \) \( \rightarrow \) \( \text{have} \) \( u' \) \( \approx_{\alpha} \) \( ?t_n' \)

by \((\text{force intro: nterm-to-term-term-to-nterm}[\text{where} \Gamma = [] \) \text{and} \( \Gamma' = [], \text{simplified}])\)

show \( ?\text{case} \)

apply (intro conjI exI)

apply (rule nrewrite.step)

apply fact

apply simp

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

from beta have closed t closed (t [t'_?]β) closed (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} \ V frees ?t_n = fempty
  and closed ?t_n'
  using term-to-nterm-vars0[where Γ = [?name]]
  using term-to-nterm-vars0[where Γ = []]
  by (force simp: term-to-nterm-nterm-to-term0)+

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

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

show ?case
  apply (rule exI)
  apply (auto simp: split-beta create-alt-def)
  apply (rule rewrite.beta)
  apply (subst subst-single-eq[symmetric])
  apply (subst **)
  apply (rule nterm-to-nterm-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 ⊢ n fst (run-state (term-to-nterm [] t) s) −→ u'
    u' ≈_α n 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])

next

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

from arg obtain t' where compile |- n fst (run-state (term-to-nterm [] u) (snd (run-state (term-to-nterm [] t) s))) → t'
     t' ≈α nth (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

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

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

end

end

3.2.6 Splitting into constants

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

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

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

proof –

have \( \text{fBall } \text{rs} (\lambda (\text{pats}_1, -). \text{fBall } \text{rs} (\lambda (\text{pats}_2, -). \text{length } \text{pats}_1 = \text{length } \text{pats}_2)) \)
using \text{assms unfolding \text{fpairwise-alt-def} by blast}

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

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

using \text{assms(2) unfolding \text{arity-def \text{fthe-elem}'}-eq by \text{rule \text{singleton-fset-holds}}}

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

qed

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

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

assumes \text{fmap: is-fmap rs}

assumes \text{nonempty: } \text{rs} \neq \{||\}

assumes \text{inner:}

\( \text{fBall } \text{rs} (\lambda (-, \text{crs}). \text{arity-compatibles } \text{crs} \land 
\text{is-fmap } \text{crs} \land
\text{patterns-compatibles } \text{crs} \land 
\text{crs} \neq \{||\} \land
\text{fBall } \text{crs} (\lambda (\text{pats}, \text{rhs}). 
\text{linears } \text{pats} \land
\text{pats} \neq [] \land
\text{fdisjnt } (\text{freess } \text{pats}) \text{ all-consts} \land 
\neg \text{shadows-consts } \text{rhs} \land
\text{frees } \text{rhs} | \subseteq | \text{freess } \text{pats} \land
\text{welldefined } \text{rhs}) \)

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

assumes \( \forall \text{name } \text{crs} . (\text{name}, \text{crs}) |\in| \text{rs} \Rightarrow \text{arity-compatibles } \text{crs} \)

assumes \( \forall \text{name } \text{crs} . (\text{name}, \text{crs}) |\in| \text{rs} \Rightarrow \text{is-fmap } \text{crs} \)

assumes \( \forall \text{name } \text{crs} . (\text{name}, \text{crs}) |\in| \text{rs} \Rightarrow \text{patterns-compatibles } \text{crs} \)

assumes \( \forall \text{name } \text{crs} . (\text{name}, \text{crs}) |\in| \text{rs} \Rightarrow \text{crs} \neq \{||\} \)

assumes \( \forall \text{name } \text{crs } \text{pats } \text{rhs} . (\text{name}, \text{crs}) |\in| \text{rs} \Rightarrow (\text{pats}, \text{rhs}) |\in| \text{crs} \Rightarrow \text{linears } \text{pats} \)

assumes \( \forall \text{name } \text{crs } \text{pats } \text{rhs} . (\text{name}, \text{crs}) |\in| \text{rs} \Rightarrow (\text{pats}, \text{rhs}) |\in| \text{crs} \Rightarrow \text{pats} \neq [] \)

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

assumes \( \forall \text{name } \text{crs } \text{pats } \text{rhs} . (\text{name}, \text{crs}) |\in| \text{rs} \Rightarrow (\text{pats}, \text{rhs}) |\in| \text{crs} \Rightarrow \neg \text{shadows-consts } \text{rhs} \)

assumes \( \forall \text{name } \text{crs } \text{pats } \text{rhs} . (\text{name}, \text{crs}) |\in| \text{rs} \Rightarrow (\text{pats}, \text{rhs}) |\in| \text{crs} \Rightarrow \text{frees } \text{rhs} | \subseteq | \text{freess } \text{pats} \)

assumes \( \forall \text{name } \text{crs } \text{pats } \text{rhs} . (\text{name}, \text{crs}) |\in| \text{rs} \Rightarrow (\text{pats}, \text{rhs}) |\in| \text{crs} \Rightarrow \text{welldefined } \text{rhs} \)

assumes \text{is-fmap } \text{rs } \text{rs} \neq \{||\} \)

92
shows crules C-info rs using assms unfolding crules-axioms-def crules-def by (auto simp: prod-fBallI intro: pre-crules-axioms)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

94
have \( \text{lhs}_1 = \text{lhs}_2 \)
proof (rule ccontr)
  assume \( \text{lhs}_1 \neq \text{lhs}_2 \)
  then consider \( \text{fst} \) \( \text{fst} (\text{strip-comb} \text{lhs}_1) \neq \text{fst} (\text{strip-comb} \text{lhs}_2) \)  
  | \( \text{snd} \) \( \text{snd} (\text{strip-comb} \text{lhs}_1) \neq \text{snd} (\text{strip-comb} \text{lhs}_2) \)
  by (metis list-strip-comb)
thus False
proof cases
  case \( \text{fst} \)
  moreover have is-const \( \text{fst} (\text{strip-comb} \text{lhs}_1) \) is-const \( \text{fst} (\text{strip-comb} \text{lhs}_2) \)
  using rs all-rules by force+
  ultimately show ?thesis
    using split const-name-simps by (fastforce simp: split-beta)
next
  case \( \text{snd} \)
  with split show ?thesis
    by (auto simp: split-beta)
qed
qed

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

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

have \( \neg \) is-const \( \text{lhs} \) is-const \( \text{fst} (\text{strip-comb} \text{lhs}) \)
  using \( \neg \text{nrule} \cdot \) by auto
thus pats \( \neq \) []
  unfolding \( \text{pats} = \cdot \) using (linear \( \text{lhs} \))
  apply (cases \( \text{lhs} \))
    apply (fold app-term-def)
  by (auto split: prod.splits)

from \( \neg \text{nrule} (\text{lhs}, \text{rhs}) \) have frees \( \text{fst} (\text{strip-comb} \text{lhs}) \) = []
  by (cases \( \text{fst} (\text{strip-comb} \text{lhs}) \)) (auto simp: is-const-def)
  hence frees \( \text{lhs} \) = freess \( \text{snd} (\text{strip-comb} \text{lhs}) \)
  by (subst frees-strip-comb) auto
thus \( \text{frees rhs} \subseteq \text{frees pats} \)

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

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

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

by force

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

by (simp add: consts-of-heads)

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

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

by (force simp: shadows-consts-def)

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

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

by simp

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

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

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

by (simp add: consts-of-heads)

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

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

by (force simp: consts-of-heads)

qed

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

3.2.7 Computability

export-code

\begin{itemize}
\item translate-rule consts-of arity nterm-to-term
\item checking Scala
\end{itemize}

end

3.3 Higher-order term rewriting with explicit pattern matching

theory Rewriting-Pterm-Elim

imports

\begin{itemize}
\item Rewriting-Nterm
\item ../Terms/Pterm
\end{itemize}

begin

3.3.1 Intermediate rule sets

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

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

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

lemma (in pre-irules) irulesI:
  assumes ∀name irs. (name, irs) ∈| rs ⇒ arity-compatibles irs
  assumes ∀name irs. (name, irs) ∈| rs ⇒ is-fmap irs
  assumes ∀name irs. (name, irs) ∈| rs ⇒ patterns-compatibles irs
  assumes ∀name irs. (name, irs) ∈| rs ⇒ irs ≠ {}{}
  assumes ∀name irs pats rhs. (name, irs) ∈| rs ⇒ (pats, rhs) ∈| irs ⇒ linears pats
  assumes ∀name irs pats rhs. (name, irs) ∈| rs ⇒ (pats, rhs) ∈| irs ⇒ abs-ish pats rhs
  assumes ∀name irs pats rhs. (name, irs) ∈| rs ⇒ (pats, rhs) ∈| irs ⇒ fdisjnt (freess pats) all-consts
  assumes ∀name irs pats rhs. (name, irs) ∈| rs ⇒ (pats, rhs) ∈| irs ⇒ closed-except rhs (freess pats)
  assumes ∀name irs pats rhs. (name, irs) ∈| rs ⇒ (pats, rhs) ∈| irs ⇒ wellformed rhs
  assumes ∀name irs pats rhs. (name, irs) ∈| rs ⇒ (pats, rhs) ∈| irs ⇒ ¬ shadows-consts rhs
  assumes ∀name irs pats rhs. (name, irs) ∈| rs ⇒ (pats, rhs) ∈| irs ⇒ welldefined rhs
  assumes is-fmap rs rs ≠ {}{}
  shows irules C-info rs
using assms unfolding irules-axioms-def irules-def
by (auto simp: prod-fBallI intro: pre-irules-axioms)

lemmas irulesI[intro!] = pre-irules.irulesI[unfolded pre-irules-def]
Translation from nterm to pterm

fun nterm-to-pterms :: nterm ⇒ pterm where
nterm-to-pterms (Nvar s) = Pvar s |
nterm-to-pterms (Nconst s) = Pconst s |
nterm-to-pterms (t, $, n, t') = nterm-to-pterms t, $, nterm-to-pterms t' |
nterm-to-pterms (Λ x. t) = (Λ x. nterm-to-pterms t)

lemma nterm-to-pterms-inj: nterm-to-pterms x = nterm-to-pterms y ⇒ x = y
by (induction y arbitrary: x) (auto elim: nterm-to-pterms.elims)

lemma nterm-to-pterms:
  assumes no-abs t
  shows nterm-to-pterms 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-pterms-frees[simp]: frees (nterm-to-pterms t) = frees t
by (induct t) auto

lemma closed-nterm-to-pterms[intro]: closed-except (nterm-to-pterms t) (frees t)
unfolding closed-except-def by simp

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

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

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

Translation from crule-set to irule-set

definition translate-crules :: crules ⇒ irules where
translate-crules = fmap (map-prod id nterm-to-pterms)

definition compile :: crule-set ⇒ irule-set where
compile = fmap (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 (blast dest: fpairwiseD)
thus arity-compatibles irs
  unfolding irs = translate-crules irs' translate-crules-def
  by (force dest: fpairwiseD)

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

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

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

fix pats rhs
assume (pats, rhs) |∈| irs
then obtain rhs' where (pats, rhs') |∈| irs' rhs = nterm-to-pterms rhs'
  unfolding ⟨irs = translate-crules irs'⟩ translate-crules-def by force
hence $\text{linears} \ pats \ pats \neq [] \ \text{frees} \ rhs' \ \subseteq \ \text{freess} \ pats \ \leadsto \ \text{shadows-consts} \ rhs'$

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

by blast+

show $\text{linears} \ pats$ by fact

show $\text{closed-except} \ rhs \ (\text{freess} \ pats)$

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

by (metis dual-order.trans closed-nterm-to-pterm closed-except-def)

show $\text{wellformed} \ rhs$

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

by blast

thus $\text{fdisjnt} \ (\text{freess} \ pats) \ (\text{pre-constants}.\text{all-consts} \ C\text{-info} \ (\text{fst} \ |'| \ \text{compile} \ rs))$

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

by blast

have $\neg \ \text{shadows-consts} \ rhs$

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

by simp

thus $\neg \ \text{pre-constants}.\text{shadows-consts} \ C\text{-info} \ (\text{fst} \ |'| \ \text{compile} \ rs)$

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

by simp

show $\text{abs-ish} \ pats \ rhs$

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

by simp

have $\text{welldefined} \ rhs'$

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

by blast

thus $\text{pre-constants}.\text{welldefined} \ C\text{-info} \ (\text{fst} \ |'| \ \text{compile} \ rs)$

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

by simp

qed

sublocale crules $\subseteq$ crules-as-irules: irules $C\text{-info} \ \text{compile} \ rs$

by (fact compile-rules)

Transformation of irule-set

definition transform-irules :: irules $\Rightarrow$ irules where

transform-irules $rs$ = ($\lambda$)

if arity $rs$ = 0 then $rs$

else map-prod id $Pabs$ |'| $fgroup-by$ ($\lambda$(pats, rhs)). (butlast pats, (last pats, rhs))) $rs$

lemma arity-compatibles-transform-irules:
assumes \( \text{arity-compatibles \( rs \)} \)
shows \( \text{arity-compatibles} \ (\text{transform-irules \( rs \)}) \)
proof (cases \( \text{arity \( rs = 0 \)} \))
case True
  thus \(?thesis\)
    unfolding \text{transform-irules-def} using \assms by simp
next
case False
let \(?rs' = \text{transform-irules \( rs \)}\)
let \(?f = \lambda (\text{pats, rhs}). (\text{butlast pats,} \ (\text{last pats, rhs}))\)
let \(?grp = \text{fgroup-by} \ ?f \)
have \( rs' : ?rs' = \text{map-prod id Pabs |'} ?grp \)
  using False unfolding \text{transform-irules-def} by simp
show \(?thesis\)
proof safe
  fix \( pats_1 \) \( rhs_1 \) \( pats_2 \) \( rhs_2 \)
  assume \( (pats_1, rhs_1) \ |\ \( pats_2, rhs_2 \) \ |\ \( ?rs' \)
  then obtain \( rhs_1' \) \( rhs_2' \)
    where \( (pats_1, rhs_1') \ |\ \( pats_2, rhs_2' \) \ |\ \( ?grp \)
    unfolding \( rs' \) by auto
  then obtain \( pats_1', pats_2' \ x y \) — dummies
    where \( \text{fst} (\ ?f (pats_1', x)) = pats_1 (pats_1', x) \ |\ \( rs \)
    and \( \text{fst} (\ ?f (pats_2', y)) = pats_2 (pats_2', y) \ |\ \( rs \)
    by (fastforce simp: split-beta elim: fgroup-byE2)
  hence \( pats_1 = \text{butlast pats}_1' \ pats_2 = \text{butlast pats}_2' \ |\ \( ?grp \)
    using \assms by (force dest: fpairwiseD)+
  thus \( \text{length} pats_1 = \text{length} pats_2 \)
    by auto
  qed
qed

lemma \text{arity-transform-irules}:
assumes \( \text{arity-compatibles} \ (rs \neq \{||\}) \)
shows \( \text{arity} (\text{transform-irules \( rs \)}) = (\text{if arity \( rs = 0 \) then 0 else arity \( rs - 1 \)}) \)
proof (cases \( \text{arity \( rs = 0 \}) \)
  case True
    thus \(?thesis\)
    unfolding \text{transform-irules-def} by simp
next
case False
let \(?f = \lambda (\text{pats, rhs}). (\text{butlast pats,} \ (\text{last pats, rhs}))\)
let \(?grp = \text{fgroup-by} \ ?f \)
let \( ?rs' = \text{map-prod id Pabs |'} ?grp \)
have \( \text{arity} \ ?rs' = \text{arity} \ ?rs - 1 \)
  proof (rule \text{arityI})
    show \( \text{fBall} ?rs' (\lambda (\text{pats, -}). \text{length} pats = \text{arity} \ ?rs - 1) \)
      proof (rule \text{prod-fBallI})
        fix \pats \ rhs
  qed
assume \((\text{pats}, \text{rhs}) \in \text{?rs}'\)
then obtain \(\text{cs}\) where \((\text{pats}, \text{cs}) \in \text{?grp} \text{ rhs} = \text{Pabs cs}\)
    by force
then obtain \(\text{pats'} x \leftarrow \text{dummy}\)
    where \(\text{pats} = \text{butlast pats'} (\text{pats'}, x) \in \text{rs}\)
    by (fastforce simp: split-beta elim: fgroup-byE2)
    hence \(\text{length pats'} = \text{arity rs}\)
    using assms by (metis arity-compatible-length)
    thus \(\text{length pats} = \text{arity rs} - 1\)
    unfolding \((\text{pats} = \text{butlast pats'})\) using False by simp
qed

next
show \(\text{?rs}' \neq \{||\}\)
    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 \Rightarrow irule-set
    where
        transform-irule-set = fimage \((\text{map-prod id transform-irules})\)

lemma transform-irule-set-heads: \(\text{fst} |'| \text{transform-irule-set rs} = \text{fst} |'| \text{rs}\)
    unfolding transform-irule-set-def by simp

lemma (in irules) rules-transform: irules C-info \((\text{transform-irule-set rs})\)
    proof
        have \(\text{is-fmap rs}\)
            using fmap by simp
        thus \(\text{is-fmap (transform-irule-set rs)}\)
            unfolding transform-irule-set-def map-prod-def id-apply by (rule is-fmap-image)
        show \(\text{transform-irule-set rs} \neq \{||\}\)
            using nonempty unfolding transform-irule-set-def by auto
        show constants C-info \((\text{fst} |'| \text{transform-irule-set rs})\)
            proof
                show \(\text{disjnt (fst |'| \text{transform-irule-set rs}) C}\)
                    using disjnt unfolding transform-irule-set-def
                    by force
                next
                show \(\text{distinct all-constructors}\)
                    by (fact distinct-ctr)
                qed
        fix name irs
        assume irs: \((\text{name}, \text{irs}) \in \text{transform-irule-set rs}\)
        then obtain \(\text{irs'}\) where \((\text{name}, \text{irs'}) \in \text{rs} \text{ irs} = \text{transform-irules irs'}\)
unfolding transform-irule-set-def by force
hence arity-compatibles irs'
  using inner by (blast dest: fpairwiseD)
thus arity-compatibles irs
unfolding irs = transform-irules irs' by (rule arity-compatibles-transform-irules)

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

show is-fmap \text{irs}

proof (cases arity \text{irs}' = 0)
  case True
  thus ?thesis
    unfolding \langle \text{irs} = \text{transform-irules} \text{irs}' \rangle \text{transform-irules-def}
    using \langle is-fmap \text{irs}' \rangle \text{ by simp}

next
  case False
  hence \text{irs'}: \text{irs} = \text{map-prod id Pabs} |\cdot| ?\text{grp}
    unfolding \langle \text{irs} = \text{transform-irules} \text{irs}' \rangle \text{transform-irules-def} \text{ by simp}

show ?thesis

proof
  fix \text{pats} \text{rhs}_1 \text{rhs}_2
  assume \langle\text{pats}, \text{rhs}_1\rangle |∈| \text{irs} (\text{pats}, \text{rhs}_2) |∈| \text{irs}
  with \text{irs'} obtain \text{cs}_1 \text{cs}_2
    where \langle\text{pats}, \text{cs}_1\rangle |∈| ?\text{grp} \text{rhs}_1 = \text{Pabs cs}_1
    and \langle\text{pats}, \text{cs}_2\rangle |∈| ?\text{grp} \text{rhs}_2 = \text{Pabs cs}_2
    by force
  moreover have is-fmap ?\text{grp}
    by auto
  ultimately show \text{rhs}_1 = \text{rhs}_2
    by (auto dest: is-fmapD)
  qed

qed

fix \text{pats} \text{rhs}
assumes \langle\text{pats}, \text{rhs}\rangle |∈| \text{irs}

show linears \text{pats}

proof (cases arity \text{irs}' = 0)
  case True
  thus ?thesis
    using \langle\text{pats}, \text{rhs}\rangle |∈| \text{irs} \langle\text{name}, \text{irs}'⟩ |∈| \text{rs} \text{ inner}
    unfolding \langle \text{irs} = \text{transform-irules} \text{irs}' \rangle \text{transform-irules-def}
    by (smt fBallE split-conv)

next
  case False
  hence \text{irs'}: \text{irs} = \text{map-prod id Pabs} |\cdot| ?\text{grp}
    unfolding \langle \text{irs} = \text{transform-irules} \text{irs}' \rangle \text{transform-irules-def} \text{ by simp}
  then obtain \text{cs} where \langle\text{pats}, \text{cs}\rangle |∈| ?\text{grp}
    using \langle\text{pats}, \text{rhs}\rangle |∈| \text{irs} \text{ by force}
  then obtain \text{pats'} \text{x} — dummy
    where \text{fst} (\langle\text{if} \text{pats}', \text{x}\rangle) = \text{pats} (\text{pats'}, \text{x}) |∈| \text{irs}'
    by (fastforce simp: split-beta elim: fgroup-byE2)
  hence \text{pats} = \text{butlast pats'}
    by simp
  moreover have linears \text{pats'}
using \langle (pats', x) \mid \in \mid irs \rangle \langle (name, irs') \mid \in \mid rs \rangle \text{ inner}

by blast
ultimately show \?thesis
by auto
qed

have \text{fdisjnt} (\text{freess} \ pats) \text{ all-consts}
proof (cases arity \text{irs}' = 0)
case True
thus \?thesis
using \langle (pats, rhs) \mid \in \mid \text{irs} \rangle \langle (name, irs') \mid \in \mid rs \rangle \text{ inner}

unfolding \langle \text{irs} = \text{transform-rules} \ irs' \rangle \text{ transform-rules-def}
by (smt fBallE split-conv)

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

unfolding \langle \text{irs} = \text{transform-rules} \ irs' \rangle \text{ transform-rules-def by simp}
then obtain \ cs \ where \langle (pats, cs) \mid \in \mid \text{irs} \rangle \ ?grp

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

then obtain \text{pats}' \ x — \ dummy
where \text{fst} (\langle \text{pats}', x \rangle) = \text{pats} \langle \text{pats}', x \rangle \mid \in \mid \text{irs}'
by (fastforce simp; split-beta elim: fgroup-byE2)

hence \text{pats} = \text{butlast} \ pats'
by simp

moreover have \text{fdisjnt} (\text{freess} \ pats') \text{ all-consts}
using \langle (pats', x) \mid \in \mid irs' \rangle \langle (name, irs') \mid \in \mid rs \rangle \text{ inner by blast}
ultimately show \?thesis
by (metis subsetI in-set-butlastD freess-subset fdisjnt-subset-left)
qed

thus \text{fdisjnt} (\text{freess} \ pats) \langle \text{pre-constants. all-consts} \ C-info (\text{fst} |' \ | \\text{transform-irule-set} \ \text{rs}) \rangle

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

show \text{closed-except} \ rhs \ (\text{freess} \ pats)
proof (cases arity \text{irs}' = 0)
case True
thus \?thesis
using \langle (pats, rhs) \mid \in \mid \text{irs} \rangle \langle (name, irs') \mid \in \mid rs \rangle \text{ inner}

unfolding \langle \text{irs} = \text{transform-rules} \ irs' \rangle \text{ transform-rules-def}
by (smt fBallE split-conv)

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

unfolding \langle \text{irs} = \text{transform-rules} \ irs' \rangle \text{ transform-rules-def by simp}
then obtain \ cs \ where \langle (pats, cs) \mid \in \mid \text{irs} \rangle \ ?grp \ rhs = \text{Pabs} \ cs

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

show \?thesis
unfolding \(\langle \text{rhs} = \text{Pabs cs} \rangle\) closed-except-simps

proof safe
  fix pat t
  assume \(\langle \text{pat}, t \rangle \mid \in \mid \text{cs}\)
  then obtain \(\text{pats'}\) where \(\langle \text{pats'}, t \rangle \mid \in \mid \text{irs'}\)
    ?f \(\langle \text{pats}, \text{pat}, t \rangle \rangle\)
    using \(\langle \text{pats}, \text{cs} \rangle \mid \in \mid \text{?grp} \rangle\) by auto
  hence closed-except t (freess \(\text{pats'}\))
    using \(\langle \text{name}, \text{irs'} \rangle \mid \in \mid \text{?grp} \rangle\) by auto
  thus closed-except t (freess \(\text{pats'}\))
    using \(\langle \text{pats}, \text{cs} \rangle \mid \in \mid \text{?grp} \rangle\) by auto
qed

show wellformed \(\text{rhs}\)
proof (cases arity \(\text{irs'} = 0\))
  case True
    thus ?thesis
      using \(\langle \text{pats}, \text{rhss} \rangle \mid \in \mid \text{irs} \rangle\) \(\langle \text{name}, \text{irs'} \rangle \mid \in \mid \text{rs} \rangle\) inner by auto
next
  case False
  hence \(\text{irs'}: \text{irs} = \text{map-prod id Pabs} \mid \mid \text{?grp} \rangle\)
    unfolding \(\langle \text{irs} = \text{transform-irules} \text{irs'} \rangle\) transform-irules-def by simp
  then obtain \(\text{cs}\) where \(\langle \text{pats}, \text{cs} \rangle \mid \in \mid \text{?grp} \rangle\)
    ?f \(\text{rhs} = \text{Pabs cs}\)
    using \(\langle \text{pats}, \text{rhss} \rangle \mid \in \mid \text{irs} \rangle\) by force
show ?thesis
unfolding \(\langle \text{rhs} = \text{Pabs cs} \rangle\)
proof (rule wellformed-PabsI)
  show \(\text{cs} \neq \{\mid \mid \}\)
    using \(\langle \text{pats}, \text{cs} \rangle \mid \in \mid \text{?grp} \rangle\) \(\langle \text{irs'} \neq \{\mid \mid \}\rangle\)
    by (meson femptyE fgroup-by-nonempty-inner)
next
  show is-fmap cs
    proof
      fix pat \(t_1, t_2\)
      assume \(\langle \text{pat}, t_1 \rangle \mid \in \mid \text{cs} \rangle\)
      \(\langle \text{pat}, t_2 \rangle \mid \in \mid \text{cs} \rangle\)
      then obtain \(\text{pats}_1', \text{pats}_2'\)
\begin{verbatim}
where (pats_1', t_1) \in| irs \iff (pats_1', t_1) = (pats, (pat, t_1))
and (pats_2', t_2) \in| irs \iff (pats_2', t_2) = (pats, (pat, t_2))
using \langle pats, cs \rangle \in| \langle ?grp \by force \rangle
moreover hence pats_1' \neq [] pats_2' \neq []
using \langle \text{arity-compatibles} irs \rangle False
unfolding prod.case
by (metis list.size(3) \text{arity-compatible-length}+)
ultimately have pats_1' = pats @ [pat] pats_2' = pats @ [pat]
unfolding split-beta fst-cone snd-conv
by (metis prod.inject snoc-eq-iff-butlast)+
with \langle is-fmap irs \rangle show t_1 = t_2
using \langle pats_1', t_1 \rangle \in| irs \ (pats_2', t_2) \in| irs
by (blast dest: is-fmapD)
qed
next
show \text{pattern-compatibles} cs
proof safe
fix pat_1 rhs_1 pat_2 rhs_2
assume (pats_1, rhs_1) \in| cs (pat_2, rhs_2) \in| cs
then obtain pats_1' pats_2'
where (pats_1', rhs_1) \in| irs \iff (pats_1', rhs_1) = (pats, (pat_1, rhs_1))
and (pats_2', rhs_2) \in| irs \iff (pats_2', rhs_2) = (pats, (pat_2, rhs_2))
using \langle pats, cs \rangle \in| \langle ?grp \rangle
by force
moreover hence pats_1' \neq [] pats_2' \neq []
using \langle \text{arity-compatibles} irs \rangle False
unfolding prod.case
by (metis list.size(3) \text{arity-compatible-length}+)
ultimately have pats_1' = pats @ [pat_1] pats_2' = pats @ [pat_2]
unfolding split-beta fst-cone snd-conv
by (metis prod.inject snoc-eq-iff-butlast)+
moreover have \text{patterns-compatible} pats_1' pats_2'
using \langle pats_1', rhs_1 \rangle \in| irs \ (pats_2', rhs_2) \in| irs \ \langle \text{patterns-compatibles} irs \rangle
by (auto dest: fpairwiseD)
ultimately show \text{pattern-compatible} pat_1 pat_2
by (auto elim: rev-accum-rel-snoc-eqE)
qed
next
fix pat
assume (pat, t) \in| cs
then obtain pats' where (pats', t) \in| irs' pat = last pats'
using \langle pats, cs \rangle \in| ?grp \ by auto
moreover hence pats' \neq []
using \langle \text{arity-compatibles} irs \rangle False
by (metis list.size(3) \text{arity-compatible-length})
ultimately have pat \in set pats'
by auto
\end{verbatim}
moreover have linears pats'
  using ⟨(pats', t) |∈| irs'⟩ ⟨(name, irs') |∈| rs |∈⟩ inner by blast
ultimately show linear pat
  by (metis linears-linear)

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

have ¬ shadows-consts rhs
proof (cases arity irs' = 0)
case True
thus ?thesis
  using ⟨(pats, rhs) |∈| irs⟩ ⟨(name, irs') |∈| rs |∈⟩ inner
  unfolding ⟨irs = transform-irules irs'⟩ transform-irules-def
  by (smt fBallE split-conv)
next
case False
hence irs': irs = map-prod id Pabs ∘ ?grp
  unfolding ⟨irs = transform-irules irs'⟩ transform-irules-def by simp
then obtain cs where ⟨(pats, cs) |∈| ?grp rhs = Pabs cs
  using ⟨(pats, rhs) |∈| irs⟩ by force
show ?thesis
  unfolding ⟨?rhs = ⊥⟩
  proof
    assume shadows-consts (Pabs cs)
    then obtain pat t where ⟨(pat, t) |∈| cs shadows-consts t |∈⟩ shadows-consts
      pat
        by force
    then obtain pats' where ⟨(pats', t) |∈| irs'⟩ ?grp: by auto
    moreover hence pats' ≠ []
      using ⟨arity-compatibles irs'⟩ False
      by (metis list.size(3) arity-compatible-length)
ultimately have pat ∈ set pats'
      by auto

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

have fdisjnt (freess pats') all-consts
using ⟨\{name, irs′\} |∈| rs⟩ ⟨\{pats′, t\} |∈| irs′⟩ inner by blast
have fdisjnt (frees pat) 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

thus \neg\text{pre-constants.shadows-consts C-info (fst |\_| transform-irule-set rs) rhs}
by (simp add: transform-irule-set-heads)

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

have welldefined rhs
proof (cases arity irs′ = 0)
case True
hence ⟨\{pats, rhs\} |∈| irs′⟩
using ⟨\{pats, rhs\} |∈| irs⟩ ⟨\{name, irs′\} |∈| rs⟩ inner
unfolding ⟨irs = transform-irules irs′; transform-irules-def⟩
by (smt fBallE split-conv)
thus ?thesis
unfolding transform-irule-set-def
using fbspec[OF inner ⟨\{name, irs′\} |∈| rs⟩, simplified]
by force
next
case False
hence irs′: irs = map-prod id Pabs |\_| ?grp
unfolding \( \text{irs} = \text{transform-irules} \text{irs}' \) \text{transform-irules-def} by simp
then obtain \( \text{cs} \) where \( (\text{pats}, \text{cs}) \in \equiv \text{?grp} \text{rhs} = \text{Pabs} \text{cs} \)
using \( (\text{pats}, \text{rhs}) \in \equiv \text{irs} \) by force
show \(?\text{thesis}\)
unfolding \( \text{rhs} = - \)
apply simp
apply (rule \text{ffUnion-least})
unfolding \text{ball-simps}
apply rule
apply (rename-tac \(x\), case-tac \(x\), hypsubst-thin)
apply simp
subgoal premises \( \text{prems} \) for \( \text{pat} t \)
proof –
from \( \text{prems} \) obtain \( \text{pats}' \) where \( (\text{pats}', \text{t}) \in \equiv \text{irs}' \)
using \( (\text{pats}, \text{cs}) \in \equiv \text{?grp} \) by auto
hence welldefined \( t \)
using \( \text{fbspec}[\text{OF inner \langle \text{name}, \text{irs}' \rangle \in \equiv \text{rs}, \text{simplified}] \) by blast
thus \(?\text{thesis}\)
unfolding \text{transform-irule-set-def}
by simp
qed
done
thus \text{pre-constants.welldefined C-info (fst \[|\] transform-irule-set \text{rs}) \text{rhs}}
unfolding \text{transform-irule-set-heads} .
qed

Matching and rewriting

definition \text{rewrite-step} :: \text{name} \Rightarrow \text{term list} \Rightarrow \text{pterm} \Rightarrow \text{pterm} \Rightarrow \text{pterm option}
where
abbreviation \text{rewrite-step}' :: \text{name} \Rightarrow \text{term list} \Rightarrow \text{pterm} \Rightarrow \text{pterm} \Rightarrow \text{pterm} \Rightarrow \text{bool} (-, -, - \vdash - \vdash - \vdash - [50,0,50] 50) \text{ where}
\text{name, pats, rhs} \vdash \text{t} \rightarrow \text{u} \equiv \text{rewrite-step name pats rhs t} = \text{Some u}

lemma \text{rewrite-step1}:
assumes \text{match (name \$ pats) t} = \text{Some env subst \text{rhs env} u}
shows \text{name, pats, rhs} \vdash \text{t} \rightarrow \text{u}
using \text{assms unfolding rewrite-step-def by simp}

inductive \text{rewrite :: irule-set} \Rightarrow \text{pterm} \Rightarrow \text{pterm} \Rightarrow \text{bool} (-/ \vdash - / - \vdash - [50,0,50] 50) \text{ for \text{irs}}
step: [\langle \text{name}, \text{rs} \rangle \in \equiv \text{irs}; (\text{pats}, \text{rhs}) \in \equiv \text{rs}; \text{name}, \text{pats}, \text{rhs} \vdash \text{t} \rightarrow \text{t}' \] \Rightarrow \text{irs} \vdash \text{t} \rightarrow \text{t}' \\
\text{beta}: [\langle \text{c} \rangle \in \equiv \text{cs}; \text{c} \vdash \text{t} \rightarrow \text{t}' \] \Rightarrow \text{irs} \vdash \text{Pabs cs \$p t} \rightarrow \text{t}' \\
\text{fun}: \text{irs} \vdash \text{t} \rightarrow \text{t}' \Rightarrow \text{irs} \vdash \text{t} \$p \text{ a} \rightarrow \text{t'} \$p \text{ u} \]
arg: $\text{irs} \vdash i \rightarrow u' \Rightarrow \text{irs} \vdash t \frac{$p}{u} \rightarrow t \frac{$p}{u'}$

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

**abbreviation** irewrite-rt :: irule-set ⇒ pterm ⇒ pterm ⇒ bool (-/ ⊢ -/ - →* -)

**lemma** (in irules) irewrite-closed:
assumes $\text{rs} \vdash t \rightarrow u \text{ closed } t$
shows closed $u$
using assms proof induction
case (step name $\text{rs} \ pats \ rhs \ t \ t'$)
then obtain env where match (name $$pats) t = \text{Some env } t' = \text{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
applies (subst frees-list-comb)
apply simp
applies (subst closed-except-def [symmetric])
using inner step by blast

next
case (beta $c \ cs \ t \ t'$)
then obtain pat rhs where $c = (pat, rhs)$
by (cases c) auto
with beta obtain env where match pat $t = \text{Some env } t' = \text{subst } rhs \ env$
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 $t' = \text{subst } rhs \ env$
apply (subst closed-except-def)
apply (subst subst-frees)
apply fact
apply (subst match-dom)
apply fact
applies simp
apply (subst closed-except-def[symmetric])
using inner beta \( \langle c = \cdot \rangle \) by (auto simp: closed-except-simps)
qed (auto simp: closed-except-def)

corollary (in irules) irewrite-rt-closed:
  assumes \( rs \vdash_i t \rightarrow^* u \) closed \( t \)
  shows closed \( u \)
using assms by induction (auto intro: irewrite-closed)

Correctness of translation

abbreviation irelated :: nterm \( \Rightarrow \) pterm \( \Rightarrow \) bool \( (- \approx_i \cdot [0,50] 50) \) where
\( n \approx_i p \equiv \text{nterm-to-pterm} n = p \)

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: \( t \approx_i u \implies \text{frees} t = \text{frees} u \)
by auto

lemma irelated-no-abs:
  assumes \( t \approx_i u \)
  shows \( \text{no-abs} t \leftrightarrow \text{no-abs} u \)
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 \( \text{subst} t \ nenv \approx_i \text{subst} u \ penv \)
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-rewrite-step:
  assumes name, pats, nterm-to-pterms 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-pterms 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' = \$$
  by (auto intro: irelated-subst)
  qed
qed

theorem (in nrules) compile-correct:
  assumes compile (consts-of rs) \vdash_i u \rightarrow u' \; t \approx_i u closed t
  obtains t' where rs \vdash_n t \rightarrow 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' t)
  then obtain crs where irs = translate-crules crs (name, crs) \in\| consts-of rs
  unfolding compile-def by force
  moreover with step obtain rhs' where rhs = nterm-to-pterms rhs' (pats, rhs')
  \in\| crs
  unfolding translate-crules-def by force
  ultimately obtain rule where split-rule rule = (name, (pats, rhs')) rule \in\| rs
  unfolding consts-of-def by blast
  hence nrule rule
  using all-rules by blast

obtain t' where unsplit-rule (name, pats, rhs') \vdash t \rightarrow t' \; t' \approx_i u'
  using (name, pats, rhs) \vdash_i u \rightarrow u' \; t \approx_i u unfolding (rhs = nterm-to-pterms rhs')
  by (elim related-rewrite-step)
  hence rule \vdash t \rightarrow t'
  using (nrule rule) (split-rule rule = (name, (pats, rhs')))
  by (metis unsplit-split)

show ?case

113
proof (rule step.prems)
  show \( rs \vdash n \, t \rightarrow t' \)
  apply (rule nrewrite.step)
    apply fact
    apply fact
  done
next
  show \( t' \approx_i u' \)
    by fact
qed

next
case (beta c cs u u')
then obtain pat rhs where \( c = (pat, \, rhs) \, (pat, \, rhs) |\in| cs \)
  by (cases c) auto
obtain v w where \( t = v \, \$_n \, w \, v \approx_i \, Pabs \, cs \, w \approx_i \, u \)
  using \( \langle t \approx_i \, Pabs \, cs \rangle \) by (auto elim: nterm-to-pterm.elims)
obtain x nrhs irhs where \( v = (\Lambda_n \, x. \, nrhs) \, cs = \{[(\text{Free} \, x, \, irhs)]\} \, nrhs \approx_i \, irhs \)
  using \( \langle v \approx_i \, Pabs \, cs \rangle \) by (auto elim: nterm-to-pterm.elims)
hence \( t = (\Lambda_n \, x. \, nrhs) \, \$_n \, \Lambda_n \, x. \, nrhs \approx_i \, \Lambda_p \, x. \, irhs \)
  unfolding \( \langle t = v \, \$_n \, w \rangle \) using \( \langle v \approx_i \, Pabs \, cs \rangle \) by auto
have pat = Free x rhs = irhs
  using \( cs = \{[(\text{Free} \, x, \, irhs)]\} \) by auto
hence \( t \vdash u \rightarrow u' \)
  using beta \( c = - \) by simp
hence \( u' = \text{subst} \, irhs \, (\text{fmap-of-list} \, [(x, \, u)]) \)
  by simp

show ?case
proof (rule beta.prems)
  show \( rs \vdash_n \, t \rightarrow \text{subst} \, nrhs \, (\text{fmap-of-list} \, [(x, \, w)]) \)
    unfolding \( \langle t = (\Lambda_n \, x. \, nrhs) \, \$_n \, w \rangle \)
    by (rule nrewrite.beta)
next
  show \( \text{subst} \, nrhs \, (\text{fmap-of-list} \, [(x, \, w)]) \approx_i \, u' \)
    unfolding \( \langle u' = \text{subst} \, irhs \rightarrow \rangle \)
    apply (rule irelated-subst)
    apply simp
    apply rule
    apply rule
    apply fact
    done
qed

next
case (fun v v' u)
obtain w x where \( t = w \, \$_n \, x \, w \approx_i \, v \, x \approx_i \, u \)
  using \( \langle t \approx_i \, v \, \$_p \, w \rangle \) by (auto elim: nterm-to-pterm.elims)
with fun obtain \( w' \) where \( rs \vdash w \rightarrow w' \approx_i v' \)

unfolding closed-except-def by auto

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

next

case (arg \( u \) \( u' \))

obtain \( w \) \( x \) where \( t = w \approx_i v \approx_i u \)
  using \( t \approx_i v \) \( \approx_i u \) by (auto elim: nterm-to-pterm.elims)

with arg obtain \( x' \) where \( rs \vdash n \rightarrow x' \approx_i u' \)
  unfolding closed-except-def by auto

show \(?case\)
proof (rule arg.prems)
  show \( rs \vdash n \rightarrow w \approx_i v \)
    unfolding \((t = w \approx_i x)\)
    by (rule nrewrite.arg) fact
next
  show \( w \approx_i v \approx_i u \)
    by auto fact+
qed

qed

corollary (in nrules) compile-correct-rt:

assumes compile ( consts-of rs ) \( \vdash_1 u \rightarrow^* t \approx_i u \) closed \( t \)

obtains \( t' \) where \( rs \vdash n \rightarrow^* t' \approx_i u' \)

using assms proof ( induction arbitrary: thesis \( t \) )

  case (step \( u' \) \( u'' \) )

obtain \( t' \) where \( rs \vdash n \rightarrow^* t' \approx_i u' \)
  using step by blast

obtain \( t'' \) where \( rs \vdash n \rightarrow^* t'' \approx_i u'' \)

proof (rule compile-correct)
  show compile ( consts-of rs ) \( \vdash_1 u' \rightarrow^* u'' \approx_i u' \)
    by fact+
next
  show closed \( t' \)
    using \( rs \vdash n \rightarrow^* t' \) (closed \( t \) )
    by (rule nrewrite-rt-closed)
qed blast

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

Completeness of translation

lemma (in nrules) compile-complete:
  assumes \( \text{rs} \vdash n \rightarrow t' \) closed \( t \)
  shows \( \text{compile} \ (\text{consts-of } \text{rs}) \vdash n \rightarrow t' \)
  using assms proof induction
  case (\( \text{step} \ r \ t \ t' \))
  then obtain \( \text{pat} \ \text{rhs}' \) where \( r = (\text{pat}, \text{rhs}') \)
    by force
  then have \( \langle \text{pat}, \text{rhs}' \rangle \vdash t \rightarrow t' \)
    using \( \text{step} \) by blast+
  then have \( \text{nrule} \ (\text{pat}, \text{rhs}') \)
    using all-rules by blast
  then obtain \( n \text{rule} \ (\text{pat}, \text{rhs}') \)
    unfolding split-rule-def \( \langle r = - \rangle \)
    by (auto simp: split-beta)
  then obtain \( \text{crs} \) where \( (\text{name}, \text{crs}) |\in| \text{consts-of } \text{rs} \ (\text{pats}, \text{rhs}') |\in| \text{crs} \)
    using \( \text{step} (r \ t \ t') \) \( \langle \text{name}, (\text{pats}, \text{rhs}') \rangle = \text{split-rule } r \) \( \text{pat} = \text{name} \)
  unfolding \( \langle r = - \rangle \)
  apply atomize-elim
  by (auto simp: split-beta)
  then obtain \( \text{irs} \)
    unfolding compile-def
    using \( (\text{name}, -) |\in| - \)
    by (metis compile-def fimageI id_def map_prod_simp)
  then obtain \( \text{rhs} \)
    unfolding translate-crules-def
    by (metis compile-def fimageI id_def map_prod_simp)
  then obtain \( \text{env}' \)
    unfolding match pat t = Some \( \text{env}' \) \( t' = \text{sub} \text{st} \text{rhs}' \text{env}' \)
    unfolding \( \langle r = - \rangle \) using rewrite-step.simps
    by force
  then obtain \( \text{env} \)
    unfolding match pat (nterm-to-pterms \( \text{t} \)) = Some \( \text{env} \) \( \text{irelated} \).P-env \( \text{env}' \)
    unfolding \( \langle t' = - \rangle \)
    apply --
apply (rule sym)
apply (rule irelated-subst)
unfolding ⟨rhs = ⟩
by auto

have name, pats, rhs ⊢ 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 (λenv. (last pats, subst rhs env)) (matches (butlast pats) args))

category irules

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

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

lemma prelated-refl[intro!]: t ≈ p t
proof (induction t)
case Pabs
  thus ?case
    by (auto simp: snds.simps fmember.rep-eq 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-ptermdf app-ptermdf intro: prelated.intros elim: prelated.cases)

lemma prelated-pvars:
  assumes t ≈ p u
  shows frees t = frees u
using assms proof (induction rule: prelated.induct)
case (pat cs1 cs2)
  show ?case
    apply simp
    apply (rule arg-cong[where f = ffUnion])
  apply (rule rel-fset-image-eq)
  apply fact
  apply auto
  done
next
case (defer name rsi args cs)
\{ 
  fix \textit{pat} \textit{t} 
  assume \((\textit{pat}, \textit{t}) \mid \in \mid \textit{cs})\) 
  with defer obtain \textit{t}' 
    where \((\textit{pat}, \textit{t}') \mid \in \mid \textit{irules-deferred-matches args rsi \textit{frees} \textit{t} = \textit{frees} \textit{t}')\) 
    by (auto elim: rel-fsetE2) 
  then obtain \textit{pats} \textit{rhs} \textit{env} 
    where \(\textit{pat} = \textit{last} \textit{pats} (\textit{pats}, \textit{rhs}) \mid \in \mid \textit{rsi}\) 
    and \(\textit{matchs} (\textit{butlast} \textit{pats}) \textit{args} = \textit{Some} \textit{env} \textit{t}' = \textit{subst} \textit{rhs} \textit{env}\) 
    by auto 

  have \textit{closed-except} \textit{rhs} (\textit{fress} \textit{pats}) \textit{linears} \textit{pats} 
    using \((\textit{pats}, \textit{rhs}) \mid \in \mid \textit{rsi}\) \((\textit{name}, \textit{rsi}) \mid \in \mid \textit{rs}\) \textit{inner} by blast+ 

  have \textit{arity-compatibles} \textit{rsi} 
    using defer inner by (blast dest: fpairwiseD) 
  have \textit{length} \textit{pats} > 0 
    by (subst \textit{arity-compatible-length}) fact+ 
  hence \textit{pats} = \textit{butlast} \textit{pats} @ \textit{[last} \textit{pats]}\) 
    by simp 

  note \((\textit{frees} \textit{t} = \textit{frees} \textit{t}')\) 
  also have \textit{frees} \textit{t}' = \textit{frees} \textit{rhs} - \textit{fmdom} \textit{env}\) 
    unfolding \(\textit{t}' = \textit{-}\) 
    apply (rule subst-frees) 
    apply (rule closed.matchs) 
    apply fact+ 
    done 
  also have \ldots = \textit{frees} \textit{rhs} - \textit{fress} (\textit{butlast} \textit{pats}) 
    using (\textit{matchs - -} = \textit{-}) by (metis \textit{matchs-dom}) 
  also have \ldots \subseteq \textit{fress} \textit{pats} - \textit{fress} (\textit{butlast} \textit{pats}) 
    using \textit{closed-except - -}\) 
    by (auto simp: closed-except-def) 
  also have \ldots = \textit{frees} (\textit{last} \textit{pats}) \mid -\mid \textit{fress} (\textit{butlast} \textit{pats}) 
    by (subst \(\textit{pats} = \textit{-}\)) (simp add: funion-fminus) 
  also have \ldots = \textit{frees} (\textit{last} \textit{pats}) 
    proof (rule funminus-triv) 
      have \textit{fdisjnt} (\textit{fress} (\textit{butlast} \textit{pats})) (\textit{fress} \textit{[last} \textit{pats]}\) 
        using \(\textit{linears} \textit{pats}\) \(?\) \textit{pats} = \textit{-}\) by (metis \textit{linears-appendD}) 
      thus \textit{frees} (\textit{last} \textit{pats}) \cap \textit{fress} (\textit{butlast} \textit{pats}) = \{\textit{-}\} 
        by (fastforce simp: fdisjnt-alt-def) 
    qed 
  also have \ldots = \textit{fress} \textit{pat} unfolding \(\textit{pat} = \textit{-}\) .. 
  finally have \textit{frees} \textit{t} \subseteq \textit{fress} \textit{pat}\) . 
\} 

hence \textit{closed} (\textit{Pabs} \textit{cs}) 
unfolding \textit{closed-except-simps} 
by (auto simp: closed-except-def)
moreover have closed (name $$\text{args}$$)

unfolding closed-list-comb by fact

ultimately show ?case

unfolding closed-except-def by simp

qed auto

corollary prelated-closed: \( t \approx_p u \Rightarrow \text{closed } t \leftrightarrow \text{closed } u \)

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 \( \text{env} \) \( \text{env} \)

by (auto intro: fmap.rel-refl)

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

Instead, we use 
\[
\left[ \text{match } ?x \ ?u = \text{Some } ?\text{env}; ?t \approx_p ?u; \land \text{env}', \left[ \text{match } ?x \ ?t = \text{Some env'}; \right. \text{prelated.P-env env'} \ ?\text{env} \right] \right] \Rightarrow ?\text{thesis}
\]

lemma prelated-subst:

assumes \( t_1 \approx_p t_2 \) prelated.P-env \( \text{env}_1 \) \( \text{env}_2 \)

shows \( \text{subst } t_1 \approx_p \text{subst } t_2 \) \( \text{env}_1 \) \( \text{env}_2 \)

using assms proof (induction arbitrary: \( \text{env}_1 \) \( \text{env}_2 \) 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 \( cs_1 \) \( cs_2 \))

let \( ?\text{drop} = \lambda \text{env}. \lambda(\text{pat::term}). \text{fmdrop-fset (frees pat)} \) \( \text{env} \)
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 $$\text{args} \approx_p \text{Pabs} \text{cs}
    apply (rule prelated.defer)
    apply fact+
    apply (rule fset.rel-mono-strong)
    apply fact
    apply force
    apply fact
done
moreover have closed (name $$\text{args})
  unfolding closed-list-comb by fact
ultimately have closed (Pabs cs)
  by (metis prelated-closed)

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

have name $$\text{args} \approx_p \text{Pabs} (?f \text{env2} |\cdot| \text{cs})
  proof (rule prelated.defer)
    show (name, rsi) \in| rs 0 < \text{arity rsi list-all closed args}
      using defer by auto
  next
    \{ 
      fix pat1 rhs1
      fix pat2 rhs2
      assume (pat2, rhs2) \in| cs
      assume pat1 = pat2 rhs1 \approx_p rhs2
      have rhs1 \approx_p subst rhs2 (fmdrop-fset (frees pat2) env2)
        by (subst subst-closed-pabs) fact+
    \}
  hence rel-fset (rel-prod (=) prelated) (id |\cdot| \text{irules-deferred-matches args rsi})
    (?f env2 |\cdot| cs)
    by (force intro!: rel-fset-image[OF rel-fset - -])
  thus rel-fset (rel-prod (=) prelated) (irules-deferred-matches args rsi) (?f env2 |\cdot| cs)
    by simp
qed

moreover have map (\lambda. \text{subst} t env1) \text{args} = \text{args}
  apply (rule map-idI)
  apply (rule subst-closed-id)
  using defer by (simp add: list-all-iff)

121
ultimately show \( ?\text{case} \)
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 \( \vdash \) \( t \rightarrow t' \) \( t' \approx_p u' \)
proof –
let \( ?\text{lhs} = \text{name} \$\$ \text{pats} \)
from assms obtain env where match \( ?\text{lhs} u = \text{Some} \text{ env} u' = \text{subst} \text{ rhs env} 
unfolding irewrite-step-def by blast
then obtain env' where match \( ?\text{lhs} t = \text{Some} \text{ env'} \) \text{prelated}.P-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 \( \vdash \) \( t \rightarrow \text{subst} \text{ rhs env'} 
unfolding irewrite-step-def using \( \langle \text{match} \ ?\text{lhs} t = \text{Some} \text{ env'} \rangle \)
by simp

next
show subst rhs env' \( \approx_p \) u'

unfolding \( \langle u' = \text{subst} \text{ rhs env} \rangle \)
by fact
qed

lemma prelated-beta: — same problem as prelated.related-match
assumes \( (\text{pat}, \text{rhs}_2) \vdash t_2 \rightarrow u_2 \) \( \text{rhs}_1 \approx_p \text{rhs}_2 \) \( t_1 \approx_p t_2 \)
obtains \( u_1 \) where \( (\text{pat}, \text{rhs}_1) \vdash t_1 \rightarrow u_1 \) \( u_1 \approx_p u_2 \)
proof –
from assms obtain env_2 where match \( \text{pat} t_2 = \text{Some} \text{ env}_2 u_2 = \text{subst} \text{ rhs}_2 \text{ env}_2 

by auto

with assms obtain env_1 where match \( \text{pat} t_1 = \text{Some} \text{ env}_1 \) \text{prelated}.P-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 \( (\text{pat}, \text{rhs}_1) \vdash t_1 \rightarrow \text{subst} \text{ rhs}_1 \text{ env}_1 
using \( \langle \text{match} \ \text{pat} t_1 = - \rangle \) by simp

next
show subst rhs_1 env_1 \( \approx_p \) \( u_2 \)

unfolding \( \langle u_2 = - \rangle \) by fact
theorem transform-correct:
  assumes transform-irule-set rs \vdash_i u \rightarrow u' t \approx_p u closed t
  obtains t' where rs \vdash_i t \rightarrow* t' — zero or one step and t' \approx_p u'
  using assms(1-3) proof (induction arbitrary: i thesis rule: irewrite.induct)
  case (\beta c cs2 u2 x2')
  obtain v u1 where t = \beta v Pabs cs2 v \approx P u1 closed u1
  using \langle t \approx P Pabs cs2 \rangle by cases
  with \beta have closed u1
  by (simp add: closed-except-def)
  obtain pat rhs2 where c = (pat, rhs2) by (cases c) auto
  from (v \approx P Pabs cs2) show \?case
  proof (cases rule: prelated-absE)
    case (pat cs1)
    with \beta obtain rhs1 where (pat, rhs1) \langle c \rangle cs1 rhs1 \approx_p rhs2
    by (auto elim: rel-fsetE2)
    with \beta obtain x1' where (pat, rhs1) \vdash_i x1' \approx_p x2'
    using (u1 \approx_p u2) assms \langle c \rangle
    by (auto elim: prelated-beta simp del: rewrite-step.simps)
  show \?thesis
  proof (rule beta.prems)
    show rs \vdash_i t \rightarrow* x1'
      unfolding (t = \_ \_ \_ \langle \_ \_ \_ \_ \rangle)
      by (intro r-into-rtranclp irewrite.beta fact+)
  next
    show x1' \approx_p x2'
      by fact
  qed
  next
  case (defer name rsi args)
  with \beta obtain rhs1' where (pat, rhs1') \langle c \rangle irules-deferred-matches
  args rsi rhs1' \approx_p rhs2
  by (auto elim: rel-fsetE2)
  then obtain envv rhs1 pats
  where matches (butlast pats) args = Some envv pat = last pats rhs1' = subst
  rhs1 envv
  and (pats, rhs1) \langle c \rangle rsi
  by auto
  hence linears pats
  using (name, rsi) \langle c \rangle rsi inner unfolding irules-def by blast
  have arity-compatibles rsi
  using defer inner by (blast dest: fpairwiseD)
  have length pats > 0

qed
qed
by (subst arity-compatible-length) fact+
hence pats = butlast pats @ [pat] unfolding \( \text{pat} = \cdot \) by simp

from beta \( c = \cdot \) obtain \( \text{env}_b \) where match \( \text{pat} \) \( u_2 \) = Some \( \text{env}_b \) \( x_2' \) = subst \( \text{rhs}_2 \) \( \text{env}_b \)
by fastforce
with \( u_1 \approx_p u_2 \) obtain \( \text{env}_b' \) where match \( \text{pat} \) \( u_1 \) = Some \( \text{env}_b' \) prelated \( P\)-env \( \text{env}_b' \) \( \text{env}_b \)
by (metis prelated.prelated-match)

have closed-env \( \text{env}_a \)
by (rule closed.matches) fact+
have closed-env \( \text{env}_b' \)
apply (rule closed.matches[where \( \text{pats} = [\text{pat}] \) and \( \text{ts} = [u_1] \)])
apply simp
apply fact
apply simp
apply fact
done

have \( \text{fmdom} \) \( \text{env}_a \) = frees \( \) (butlast \( \text{pats} \))
by (rule matches-dom) fact
moreover have \( \text{fmdom} \) \( \text{env}_b' \) = frees \( \text{pat} \)
by (rule match-dom) fact
moreover have \( \text{fdisjnt} \) (frees \( \) (butlast \( \text{pats} \))) (frees \( \text{pat} \))
using pats = \( \cdot \) \( \) (linesars \( \text{pats} \))
by (metis frees-single linears-appendD(3))
ultimately have \( \text{fdisjnt} \) (fmdom \( \text{env}_a \)) (fmdom \( \text{env}_b' \))
by simp

show \( \text{thesis} \)
proof (rule beta.prems)
have \( \text{rs} \vdash_i \text{name} $$ \) \( \text{args} \) \( \$ \) \( \text{u}_1 \) \( \rightarrow \) subst \( \text{rhs}_1' \) \( \text{env}_b' \)
proof (rule irewrite.step)
show \( \text{name} \) \( \text{rsi} \) \( |\in| \) \( \text{rs} \) \( \) \( \text{pats} \) \( \text{rhs}_1 \) \( |\in| \) \( \text{rsi} \)
by fact+
next
show \( \text{name} \), \( \text{pats} \), \( \text{rhs}_1 \) \( \vdash_i \) \text{name} $$ \) \( \text{args} \) \( \$ \) \( \text{u}_1 \) \( \rightarrow \) subst \( \text{rhs}_1' \) \( \text{env}_b' \)
apply (rule irewrite-stepI)
apply (fold app-ptermd-def)
apply (subst list-comb-snoc)
apply (subst matches-match-list-comb)
apply (subst \( \cdot \)pats = \( \cdot \))
apply (rule matches-appI)
apply fact
apply simp
apply fact
unfolding \( \text{rhs}_1' = \cdot \)
apply (rule subst-indep')
apply fact+
done
qed
thus rs ⊢ it →∗ subst rhs₁' envᵇ'
unfolding ⟨t = _⟩ ⟨v = _⟩
by (rule r-into-rtranclp)
next
show subst rhs₁' envᵇ' ⪅ x₂'
unfolding ⟨x₂' = _⟩
by (rule prelated-subst) fact+
qed
next
case (step name rs₂ pats rhs u u')
then obtain rs₁ where rs₂ = transform-irules rs₁ (name, rs₁) |∈| rs
unfolding transform-irule-set-def by force
hence arity-compatibles rs₁
using inner by (blast dest: fpairwiseD)

show ?case
proof (cases arity rs₁ = 0)
  case True
  hence rs₂ = rs₁
  unfolding ⟨rs₂ = _⟩ transform-irules-def by simp
  with step have (pats, rhs) |∈| rs₁
  by simp
  from step obtain t' where name, pats, rhs ⊢ i t → t' t' ⪅_p u'
  using assms
  by (auto elim: prelated-step)

show ?thesis
proof (rule step.prems)
  show rs ⊢₁ t →∗ t'
  by (intro conjI exI r-into-rtranclp irewrite.step) fact+
qed fact
next
let ?f = λ(pats, rhs). (butlast pats, last pats, rhs)
let ?grp = fgroup-by ?f rs₁

  case False
  hence rs₂ = map-prod id Pabs |'[ ?grp
  unfolding ⟨rs₂ = _⟩ transform-irules-def by simp
  with step obtain cs where rhs = Pabs cs (pats, cs) |∈| ?grp
  by force
  from step obtain env₂ where match (name $₈ $pats) u = Some env₂ u' = subst rhs env₂
  unfolding irewrite-step-def by auto

125
then obtain \( \text{args}_2 \) where \( u = \text{name } \$ \text{args}_2 \) \( \text{matches pats args}_2 = \text{Some env}_2 \)
\[\text{by (auto elim: match-list-combE)}\]
\[\text{with step obtain } \text{args}_1 \text{ where } t = \text{name } \$ \text{args}_1 \text{ list-all2 related args}_1 \text{ args}_2 \]
\[\text{by (auto elim: related.list-combE)}\]
then obtain \( \text{env}_1 \) where \( \text{matches pats args}_1 = \text{Some env}_1 \text{ prelated.P-env env}_2 \)
\[\text{using } \langle \text{matches pats args}_2 = \text{\_} \rangle \text{ by (metis prelated.related-matches)}\]
\[\text{hence } \text{fndom env}_1 = \text{fress pats} \]
\[\text{by (auto simp: matches-dom)}\]
obtain \( cs' \) where \( u' = \text{Pabs cs'} \)
\[\text{unfolding } \langle u' = \text{\_} \rangle \langle \text{rhs} = \text{\_} \rangle \text{ by auto} \]
\[\text{hence } cs' = (\lambda (\text{pat, rhs}). (\text{pat, subst rhs (fmdrop-fset (frees pat) env}_2 ))) |' | \]
\[\text{cs using } \langle u' = \text{subst rhs env}_2 \rangle \text{ unfolding } \langle \text{rhs} = \text{\_} \rangle \]
\[\text{by simp} \]
show \( ?\text{thesis} \)
proof (rule step.prems)
\[\text{show } rs \vdash_1 t \rightsquigarrow^* t \]
\[\text{by (rule rtranclp.rtrancl-refl)}\]
next
\[\text{show } t \approx_p u' \]
\[\text{unfolding } \langle u' = \text{Pabs cs'} \rangle \langle t = \text{\_} \rangle \]
proof (intro prelated.defer rel-fsetI; safe?)
\[\text{show } (\text{name, rs}_1) \mid \in| rs \]
\[\text{by fact} \]
next
\[\text{show } 0 < \text{arity rs}_1 \]
\[\text{using False by simp} \]
next
\[\text{show list-all closed args}_1 \]
\[\text{using \( \langle \text{closed t} \rangle \text{ unfolding } \langle t = \text{\_} \rangle \text{ closed-list-comb} \)} \]
next
\[\text{fix } \text{pat rhs}' \]
assume \( (\text{pat, rhs}') \mid \in| \text{irules-deferred-matches args}_1 \text{ rs}_1 \)
then obtain \( \text{pats}' \text{ rhs env} \)
\[\text{where } (\text{pats}', \text{ rhs}) \mid \in| \text{rs}_1 \]
\[\text{and } \text{matches (butlast pats')} \text{ args}_1 = \text{Some env pat = last pats}' \text{ rhs}' = \text{subst rhs env} \]
\[\text{by auto} \]
with False have \( \text{pats}' \neq [] \)
\[\text{using \( \langle \text{arity-compatibles rs}_1 \rangle \)} \]
\[\text{by (metis list.size(3) arity-compatible-length)} \]
\[\text{hence butlast pats}' @ [\text{last pats}'] = \text{pats}' \]
by simp

from ⟨(pats, cs) |∈| ?grp⟩ obtain pats_e rhs_e
where ⟨(patse, rhs_e) |∈| rs1⟩ pat = 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) |∈| rs1⟩ ⟨(patse, rhs_e) |∈| rs1⟩ ⟨(name, rs1) |∈| rs⟩
inner
by (blast dest: fpairwiseD)

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

have butlast pats' = pats env = env1
apply (rule matchs-compatible-eq)
subgoal by fact
subgoal
apply (rule linears-buttlastI)
using ⟨(pats', rhs) |∈| rs1⟩ ⟨(name, rs1) |∈| rs⟩ inner by blast
subgoal
using ⟨(pats, -) |∈| rs'2⟩ ⟨(name, rs2) |∈| transform-irule-set rs⟩
using irules':inner by blast
apply fact+
subgoal
apply (rule matchs-compatible-eq)
apply fact
apply (rule linears-buttlastI)
using ⟨(pats', rhs) |∈| rs1⟩ ⟨(name, rs1) |∈| rs⟩ inner
apply blast
using ⟨(pats, -) |∈| rs'2⟩ ⟨(name, rs2) |∈| transform-irule-set rs⟩
using irules':inner apply blast
by fact+
done

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

have fn dom env2 = frees pats
using :match (- $S -) = 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 (frees (butlast pats')) (frees (last pats'))
apply (subst freess-single[symmetric])
apply (rule linears-appendD)

127
apply (subst (butlast pats′@[last pats′] = pats′))
using (pats′, rhs) |∈| rs₁₁ \((name, rs₁₁) |∈| rs)\ inner
by blast

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 (frees (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) |∈| rs₁₁ \((name, rs₁₁) |∈| -)\ inner by blast
ultimately have \( \text{fdisjnt} (\text{fdom env}_1) \) (frees pat)
unfolding \( \text{fdom env}_1 = - \)
by (auto dest: linears-appendD)

let \( \text{?rhs-subst} = \lambda \text{env}. \text{subst rhs} (\text{fmdrop-fset} \text{(frees pat)} \text{env}) \)

show \( \text{fBex} \ (\text{irules-deferred-matches args}_1 \text{rs}_1) \) (\( \lambda \text{e. rel-prod} = \) prelated\( e \) (pat, rhs'))
unfolding \( \langle \text{rhs'} = - \rangle \)
proof (rule fBexI, rule rel-prod.intros)
  show \( \text{?rhs-subst env}_1 \approx \text{?rhs-subst env}_2 \)
    using (prelated.P-env \text{env}_1 \text{env}_2) inner
    by (auto intro: prerelated-subst)
next
have matches (butlast pats') \text{args}_1 = \text{Some env}_1
  using (matches pats \text{args}_1 = -) (pats = -) by simp
moreover have subst rhs \text{env}_1 = \?rhs-subst env_1
  apply (rule arg-cong[where \( f = \text{subst rhs} \])
unfolding \( \text{fmfilter-alt-defs} \)
apply (rule \( \text{fmfilter-true}[\text{symmetric}] \))
using \( \text{fdisjnt} (\text{fdom env}_1) \)
by (auto simp: fdisjnt-alt-def intro: fmdomI)
ultimately show \( (\text{pat}, \text{?rhs-subst env}_1) \) \( |\in| \) \( \text{irules-deferred-matches} \)
\text{args}_1 \text{rs}_1
  using \( \langle \text{pats', rhs} \rangle |\in| \text{rs}_1; \langle \text{pat = last pats'} \rangle \)
  by auto
qed simp
qed

next

case (fun \( v \ v' \ u \))
obtain \( w \ x \) where \( t = w \$p \ x \approx_p v \ x \approx_p u \) closed \( w \)
  using \( \langle t \approx_p v \$p \ w \rangle \) \( \text{closed} t \)
    by cases (auto simp: closed-except-def)
with fun obtain \( w' \) where \( \text{rs} \vdash w \rightarrow^* w' \) \( w' \approx_p v' \)
  by blast

show \( \text{?case} \)
proof (rule fun.prems)
  show \( \text{rs} \vdash t \rightarrow^* w' \$p \ x \)
    unfolding \( \langle t = - \rangle \)
    by (intro irewrite.rt-comb[unfolded app-pterms-def] rtranclp.rtrancl_refl) fact
next
  show \( w' \$p \ x \approx_p v' \$p \ u \)
    by (rule prelated.app) fact+
qed
next

case (arg \( u \ u' \ a \))
obtain \( w x \) where \( t = w \$p \ x \ w \approx_p v \ x \approx_p u \) closed \( x \)
using \( t \approx_p v \) \( \langle \text{closed } t \rangle \) by cases (auto simp; closed-except-def)

with \( \text{arg obtain } x' \text{ where } rs \vdash_i x \rightarrow^* x' \approx_p u' \)

by blast

show \(?case\)

proof (rule \text{arg.prems})

show \( rs \vdash_i t \rightarrow^* w \) \( \) \( \approx_p x' \)

unfolding \( (t = w \approx_p x) \)

by (intro \text{irewrite.rt-comb[unfolded app-pterm-def] rtranclp.rtrancl-refl}) fact

next

show \( w \approx_p x' \approx_p v \approx_p u' \)

by (rule \text{prelated.app}) fact+

qed

qed

end

Completeness of transformation

lemma (in \text{irules}) transform-completeness:

assumes \( rs \vdash_i t \rightarrow t' \) closed \( t \)

shows \( \text{transform-irule-set } rs \vdash_i t \rightarrow^* t' \)

using \text{assms proof} induction

\text{case } \text{(step name } irs', \text{pats'} \text{, rhs'} \text{, } t \text{, } t') \text{ }

then obtain \text{irs where } irs = \text{transform-irules } irs' \text{ (name, } \text{irs) } \langle \in \rangle \text{ transform-irule-set } \text{rs}

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

by (metis fimageI id-apply map-prod-simp)

show \(?case\)

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

case True

hence \text{irs} = \text{irs'}

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

unfolding \text{transform-irules-def}

by simp

\text{with} \text{step have } \text{(pats', rhs') } \langle \in \rangle \text{ } \text{irs name, pats', rhs'} \vdash_i t \rightarrow t'

by blast+

\text{have transform-irule-set } rs \vdash_i t \rightarrow^* t'

apply (rule \text{r-into-rtranclp})

apply rule

by fact+

show \(?thesis\) by fact

next

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

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

\text{note } \text{closed-except-def } \langle \text{simp add} \rangle

\text{case False}

\text{then have } \text{irs} = \text{map-prod id Pabs } \langle \rangle \ ?grp
unfolding \( \text{irs} = \cdot \)
unfolding \( \text{transform-irules-def} \)
by simp
with \( \text{False have} \ \text{irs} = \text{transform-irules irs}' \)
unfolding \( \text{transform-irules-def} \)
by simp
obtain \( \text{pat} \ \text{pats} \) where \( \text{pat} = \text{last} \ \text{pats}' \ \text{pats} = \text{butlast} \ \text{pats}' \)
by blast
from \( \text{step False have} \ \text{length} \ \text{pats}' \neq 0 \)
using \( \text{arity-compatible-length inner} \)
by (smt \text{fBallE prod.simps(2)})
then have \( \text{pats}' = \text{pat} @ [\text{pats}] \)
unfolding \( \text{pat} = \cdot \ \text{pats} = \cdot \)
by simp
from \( \text{step have} \ \text{linears} \ \text{pats}' \)
using inner \text{fBallE}
by (metis (mono-tags, lifting) old.prod.case)
then have \( \text{fdisjnt (frees} \ \text{pats}) (\text{frees} \ \text{pat}) \)
unfolding \( \text{pats}' = \cdot \)
using \( \text{linears-appendD(3) frees-single} \)
by force
from \( \text{step obtain} \ \text{cs} \) where \( (\text{pats}, \ \text{cs}) \subseteq \ ?\text{grp} \)
unfolding \( \text{pats} = \cdot \)
by (metis (no-types, lifting) \text{fgroup-by-complete fst-conv prod.simps(2)})
with \( \text{step have} \ (\text{pat}, \ \text{rhs'}) \subseteq \ ?\text{cs} \)
unfolding \( \text{pat} = \cdot \ \text{pats} = \cdot \)
by (meson \text{fgroup-byD old.prod.case})
have \( (\text{pats}, \ \text{Pabs} \ \text{cs}) \subseteq \ ?\text{irs} \)
using \( \text{irs} = \text{map-prod id Pabs} |\cdot| ?\text{grp} \ (\text{pats}, \ \text{cs}) \subseteq \cdot \)
by (simp add: \text{app-term-def})
then obtain \( t_0 \ t_1 \ \text{env}_0 \ \text{env}_1 \) where \( t = t_0 \$_p \ t_1 \ \text{match} \ (\text{name} \$_p \ \text{pats'}) \ t_0 = \text{Some} \ \text{env}_0 \ \text{match} \ t_1 = \text{Some} \ \text{env}_1 \ \text{env}' = \text{env}_0 ++_f \ \text{env}_1 \)
using \( \text{match-appE-split[OF \text{match} (\text{name} \$_p \ \text{pats'}) = \cdot |\cdot| \text{unfolded} \ \text{name} \$_p \ \text{pats'} = \cdot], unfolded \ \text{app-term-def}] \)
by blast
with \( \text{step have} \ \text{closed} \ t_0 \ \text{closed} \ t_1 \)
by auto
then have \( \text{closed-env} \ \text{env}_0 \ \text{closed-env} \ \text{env}_1 \)
using \( \text{match-vars[OF \text{match} - t_0 = \cdot] \ \text{match-vars[OF \text{match} - t_1 = \cdot]} \)
unfolding \( \text{closed-except-def} \)
by auto

131
obtain $t_0'$ where subst $(Pabs\ cs)\ env_0 = t_0'$
  by blast
then obtain $cs'$ where $t_0' = Pabs\ cs'\ cs' = ((\lambda(pat,\ rhs).\ (pat,\ subst\ rhs
\ (fmdrop-fset\ (frees\ pat)\ env_0)))\ |'\ cs)$
  using subst-pterms.simps(3) by blast
obtain $rhs$ where subst $rhs'\ (fmdrop-fset\ (frees\ pat)\ env_0) = rhs$
  by blast
then have $(pat,\ rhs)\ |\in|\ cs'$
  unfolding $(cs' = -)$
  using $(\cdot\ |\in|\ cs)$
  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 fnadd-drop-left-dom[symmetric])
  using (match\ pat\ -\ =\ match-dom)
  by (metis)
have $fdisjnt\ (fndom\ env_0)\ (fndom\ env_1)$
  using match-dom
  using (match\ pat\ -\ =\ match\ (name\ $$\ pats)\ -\ =\ -$)
  using $fdisjnt\ -\ -$
  unfolding fdisjnt-alt-def
  by (metis matches-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 $fdisjnt\ (fndom\ -\ -)$
  unfolding fdisjnt-alt-def
  by auto
show $\theta$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-pterms-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

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 (\lambda (-, rhs). \sim shadows-consts rhs)
  assumes welldefined-rs: fBall rs (\lambda (-, rhs). welldefined rhs)

Rewriting
inductive prewrite :: prule fset \Rightarrow pterm \Rightarrow pterm \Rightarrow bool 
  (-/ \vdash_p/ - \Rightarrow/ - [50,0,50] 50) for rs
where
step: (name, rhs) |∈| rs \Rightarrow rs \vdash_p Pconst name \Rightarrow rhs |
beta: c |∈| cs \Rightarrow c \vdash t \Rightarrow t' \Rightarrow rs \vdash_p Pabs cs \vdash_p t \Rightarrow t' |
fun: rs \vdash_p t \Rightarrow t' \Rightarrow rs \vdash_p t \vdash_p u \Rightarrow t' \$p u |
arg: rs \vdash_p u \Rightarrow u' \Rightarrow rs \vdash_p t \vdash_p u \Rightarrow t \$p u'

global-interpretation prewrite: rewriting prewrite rs for rs
by standard (auto intro: prewrite.intros simp: app-ptermdf)+

abbreviation prewrite-rt :: prule fset \Rightarrow pterm \Rightarrow pterm \Rightarrow bool 
  (-/ \vdash_p/ - \Rightarrow*/ - [50,0,50] 50) where
prewrite-rt rs \equiv (prewrite rs)**

Translation from irule-set to prule fset
definition finished :: irule-set \Rightarrow bool where
finished rs = fBall rs (λ(-, irs). arity irs = 0)

**definition** translate-rhs :: irules ⇒ pterm where
translate-rhs = snd o 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-cons 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 (blast dest: fpairwiseD)
    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 blast
    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 blast
  have closed-except u {||}
    using u inner ⟨(name, irs) |∈| rs⟩
    by (metis (mono-tags, lifting) case-prod-conv fbspec freess-empty)
  thus closed rhs
    unfolding (rhs = u).
{
  assume shadows-cons rhs
  hence shadows-cons u

135
unfolding compile-def \( \langle \text{rhs} = w \rangle \) by simp

moreover have \( \neg \text{shadows-consts } u \)
  using inner \( \langle [], u \rangle \mid \in \mid \text{irs} \) \( \langle \text{name, irs} \rangle \mid \in \mid \text{rs} \) by blast

ultimately show False by blast

}\)

have welldefined u
  using fbspec[OF inner \( \langle \text{name, irs} \rangle \mid \in \mid \text{rs} \), simplified] \( \langle [], u \rangle \mid \in \mid \text{irs} \)
  by blast

thus welldefined rhs
  unfolding \( \text{rhs} = u \) compile-def
  by simp

qed

thus

\( \text{fBall (compile rs) prule} \)
  \( \text{fBall (compile rs) } (\lambda (-, \text{rhs}). \neg \text{pre-constants.shadows-consts } C\text{-info } (\text{fst } |\cdot| \text{compile rs}) \text{ rhs}) \)
  \( \text{fBall (compile rs) } (\lambda (-, \text{rhs}). \text{pre-constants.welldefined } C\text{-info } (\text{fst } |\cdot| \text{compile rs}) \text{ rhs}) \)
  unfolding compile-heads by auto

next

show distinct all-constructor
  by (fact distinct-ctr)

qed

theorem (in irules) compile-correct:
  assumes compile rs \( \vdash p t \rightarrow t' \) finished rs
  shows rs \( \vdash i t \rightarrow t' \)
  using assms(1) proof induction
  case (step name rhs)
  then obtain irs where rhs = translate-rhs irs \( \langle \text{name, irs} \rangle \mid \in \mid \text{rs} \)
      unfolding compile-def by force
  hence arity-compatibles irs
      using inner by (blast dest: fpairwiseD)

  have is-fmap irs \( \neq \{[]\} \) arity irs = 0
    using assms inner \( \langle \text{name, irs} \rangle \mid \in \mid \text{rs} \) unfolding finished-def by blast+
    then obtain u where irs = \( \{ ([]) , u \} \)
      using \( \langle \text{arity-compatibles irs} \rangle \)
      by (metis arity-zero-shape)

  show ?case
    unfolding \( \langle \text{rhs} = \cdot \rangle \)
    apply (rule irewrite.step)
    apply fact
    unfolding \( \langle \text{irs = \cdot} \rangle \) translate-rhs-def irewrite-step-def
    by (auto simp: const-term-def)

  qed (auto intro: irewrite.intros)
theorem (in irules) compile-complete:
  assumes rs ⊢ᵢ t → t' finished rs
  shows compile rs ⊢ₚ t → t'
using assms(1) proof induction
  case (step name irs params rhs t t')
  hence arity-compatibles irs
    using inner by (blast dest: fpairwiseD)
  have is-fmap irs irs ≠ {||} arity irs = 0
    using assms inner step unfolding finished-def by blast+
  then obtain u where irs = {|| (||, u) ||}
    using (arity-compatibles irs)
    by (metis arity-zero-shape)
  with step have name, ||, u ⊢ᵢ t → t'
    by simp
  hence t = Pconst name
    unfolding irewrite-step-def
    by (cases t) (auto split: if-splits simp: const-term-def)
  hence t' = u
    unfolding irewrite-step-def
    by (cases t) (auto split: if-splits simp: const-term-def)
  have (name, t') ∈| compile rs
    unfolding compile-def
proof
  show (name, t') = map-prod id translate-rhs (name, irs)
    using (irs = -) t' = u
    by (simp add: split-beta translate-rhs-def)
  qed fact
thus ?case
  unfolding (t = -)
  by (rule prewrite.step)
qed (auto intro: prewrite.intros)

export-code
  compile finished
  checking Scala

end

3.4 Sequential pattern matching

theory Rewriting-Sterm
imports Rewriting-Pterm
begin

  type-synonym srule = name × sterm

end

137
abbreviation closed-srules :: srule list ⇒ bool where
closed-srules ≡ list-all (closed ◦ snd)

primrec srule :: srule ⇒ bool where
srule (name, rhs) ⇒ wellformed rhs ∧ closed rhs ∧ is-abs rhs

lemma sruleI[intro!]: wellformed rhs ⇒ closed rhs ⇒ is-abs rhs ⇒ srule (name, rhs)
by simp

locale srules = constants C-info fst | fset-of-list rs for C-info and rs :: srule list
+
  assumes all-rules: list-all srule rs
  assumes distinct: distinct (map fst rs)
  assumes not-shadows: list-all (λ(-, rhs). ¬ shadows-consts rhs) rs
  assumes swelldefined-rs: list-all (λ(-, rhs). swelldefined rhs) rs
begin

lemma map: is-map (set rs)
using distinct by (rule distinct-is-map)

lemma clausesE:
  assumes (name, rhs) ∈ set rs
  obtains cs where rhs = Sabs cs
proof –
  from assms have is-abs rhs
  using all-rules unfolding list-all-iff by auto
  then obtain cs where rhs = Sabs cs
  by (cases rhs) (auto simp: is-abs-def term-cases-def)
  with that show thesis .
qed

end

Rewriting

inductive 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
done
show ?thesis
  unfolding ⟨r = -⟩ ⟨name = -⟩ ⟨rhs = -⟩
  by (rule srewrite-step.cons-match)
qed
qed auto

lemma (in srules) srewrite-stepI : (name, rhs) ∈ set rs ⇒ srewrite-step rs name rhs
  using map
  by (metis srewrite-stepI0)

hide-fact srewrite-stepI0

inductive srewrite :: srule list ⇒ sterm ⇒ sterm ⇒ bool (-/ ⊢ s/ - −→/ - [50,0,50] 50) for rs where
step: srewrite-step rs name rhs ⇒ rs ⊢ Sconst name −→ rhs |
beta: rs ⊢ t $ u −→ t' |
fun: rs ⊢ t $ u −→ t' |
arg: rs ⊢ t $ u −→ t $ u' |

code-pred srewrite .

abbreviation srewrite-rt :: srule list ⇒ sterm ⇒ sterm ⇒ bool (-/ ⊢ s/ - −→*/* - [50,0,50] 50) where
srewrite-rt rs ≡ (srewrite rs)**

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

code-pred (modes: i ⇒ i ⇒ o ⇒ bool) srewrite-step .
code-pred (modes: i ⇒ i ⇒ o ⇒ bool) srewrite .
Translation from *pterm* to *sterm*

In principle, any function of type \((\texttt{a} \times \texttt{b}) \text{ fset} \Rightarrow (\texttt{a} \times \texttt{b}) \text{ list}\) that orders by keys would do here. However, for simplicity’s sake, we choose a fixed one (\texttt{ordered-fmap}) here.

```haskell
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 \$\_ u) = pterm-to-sterm t \$\_ pterm-to-sterm u |
pterm-to-sterm (Pabs cs) = Sabs (ordered-fmap (map-prod id pterm-to-sterm |\texttt{\_} | cs))
```

**Lemma** *pterm-to-sterm*:

- **Assumes** no-abs \(t\)
- **Shows** \(pterm-to-sterm t = \text{convert-term} t\)

**Using** \texttt{assms} **Proof** induction

- **Case** (free name)
  - **Show** ?\text{case}
    - **Apply** simp
    - **Apply** (simp add: free-sterm-def free-pterm-def)
    - done

**Next**

- **Case** (const name)
  - **Show** ?\text{case}
    - **Apply** simp
    - **Apply** (simp add: const-sterm-def const-pterm-def)
    - done

**Next**

- **Case** (app \(t_1 t_2\))
  - then **Show** ?\text{case}
    - **Apply** simp
    - **Apply** (simp add: app-sterm-def app-pterm-def)
    - done

**Qed**

*sterm-to-pterms* 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** \texttt{assms} **Proof** (induction \(t\) rule: pterm-induct)

- **Case** (Pabs \(cs\))
  - **Show** ?\text{case}
    - **Apply** simp
    - unfolding map-prod-def id-apply
    - **Apply** (intro conjI)
    - **Subgoal**
      - **Apply** (subst list-all-iff-fset)
      - **Apply** (subst ordered-fmap-set-eq)
      - **Apply** (rule is-fmap-image)

140
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

qed auto

lemma pterm-to-sterm-sterm-to-pterms:
assumes wellformed t
shows sterm-to-pterms (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 (subgoal-tac cs \neq \{\} )
    unfolding ordered-map-def
    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]) +
      simp
      apply (subgoal-tac cs \neq \{\} )
      unfolding id-def
      apply (rule fset-map-snd-id)
      simp
      apply (rule Pabs)
using Pabs(2) by (auto simp: fmember.rep-eq snds.simps)
done
qed auto

corollary pterm-to-sterm-frees: wellformed t \implies \frees (pterm-to-sterm t) = \frees t
by (metis pterm-to-sterm-sterm-to-pterm sterms-to-pterm-frees)

corollary pterm-to-sterm-closed:
  \closed-except t S \implies wellformed t \implies \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 \implies \consts (pterm-to-sterm t) = \consts t
by (metis pterm-to-sterm-sterm-to-pterm sterms-to-pterm-consts)

corollary (in \constants) pterm-to-sterm-shadows:
  wellformed t \implies \shadows-consts t \leftarrow \shadows-consts (pterm-to-sterm t)
unfolding shadows-consts-def
by (metis pterm-to-sterm-sterm-to-pterm sterms-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 \srcule (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
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-stermconsts)
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 |ι| 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 (\( \lambda (-, rhs). \neg \text{shadows-consts} \) 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 \( P = \lambda (-, rhs). \neg \text{shadows-consts} \) 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 (\( \lambda (-, rhs). \neg \text{shadows-consts} \) 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 (\( \lambda p. \) p = sterm-to-pterms)
proof (standard, goal-cases)
case (5 name t) then show ?case by (cases t) (auto simp: const-sterm-def const-ptermdf split: option.splits)
next
case (6 u1 u2 t)
then show ?case by (cases t) (auto simp: app-sterm-def app-ptermdf split: option.splits)
qed (auto simp: const-sterm-def const-ptermdf app-sterm-def app-ptermdf)

lemma srelated-subst:
assumes srelated.
shows subst (sterm-to-pterm t) penv = stern-to-ptermd (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 ⇒ rs' ≠ []
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-ptermd rhs (name, rhs) |∈| rs rest = compile
(rs − {| (name, rhs) |})
proof –
from assms have ordered-fmap (map-prod id pterm-to-ptermd |'| rs) = (name, rhs') ≠ rest
unfolding compile-def
by simp

hence \((\text{name, rhs'}) \in \text{set (ordered-fmap (map-prod id pterm-to-sterm |' | rs))}\)
by simp

have \((\text{name, rhs'}) |\in| \text{map-prod id pterm-to-sterm |' | rs}\)
apply (rule ordered-fmap-sound)

subgoal
  unfolding map-prod-def id-apply
  apply (rule is-fmap-image)
  done

subgoal by fact

done

then obtain \(\text{rhs where rhs'} = \text{pterm-to-sterm rhs (name, rhs) |\in| rs}\)
by auto

have rest = compile \((rs - \{|(\text{name, rhs})\}|)\)
unfolding compile-def
apply (subst inj-on-fimage-set-diff[where C = rs])

subgoal
  apply (rule inj-onI)
  apply safe
  apply auto
  apply (subst (asm) fmember.rep-eq[symmetric])+
  using is-fmap rs; by (blast dest: is-fmapD)

subgoal by simp

subgoal using \((\text{name, rhs}) |\in| rs\) 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) |\in| 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 \((\text{name, sterm-to-pterms rhs) |\in| rs}\)
using asms 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 name1 rest rhs rhs1’)
  then obtain rhs1 where rhs1’ = pterm-to-sterm rhs1 (name1, rhs1) ∈| rs rest
    = compile (rs − {{name1, rhs1}})
    by (auto elim: compile-consE)

  let ?rs’ = rs − {{name1, rhs1}}
  have (name, sterm-to-pterms rhs) ∈| ?rs’
    proof (intro cons-nomatch)
      show rest = compile ?rs’
        by fact
      show is-fmap (rs |−| {{name1, rhs1}})
        using ⟨is-fmap rs⟩
        by (rule is-fmap-subset) auto
      show fBall ?rs’ prule
        using cons-nomatch by blast
      qed
    thus ?case
      by simp
  qed

lemma compile-correct0:
  assumes compile rs ⊢ s u −→ u’ prules C rs
  shows rs ⊢ p sterm-to-pterms u −→ sterm-to-pterms u’
using assms proof induction
  case (beta cs t t’)
  then obtain pat rhs env where (pat, rhs) ∈ set cs match pat t = Some env t’
    = subst rhs env
    by (auto elim: rewrite-firstE)

  then obtain env’ where match pat (sterm-to-pterms t) = Some env’ srelated.P-env env’ env
    by (metis option.distinct(1) option.inject option.rel-cases srelated.match-rel)
  hence subst (sterm-to-pterms rhs) env’ = sterm-to-pterms (subst rhs env)
    by (simp add: srelated-subst)

  let ?rhs’ = sterm-to-pterms rhs
have \((\mathit{pat}, \mathit{\?rhs}') \in \mathsf{fset-of-list} (\mathsf{map-prod} \mathsf{id} \mathsf{sterm-to-pterm}) \mathsf{cs})\)
using \((\mathit{pat}, \mathsf{rhs}) \in \mathsf{set} \mathsf{cs})
including \(\mathsf{fset} \cdot \mathsf{lifting}
by \mathsf{transfer}' \mathsf{force}

\[\text{note} \ \mathsf{fset-of-list-map}[\text{simp del}]\]
show \(\mathsf{?case}\)
apply \(\text{simp}\)
apply \(\text{rule prewrite.intros}\)
apply \(\text{fact}\)
unfolding \(\text{rewrite-step.simps}\)
apply \(\text{subst map-option-eq-Some}\)
apply \(\text{intro exI conjI}\)
apply \(\text{simp del}\)

Completeness of translation

\[\text{global-interpretation} \ \mathsf{srelated'}: \ \mathsf{term-struct-rel-strong} (\lambda p \ s. \mathsf{pterm-to-sterm} p = s)\]

proof \(\text{(standard, goal-cases)}\)
\[\text{case} \ (1 \ t \ \text{name}) \]
then show \(\mathsf{?case} \ \text{by} \ (\text{cases} t) \ (\text{auto simp: const-sterm-def const-pterm-def split: option.splits})\)
next 
\[\text{case} \ (3 \ t \ u_1 \ u_2) \]
then show \(\mathsf{?case} \ \text{by} \ (\text{cases} t) \ (\text{auto simp: app-sterm-def app-ptermd-def split: option.splits})\)
qed \(\text{(auto simp: const-sterm-def const-ptermd-def app-sterm-def app-ptermd-def)}\)

\[\text{corollary} \ \mathsf{srelated-env-unique}:\]
\[ \text{srelated'} \cdot P\text{-env} \ penv \ senv \implies \text{srelated'} \cdot P\text{-env} \ penv \ senv' \implies \text{senv} = \text{senv'} \]

apply (subst (asm) fmrel-iff)+
apply (subst (asm) option.rel-sel)+
apply (rule fmap-ext)
by (metis option.exhaust-sel)

lemma 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 (subst (asm) fmember.rep-eq)
      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 (=) (λp s. pterm-to-sterm p = s)) cs ?cs'
    unfolding list.rel-map
  by (auto intro: list.rel-refl)
obtain \texttt{senv0}
  where \texttt{find-match \ ?cs' \ ?t' = Some (senv0, pat, pterm-to-sterm rhs) srelated'.P-env penv senv0}
    using \texttt{srelated'.find-match-rel\{OF \ * \ refl, where \ t = t, unfolded \ assms\}}
  unfolding \texttt{option-rel-Some1 rel-prod-conv}
  by \texttt{auto}
with \texttt{assms have senv = senv0}
  by (metis \texttt{srelated-env-unique})
show \texttt{?thesis}
  unfolding \texttt{⟨senv = -⟩ by fact}
qed

lemma (in \texttt{prules}) compile-complete:
  assumes \texttt{rs \vdash_{p} t \rightarrow t' wellformed t}
  shows \texttt{compile rs \vdash_{s} pterm-to-sterm t \rightarrow pterm-to-sterm t'}
using \texttt{assms proof} induction
  case (\texttt{step name rhs})
    then show \texttt{?case}
      apply \texttt{simp}
      apply \texttt{rule}
      apply \texttt{(rule prules-as-srules. srewrite-stepI)}
    unfolding \texttt{compile-def}
    apply \texttt{(subst fset-of-list-elem[ symmetric])}
    apply \texttt{(subst ordered-fmap-set-eq)}
    apply \texttt{(insert fmap)}
    apply \texttt{(rule is-fmapI)}
    apply \texttt{(force dest: is-fmapD)}
    by \texttt{(simp add: rev-fimage-eqI)}
next
  case (\texttt{beta c cs t t'})
  from \texttt{beta obtain pat rhs penv where \ c = (pat, rhs) match pat t = Some penv}
  subst \texttt{rhs penv = t'}
    by \texttt{(metis (no-types, lifting) map-option-eq-Some rewrite-step.simps surj-pair)}
then obtain \texttt{senv where match pat (pterm-to-sterm t) = Some senv srelated'.P-env penv senv}
  by \texttt{(metis option-rel- Some1 srelated'.match-rel)}
have \texttt{wellformed rhs}
  using \texttt{beta \ (c = -) prules.all-rules prule.simps}
  by \texttt{force}
then have \texttt{subst (pterm-to-sterm rhs) senv = pterm-to-sterm t'}
  using \texttt{srelated-subst' \ (c = t' \ srelated'.P-env - -)}
  by \texttt{metis}
have \texttt{(pat, pterm-to-sterm rhs) |\in| map-prod id pterm-to-sterm \ |\ '|' \ cs}
  using \texttt{beta \ (c = -)}
  by \texttt{(metis fimage-eqI id-def map-prod-simp)}
have \texttt{is-fmap cs}
  using \texttt{beta}
  by \texttt{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
3.5.1 Big-step semantics evaluating to irreducible sterms

inductive (in constructors) seval :: srule list ⇒ (name, sterms) fmap ⇒ sterms ⇒ bool (\- \- / \- /- [50,0,50] 50)
for rs where
const: (name, rhs) ∈ set rs ⇒ rs, Γ \-s Sconst name \- rhs |
var: fmlookup Γ name = Some val ⇒ rs, Γ \-s Svar name \- val |
abs: rs, Γ \-s Sabs cs \- Sabs (map (\lambda (pat, t). (pat, subst t (fmdrop-fset (frees pat) Γ))) cs) |
combs:
rs, Γ \-s t \- Sabs cs \- rs, Γ \-s u \- u' ⇒\nfind-match cs u' = Some (env, \-, rhs) ⇒\nrs, Γ ++f env \-s rhs \- val ⇒\nrs, Γ \-s t $s u \- val |
constrs:
name |∈| C \-⇒\nlist-all2 (seval rs Γ) ts us \-⇒\nrs, Γ \-s name $s ts \- name $s us

lemma (in constructors) seval-closed:
asums rs, Γ \-s t \- u closed-srules rs closed-env Γ closed-except t (fmdom Γ)
sows 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 Γ 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

\textbf{case} \((\text{abs } \Gamma \ cs)\)

\textbf{show} \(\text{?case}\)

\hspace{1em} apply \((\text{subst subst-sterm.simps[symmetric]})\)

\hspace{1em} apply \((\text{subst closed-except-def})\)

\hspace{1em} apply \((\text{subst subst-freses})\)

\hspace{1em} apply \text{fact+}

\hspace{1em} apply \((\text{subst fminus-fsubset-conv})\)

\hspace{1em} apply \((\text{subst closed-except-def[symmetric]})\)

\hspace{1em} apply \((\text{subst funion-fempty-right})\)

\hspace{1em} apply fact

\hspace{1em} done

\textbf{next}

\textbf{case} \((\text{constr name } \Gamma \ ts \ us)\)

\textbf{have} \(\text{list-all closed us}\)

\hspace{1em} using \((\text{list-all2 - - -}) \ (\text{closed-except} \ (\text{list-comb} - - -))\)

\textbf{proof} \((\text{induction ts us rule: list.rel-induct})\)

\hspace{1em} \textbf{case} \((\text{Cons } v \ vs \ u \ us)\)

\hspace{1em} \textbf{thus} \(\text{?case}\)

\hspace{2em} using constr unfolding closed.list-comb

\hspace{2em} by \text{auto}

\textbf{qed simp}

\hspace{1em} \textbf{thus} \(\text{?case}\)

\hspace{2em} unfolding closed.list-comb

\hspace{2em} by \((\text{simp add: closed-except-def})\)

\textbf{qed auto}

\textbf{lemma} \((\text{in srules}) \ \text{seval-wellformed}:\)

\hspace{1em} \textbf{assumes} \(rs, \Gamma \Gamma \_ t \downarrow u \text{ wellformed } t \text{ wellformed-env } \Gamma\)

\hspace{1em} \textbf{shows} \(\text{wellformed } u\)

\textbf{using} \(\text{assms proof induction}\)

\hspace{1em} \textbf{case} \((\text{const name rhs } \Gamma)\)

\hspace{2em} \textbf{thus} \(\text{?case}\)

\hspace{3em} using \(\text{all-rules}\)

\hspace{3em} by \((\text{auto simp: list-all-iff})\)

\textbf{next}

\textbf{case} \((\text{comb } \Gamma \ t \ cs \ u \ u’ \text{ env } \text{pat } \text{rhs } \text{val})\)

\textbf{hence} \((\text{pat, rhs }) \in \text{ set cs } \text{match pat u’ = Some env}\)

\hspace{1em} by \((\text{auto simp: find-match-elem})\)

\textbf{show} \(\text{?case}\)

\hspace{1em} \textbf{proof} \((\text{rule comb})\)

\hspace{2em} \textbf{show} \(\text{wellformed rhs}\)

\hspace{3em} using \((\text{pat, rhs }) \in \text{ set cs } \text{comb}\)

\hspace{3em} by \((\text{auto simp: list-all-iff})\)

\textbf{next}

\hspace{1em} \textbf{have} \(\text{wellformed-env env}\)
apply (rule wellformed.match)
apply fact
apply (rule comb)
using comb apply simp
apply fact+
done
thus wellformed-env (Γ ++ f env)
using comb by auto
qed

next
case (abs Γ cs)
thus ?case
by (metis subst-sterm.simps subst-wellformed)

next
case (constr name Γ ts us)
have list-all wellformed us
  using ⟨list-all2 - - ⟩ (wellformed (list-comb - -))
proof (induction ts us rule: list.rel-induct)
case (Cons v vs u us)
  thus ?case
  using constr by (auto simp: wellformed.list-comb app-sterm-def)
qed simp
thus ?case
  by (simp add: wellformed.list-comb const-sterm-def)
qed auto

lemma (in constants) seval-shadows:
  assumes rs, Γ t 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

154
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 `⊢ `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)`

155
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 (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 ⊢ 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 ⊢ 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)
qued (auto intro: seval.intros)

lemma (in constructors) seval-agree-eq:
  assumes rs, Γ ⊢ t ⊥ u fmrestrict-fset S Γ = fmrestrict-fset S Γ' closed-except t
  S
  assumes S |
     fmdom Γ closed-srules rs closed-env Γ
  shows rs, Γ' ⊢ t ⊥ u
using assms proof (induction arbitrary: Γ' S)
case (var Γ name val)
hence name ∈| S
  by (simp add: closed-except-def)
hence fmlookup Γ name = fmlookup Γ' name
  using (fmrestrict-fset S Γ = _)
  unfolding fnfilter-all-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)
have \( \ast : \text{fmdrop-fset } S \) \( (\text{fmrestrict-fset } T \ m) \) \( = \text{fmrestrict-fset } (T \cup S) \) \( (\text{fmdrop-fset } S \ m) \) for \( S \ T \ m \)

unfolding \( \text{fmfilter-alt-defs fnfilter-comp} \)

by (rule \( \text{fmfilter-cong} \)) auto

\{
  fix \( \text{pat} \ t \)
  assume \((\text{pat}, t) \in \text{set } \text{cs}\)
  with \( \text{abs } \) have \( \text{closed-except } t \) \((S \cup \text{frees pat})\)
  by (auto simp: \( \text{Sterm.} \) \( \text{closed-except-simps} \) \( \text{list-all-iff} \))

  have
    subst \( t \) \((\text{fmdrop-fset } (\text{frees pat}) (\text{fmrestrict-fset } S \ \Gamma))\) \( = \) subst \( t \) \((\text{fmdrop-fset } (\text{frees pat}) \ \Gamma)\)
      apply \((\text{subst } \ast )\)
      apply (rule \( \text{subst-restrict-closed} \))
      apply \( \text{fact} \)
      done

  moreover have
    subst \( t \) \((\text{fmdrop-fset } (\text{frees pat}) (\text{fmrestrict-fset } S \ \Gamma'))\) \( = \) subst \( t \) \((\text{fmdrop-fset } (\text{frees pat}) \ \Gamma')\)
      apply \((\text{subst } \ast )\)
      apply (rule \( \text{subst-restrict-closed} \))
      apply \( \text{fact} \)
      done

  ultimately have
    subst \( t \) \((\text{fmdrop-fset } (\text{frees pat}) \ \Gamma)\) \( = \) subst \( t \) \((\text{fmdrop-fset } (\text{frees pat}) \ \Gamma')\)
      using \( \text{abs } \) by \( \text{metis} \)
  \}

hence \( \text{map } (\lambda(\text{pat}, t). (\text{pat}, \text{subst } t \ (\text{fmdrop-fset } (\text{frees pat}) \ \Gamma))) \) \( \text{cs} = \) \( \text{map } (\lambda(\text{pat}, t). (\text{pat}, \text{subst } t \ (\text{fmdrop-fset } (\text{frees pat}) \ \Gamma'))) \) \( \text{cs} \)

by \( \text{auto} \)

thus \( \ast \text{case} \)
  by (metis \( \text{seval.abs} \))

next

\begin{itemize}
  \item \text{case} \((\text{comb } \Gamma \ t \ cs \ u \ u' \ \text{env pat rhs val})\)
  \item have \( \text{fndonm env } = \text{frees pat} \)
    apply (rule \( \text{match-dom} \))
    apply (rule \( \text{find-match-elem} \))
    apply \( \text{fact} \)
    done
  \item show \( \ast \text{case} \)
    proof (rule \( \text{seval.comb} \))
\end{itemize}
show \( rs, \Gamma' \vdash_s t \downarrow \text{Sabs cs rs}, \Gamma' \vdash_s u \downarrow u' \)
using \( \text{comb by (auto simp: Sterm.closed-except-simps)} \)

next
show \( rs, \Gamma' ++_f \text{env} \vdash_s \text{rhs} \downarrow \text{val} \)
proof (rule \( \text{comb} \))
  have \( \text{fnrestrict-fset (S |∪| \text{fmdom env}) (\Gamma' ++_f \text{env}) = fnrestrict-fset (S |∪| \text{fmdom env}) (\Gamma' ++_f \text{env})} \)
  using \( \text{comb(8)} \)
  unfolding \( \text{fnfilter-alt-defs} \)
  including \( \text{fmap.lifting fset.lifting} \)
  by \( \text{transfer’ (auto simp: map-filter-def fun-eq-iff map-add-def split: option.splits if-splits)} \)
thus \( \text{fnrestrict-fset (S |∪| \text{frees pat}) (\Gamma' ++_f \text{env}) = fnrestrict-fset (S |∪| \text{frees pat}) (\Gamma' ++_f \text{env})} \)
  unfolding \( \text{⟨fmdom env = ∅⟩} \).

next
have \( \text{closed-except t S} \)
using \( \text{comb by (simp add: Sterm.closed-except-simps)} \)

have \( \text{closed (Sabs cs)} \)
apply (rule \( \text{seval-closed} \))
apply \( \text{fact} + \)
using \( \text{⟨closed-except t S⟩ (S |⊆| \text{fmdom Γ})} \)
unfolding \( \text{closed-except-def} \) apply simp
done

have \( \text{(pat, rhs) ∈ set cs} \)
using \( \text{⟨find-match - - = ∅⟩ by (rule find-match-elem)} \)
hence \( \text{closed-except rhs (frees pat)} \)
using \( \text{⟨closed (Sabs cs)⟩ by (auto dest: closed-except-sabs)} \)
thus \( \text{closed-except rhs (S |∪| \text{frees pat})} \)
unfolding \( \text{closed-except-def} \) by auto

next
show \( S |∪| \text{frees pat} |⊆| \text{fmdom (\Gamma' ++_f \text{env})} \)
apply simp
apply (intro conjI)
using \( \text{comb(10)} \) apply blast
unfolding \( \text{⟨fmdom env = ∅⟩ by blast} \)

next
have \( \text{closed-except u S} \)
using \( \text{comb by (auto simp: closed-except-def)} \)

show \( \text{closed-env (\Gamma' ++_f \text{env})} \)
apply \( \text{rule} \)
apply \( \text{fact} \)
apply (rule \( \text{closed.match[where t = u' and pat = pat]} \))
subgoal
by \( \text{(rule find-match-elem) fact} \)
subgoal
  apply (rule seval-closed)
  apply fact+
  using ⟨closed-except u S⟩ ⟨S |⊆| fmdom Γ⟩ unfolding closed-except-def
by blast
  done
qed fact

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, Γ ⊢ s t ↓ u closed-except t (fmdom Γ) closed-env Γ
  shows rs ⊢ subst t Γ −→∗ u
  using assms proof induction
  case (const name rhs Γ)
    have srewrite-step rs name rhs
      by (rule srewrite-stepI) fact
    thus ?case
      by (auto intro: srewrite.intros)
  next
  case (comb Γ t cs u u′ env pat rhs val)
    hence closed-except t (fmdom Γ) closed-except u (fmdom Γ)
      by (simp add: 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) ∈ set cs match pat u′ = Some env
      by (auto simp: find-match-elem)
    hence closed-except rhs (frees pat)
      using :closed (Sabs cs) by (auto dest: closed-except-sabs)
    hence frees rhs |⊆| frees pat

160
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 |⊆| fmdom env
by simp
hence subst rhs (Γ ++ f env) = subst rhs env
by (rule subst-add-shadowed-env)

have rs ⊢ subst t Γ $ s subst u Γ −→∗ Sabs cs $ s u'
also have rs ⊢ subst rhs (Γ ++ f env) −→∗ subst rhs env
  using comb ⟨match pat u' = Some env⟩ ⟨fmdom env = ⊥⟩ ⟨frees rhs |⊆| 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, fnempty ⊢ t ⊖ u closed t
  shows rs ⊢ t −→∗ u
proof
  have closed-except t (fndom fmempty)
    using assms by simp
  with assms have rs ⊢ subst t fmempty →∗ u
    by (fastforce intro: seval-correct0)
  thus ?thesis
    by simp
qed

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 (-, -/ ⊢ v -/ -/ [50,0,50]) for rs where
  const: (name, rhs) ∈ set rs ⇒ rs, Γ ⊢v Sconst name ↓ rhs |
  var: fnlookup Γ name = Some val ⇒ rs, Γ ⊢v Svar name ↓ val |
  abs: rs, Γ ⊢v Sabs cs ↓ Vabs cs Γ |
  comb:
    rs, Γ ⊢v t ↓ Vabs cs Γ’ ⇒ rs, Γ ⊢v u ↓ u’ ⇒ vfind-match cs u’ = Some (env, -, rhs) ⇒
\[
rs, \Gamma' + f env \vdash_v \text{rhs} \downarrow \text{val} \quad \Rightarrow \\
rs, \Gamma \vdash_v \text{t}$s u \downarrow \text{val} \\
\text{rec-comb:} \\
rs, \Gamma \vdash_v \text{t} \downarrow \text{Vrecabs css name } \Gamma' \quad \Rightarrow \\
\text{fmllookup css name } = \text{Some cs} \quad \Rightarrow \\
rs, \Gamma \vdash_v \text{u} \downarrow \text{u'} \quad \Rightarrow \\
vfind-match cs \text{ u'} = \text{Some (env, - rhs)} \quad \Rightarrow \\
rs, \Gamma' + f env \vdash_v \text{rhs} \downarrow \text{val} \quad \Rightarrow \\
\text{constr:} \\
\text{name } \in\ C \quad \Rightarrow \\
\text{list-all2 (veval rs } \Gamma) \text{ ts us } \quad \Rightarrow \\
rs, \Gamma \vdash_v \text{name } \text{ $$$ ts } \downarrow \text{Vconstr name us}
\]

\text{lemma (in vrules) veval-wellformed:} \\
\text{assumes } rs, \Gamma \vdash_v \text{t} \downarrow \text{v wellformed t wellformed-venv } \Gamma \\
\text{shows } v\text{wellformed v} \\
\text{using } \text{assms proof induction} \\
\text{case const} \\
\text{thus } \text{?case} \\
\text{using } \text{all-rules} \\
\text{by (auto simp: list-all-iff)} \\
\text{next} \\
\text{case comb} \\
\text{show } \text{?case} \\
\text{apply (rule comb)} \\
\text{using } \text{comb } \text{by (auto simp: list-all-iff dest: vfind-match-elem intro: vwellformed.vmatch-env)} \\
\text{next} \\
\text{case } (\text{rec-comb } \Gamma \text{ t css name } \Gamma' \text{ cs u u'} \text{ env pat rhs val}) \\
\text{hence } (\text{pat, rhs} ) \in \text{set cs vmatch (mk-pat pat) u'} = \text{Some env} \\
\text{by (metis vfind-match-elem)+} \\
\text{show } \text{?case} \\
\text{proof (rule rec-comb)} \\
\text{have wellformed t} \\
\text{using rec-comb by simp} \\
\text{have } v\text{wellformed } (\text{Vrecabs css name } \Gamma') \\
\text{by (rule rec-comb) fact+} \\
\text{thus wellformed rhs} \\
\text{using rec-comb } ((\text{pat, rhs}) \in \text{set cs}) \\
\text{by (auto simp: list-all-iff)} \\
\text{have wellformed-venv } \Gamma' \\
\text{using } v\text{wellformed } (\text{Vrecabs css name } \Gamma') \text{ by simp} \\
\text{moreover have wellformed-venv env} \\
\text{using rec-comb vmatch (mk-pat pat) u' = Some env} \\
\text{by (auto intro: vwellformed.vmatch-env)} \\
\text{ultimately show wellformed-venv } (\Gamma' + f env)
next

case (constr name \( \Gamma \) ts us)

have list-all uwellformed us

using \( \langle \text{list-all2 - - -} \rangle (\text{wellformed (list-comb - -)}) \)

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 \( \text{vrules} \)) veval-closed:

assumes \( rs, \Gamma \vdash v \downarrow v \text{ closed-except } t \) (fmdom \( \Gamma \)) closed-venv \( \Gamma \)

assumes wellformed t wellformed-venv \( \Gamma \)

shows \( v \text{closed} v \)

using assms proof induction

case (const name rhs \( \Gamma \))

hence \( (\text{name}, rhs) \in \text{set } rs \)

by (auto dest: map-of-SomeD)

thus ?case

using const all-rules

by (auto simp: list-all-iff)

next

case (comb \( \Gamma \) t cs \( \Gamma' \) u u' env pat rhs val)

hence pat: \( (\text{pat, rhs}) \in \text{set cs vmatch (mk-pat pat) } u' = \text{Some env} \)

by (metis vfind-match-elem) +

show ?case

proof (rule comb)

have \( v \text{closed} u' \)

using comb by (auto simp: \text{Sterm.closed-except-simps})

have closed-venv env

by (rule vclosed.vmatch-env) fact+

thus closed-venv \( (\Gamma' + + f \text{ env}) \)

using comb by (auto simp: \text{Sterm.closed-except-simps})

next

have wellformed t

using comb by simp

have \( v \text{wellformed (Vabs cs } \Gamma' \) \)

by (rule veval-wellformed) fact+

thus wellformed rhs

using pat by (auto simp: list-all-iff)

have wellformed-venv \( \Gamma' \)

using \( \langle v \text{wellformed (Vabs cs } \Gamma' \rangle \) by simp
moreover have \textit{wellformed-venv} env
using \textit{comb} \textit{pat}
by (auto intro: \textit{vwellformed.vmatch-env veval-wellformed})
ultimately show \textit{wellformed-venv} (Γ’ ++ \textit{env})
by blast

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

moreover have \textit{fmdom} \textit{env} = \textit{frees pat}
using \langle \textit{vwellformed} (Vabs cs Γ’): \textit{pat} \rangle by (auto simp: \textit{vmatch-dom mk-pat-frees list-all-iff})

ultimately show \textit{closed-except rhs} (\textit{fmdom} (Γ’ ++ \textit{f env}))
using \langle \textit{vclosed} (Vabs cs Γ’): \textit{pat} \rangle by simp
qed

next
case (\textit{rec-comb} Γ \textit{t css name} Γ’ \textit{cs u u’ env rhs val})
hence \textit{pat}: (\textit{pat}, \textit{rhs}) ∈ \textit{set cs vmatch} (\textit{mk-pat pat}) u’ = Some env
by (metis \textit{vfind-match-elem})+

show \textit{?case}
proof (rule \textit{rec-comb})
have \textit{vclosed u’}
using \textit{rec-comb} by (auto simp: \textit{Sterm.closed-except-simps})

have \textit{closed-venv} \textit{env}
by (rule \textit{vclosed.vmatch-env}) fact+
thus \textit{closed-venv} (Γ’ ++ \textit{f env})
using \textit{rec-comb} by (auto simp: \textit{Sterm.closed-except-simps})

next
have \textit{wellformed} \textit{t}
using \textit{rec-comb} by simp

have \textit{vwellformed} (Vrecabs css name Γ’)
by (rule \textit{veval-wellformed}) fact+
thus \textit{wellformed} \textit{rhs}
using \textit{pat rec-comb} by (auto simp: \textit{list-all-iff})

have \textit{wellformed-venv} Γ’
using \langle \textit{vwellformed} (Vrecabs css name Γ’): \textit{by simp} \rangle
moreover have \textit{wellformed-venv} \textit{env}
using \textit{rec-comb} \textit{pat}
by (auto intro: \textit{vwellformed.vmatch-env veval-wellformed})
ultimately show \textit{wellformed-venv} (Γ’ ++ \textit{f env})
by blast

have \textit{wellformed-clauses} \textit{cs}
using \langle \text{wellformed} (\text{Vrecabs css name } \Gamma') \rangle \langle \text{fmlookup css name } = \text{Some cs} \rangle 
by \text{auto}

have \text{vclosed} (\text{Vrecabs css name } \Gamma')
using \text{rec-comb} \by \text{(auto simp: \text{Sterm.closed-except-simps})}
hence \text{closed-except rhs} (\text{fmdom } \Gamma' \cup \text{frees pat})
using \text{rec-comb pat} \by \text{(auto simp: list-all-iff)}
moreover have \text{fmdom env} = \text{frees pat}
using \langle \text{wellformed-clauses cs} \rangle \text{pat}
by \text{(auto simp: list-all-iff \text{vmatch-dom mk-pat-frees})}
ultimately show \text{closed-except rhs} (\text{fmdom} (\Gamma' \++ f \text{ env}))
using \langle \text{vclosed} (\text{Vrecabs css name } \Gamma') \rangle 
by \text{simp}
qed

next
case (\text{constr name } \Gamma \ ts \ us)
have \text{list-all vclosed us}
using \langle \text{dist-all2 - - } \rangle (\text{closed-except (- \ $$ \ -) \ \text{wellformed} \ (\text{name $$ \ ts})}:
proof (\text{induction ts us rule: list.rel-induct})
case (\text{Cons v vs u us})
with constr show \text{?case}
  unfolding \text{closed-list-comb wellformed.list-comb}
  by (\text{auto simp: list-all-iff \text{Sterm.closed-except-simps})}
qed \text{simp}
thus \text{?case}
  by (\text{simp add: list-all-iff})
qed \text{(auto simp: \text{Sterm.closed-except-simps})}

lemma (in vrules) \text{veval-constructor-value}:
assumes \text{rs, } \Gamma \vdash v \downarrow v \text{vconstructor-value-env} \Gamma
shows \text{vconstructor-value} v
using assms proof induction
case (\text{comb } \Gamma' \ ts \ cs \ \Gamma' \ u \ u' \ env \ pat \ rhs \ val)
hence (\text{pat, rhs} \in \text{set cs} v\text{match} (\text{mk-pat pat}) \ u' = \text{Some env})
  by (metis vfind-match-elem)+
hence \text{vconstructor-value-env} (\Gamma' \++ f \text{ env})
  using \text{comb} \by \text{(auto intro: vconstructor-value.vmatch-env)}
thus \text{?case}
  using \text{comb} \by \text{auto}
next
case (\text{constr name } \Gamma \ ts \ us)
hence \text{list-all vconstructor-value us}
  by (\text{auto elim: list-all2-rightE})
with constr show \text{?case}
  by \text{simp}
next
case \text{const}
thus \text{?case}
  using \text{vconstructor-value-rs}

166
by (auto simp: list-all-iff vconstructor-value-rs-def)
next
case (rec-comb Γ t css name Γ' cs u u' env pat rhs val)
hence (pat, rhs) ∈ set cs vmatch (mk-pat pat) u' = Some env
by (metis vfind-match-elem)+
hence vconstructor-value-env (Γ' ++f env)
using rec-comb by (auto intro: vconstructor-value.vmatch-env)
thus ?case
using rec-comb by auto
qed (auto simp: list-all-iff vconstructor-value-rs-def)

lemma (in vrules) veval-welldefined:
assumes rs, Γ ⊢ v t ↓ v fmpred (λ-. vwelldefined) Γ welldefined t
shows vwelldefined v
using assms proof induction
  case const
  thus ?case
  using vwelldefined-rs assms
  unfolding list-all-iff
by (auto simp: list-all-iff)
next
case (comb Γ t cs Γ' u u' env 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 vwelldefined 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 \textup{fmpred}\ (\lambda \cdot \textup{wvwelldefined})\ (\Gamma' \++_f \textup{env})

proof
  show \textup{fmpred}\ (\lambda \cdot \textup{wvwelldefined})\ \textup{env}
  using \textsc{rec-comb} by (auto dest: vfind-match-elem intro:vwelldefined.vmatch-env)
next
  show \textup{fmpred}\ (\lambda \cdot \textup{wvwelldefined})\ \Gamma'
  using \textsc{rec-comb} by auto
qed

next
have \textup{wvwelldefined}\ (Vrecabs\ css\ name\ \Gamma')
  using \textsc{rec-comb} by auto
thus \textup{welldefined}\ \textup{rhs}
  apply simp
  apply (elim\ \textsc{conjE})
  apply (drule\ \textup{fmpredD}[where\ m = \textsc{css}])
  using \langle (\textsc{pat}, \textup{rhs}) \in \textup{set\ cs} \rangle\ \textsc{rec-comb} by (auto simp: list-all-iff)
qed

next
case (\textsc{constr} name\ \Gamma\ ts\ us)
have \textup{list-all}\ \textup{wvwelldefined}\ \textup{us}
  using \langle \textup{list-all2} \ - \ - \ \rangle\\ \langle \textup{welldefined}\ (-\ $$-\ )\rangle
proof (induction\ ts\ us\ rule:\ \textsc{list.rel-induct})
  case (\textsc{Cons} \textsc{v} \textsc{vs} \textsc{u} \textsc{us})
    with \textsc{constr} show \textup{case}
    unfolding \textup{welldefined.\list-comb}
    by auto
  qed simp
with \textsc{constr} show \textup{case}
  by (simp\ add: \textup{list-all-iff}\ \textup{all-consts-def})
next
  case \textsc{abs}
  thus \textup{case}
    unfolding \textup{welldefined-sabs} by auto
qed\ \textit{auto}

\textbf{Correctness wrt} constructors.seval

\textit{context vrules begin}

\textbf{definition} rs':: \textup{srule\ list\ where}
rs' = \textup{value-to-sterm-rules}\ rs

\textbf{lemma} \textup{value-to-sterm-srules}: srules C-info rs'
proof
  show \textup{distinct}\ (\textup{map\ \textsc{fst}\ rs'})
    unfolding rs'-def
    using \textup{distinct}\ by\ auto
next
show \textit{list-all srule rs'}

unfolding \textit{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 \texttt{a b} by (cases \texttt{b}) (auto simp: is-abs-def term-cases-def)
done

next

show \textit{fdisjnt (fst |'| fset-of-list rs')} \texttt{C}
using vconstructor-value-rs unfolding \textit{rs'}-def vconstructor-value-rs-def
by auto

interpret \texttt{c: constants - fst |'| fset-of-list rs'}
by standard (fact | fact distinct-ctr)+
have all-cons: \texttt{c.all-cons = all-cons}
unfolding c.all-cons-def all-cons-def
by (simp add: \textit{rs'}-def)
have shadows-cons: \texttt{c.shadows-cons \texttt{rhs = shadows-cons \texttt{rhs}} for \texttt{rhs :: sterm}}
by (induction \texttt{rhs}; fastforce simp: all-cons list-ex-iff)

have \textit{list-all (\lambda z. rhs). \neg shadows-cons \texttt{rhs \texttt{rs'}}} \textit{rs'}
unfolding \textit{rs'}-def
unfolding list.pred-map map-prod-def id-def case-prod-twice list-all-iff
apply auto
unfolding comp-def all-cons-def
using not-shadows
by (fastforce simp: list-all-iff dest: not-shadows-vconsts.value-to-sterm)
thus \textit{list-all (\lambda z. rhs). \neg c.shadows-cons \texttt{rhs \texttt{rs'}}} \textit{rs'}
unfolding shadows-cons .

have \textit{list-all (\lambda z. rhs). welldefined \texttt{rhs \texttt{rs'}}} \textit{rs'}
unfolding \textit{rs'}-def list.pred-map
apply (rule list.pred-mono-strong[OF vwelldefined-rs])
subgoal for \texttt{z}
apply (cases \texttt{z}; hypsubst-thin)
apply simp
apply (erule vwelldefined.value-to-sterm)
done

moreover have \textit{map fst rs = map fst rs'}
unfolding \textit{rs'}-def by simp
ultimately have \textit{list-all (\lambda z. rhs). welldefined \texttt{rhs \texttt{rs'}}} \textit{rs'}
by simp
thus \textit{list-all (\lambda z. rhs). c.welldefined \texttt{rhs \texttt{rs'}}} \textit{rs'}
unfolding all-cons .

next

show \textit{distinct all-constructors}
by (fact distinct-ctr)
qed

169
When we evaluate terms using `veval`, the result is a `value` which possibly contains a closure (constructor `Vabs`). Such a closure is essentially a case-lambda (like `Sabs`), but with an additionally captured environment of type `string → value` (which is usually called $\Gamma'$). The contained case-lambda might not be closed.

The proof idea is that we can always substitute with $\Gamma'$ and obtain a regular `term` value. The only interesting part of the proof is the case when a case-lambda gets applied to a value, because in that process, a hidden environment is unveiled. That environment may not bear any relation to the active environment $\Gamma$ at all. But pattern matching and substitution proceeds only with that hidden environment.

```plaintext
class vrules begin

private lemma veval-correct0: 
  assumes rs, $\Gamma \vdash v \downarrow v \text{ wellformed } t \text{ wellformed-venv } \Gamma$
  assumes closed-except t (fmdom $\Gamma$) closed-venv $\Gamma$
  assumes econstructor-value-venv $\Gamma$
  shows rs', fmmap value-to-sterm $\Gamma \vdash t \downarrow \text{ value-to-sterm } v$

using assms proof induction

  case (constr name $\Gamma$ ts us)
  have list-all2 (seval rs' (fmmap value-to-sterm $\Gamma$)) ts (map value-to-sterm us)
    unfolding list-all2-map2
    proof (rule list.rel-mono-strong[OF list-all2 - - -], elim conjE impE)
      fix t u
      assume t ∈ set ts u ∈ set us
      assume rs, $\Gamma \vdash v \downarrow u$

      show wellformed t closed-except t (fmdom $\Gamma$)
        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 $\Gamma$ t cs $\Gamma'$ u u' venv 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 vvwellformed (Vabs cs Γ')
  by (rule veval-wellformed) fact+

hence
  list-all (linear o fst) cs
  wellformed-venv Γ'
  by (auto simp: list-all-iff split-beta)

have rel-option match-related (vfind-match cs u') (find-match cs (value-to-sterm u'))
  apply (rule find-match-eq)
  apply fact
  apply (rule veval-constructor-value)
  apply fact+
  done

then obtain senv
  where find-match cs (value-to-sterm u') = Some (senv, pat, rhs)
  and env-eq venv senv
  using vfind-match - - = -
  by cases auto
hence (pat, rhs) ∈ set cs match pat (value-to-sterm u') = Some senv
  by (auto dest: find-match-elem)
hence fdom 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 fdom venv = frees pat
  by simp

have closed-except t (fdom 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 \( \text{wellformed } u \)
  using \( \text{comb by } \text{simp} \)
have \( \text{wellformed } u' \)
  by (rule \( \text{veval-wellformed} \)) \( \text{fact}+ \)
have \( \text{wellformed-venv } \text{venv} \)
  by (rule \( \text{veval-wellformed} \). \( \text{vmatch-venv} \)) \( \text{fact}+ \)

have \( \text{closed-except } u \) (\( \text{fmdom } \Gamma \))
  using \( \text{comb by } \) (simp add: \( \text{closed-except-def} \))
have \( \text{velosed } u' \)
  by (rule \( \text{veval-closed} \)) \( \text{fact}+ \)
have \( \text{closed-venv } \text{venv} \)
  by (rule \( \text{velosed} \). \( \text{vmatch-venv} \)) \( \text{fact}+ \)

have \( \text{closed-venv } \Gamma' \)
  using \( \text{velosed } (\text{Vabs } cs \Gamma') \) by simp

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

1. We know the following (induction hypothesis): \( rs', \text{fmmap } \text{value-to-sterm } (\Gamma' + + f \text{venv}) \downarrow_s \text{rhs } \downarrow \text{value-to-sterm } \text{val} \)

2. ... first, we can deduce using \( \text{ssubst-eval} \) that this is equivalent to substituting \( \text{rhs first}: rs', \text{fmmap } \text{value-to-sterm } (\Gamma' + + f \text{venv}) \downarrow_s \text{subst } \text{rhs } (\text{fmdrop-fset } (\text{frees } \text{pat}) \text{ (fmmap } \text{value-to-sterm } \Gamma')) \downarrow \text{value-to-sterm } \text{val} \)

3. ... second, we can replace the hidden environment \( \Gamma' \) by the active environment \( \Gamma \) using \( \text{seval-agree-eq} \) because it does not contain useful information at this point: \( rs', \text{fmmap } \text{value-to-sterm } (\Gamma + + f \text{venv}) \downarrow_s \text{subst } \text{rhs } (\text{fmdrop-fset } (\text{frees } \text{pat}) \text{ (fmmap } \text{value-to-sterm } \Gamma')) \downarrow \text{value-to-sterm } \text{val} \)

4. ... finally we can apply a step in the original semantics and arrive at the conclusion: \( rs', \text{fmmap } \text{value-to-sterm } \Gamma \downarrow_s t \$_x \text{u } \downarrow \text{value-to-sterm } \text{val} \)

have \( rs', \text{fmmap } \text{value-to-sterm } (\Gamma' + + f \text{venv}) \downarrow_s ?\text{subst } \text{pat } \text{rhs } \downarrow \text{value-to-sterm } \text{val} \)
proof (rule \( \text{ssubst-eval} \))
  show \( rs', \text{fmmap } \text{value-to-sterm } (\Gamma' + + f \text{venv}) \downarrow_s \text{rhs } \downarrow \text{value-to-sterm } \text{val} \)
proof (rule \( \text{comb} \))
  have \( \text{linear } \text{pat } \text{closed-except } \text{rhs } (\text{fmdom } \Gamma' | | \text{frees } \text{pat}) \)
    using \( \langle \text{pat}, \text{rhs} \rangle \in \text{set } cs \rangle \langle \text{wellformed } (\text{Vabs } cs \Gamma') \rangle \langle \text{velosed } (\text{Vabs } cs \Gamma') \rangle \)
    by (auto simp: \( \text{mk-pat-frees } \text{vmatch-dom} \))
  hence \( \text{closed-except } \text{rhs } (\text{fmdom } \Gamma' | | \text{fmdom } \text{venv}) \)
  using \( \text{vmatch } (\text{mk-pat } \text{pat}) u' = \text{Some } \text{venv} \)
  by (auto simp: \( \text{mk-pat-frees } \text{vmatch-dom} \))
  thus \( \text{closed-except } \text{rhs } (\text{fmdom } (\Gamma' + + f \text{venv})) \)
  by simp
next
show \( \text{wellformed-venv } (\Gamma' + + f \text{venv}) \)

172
using \( \text{wellformed-venv} \, \Gamma' \) \( \langle \text{wellformed-venv} \, \text{venv} \rangle \)
by blast
next
show \( \text{closed-venv} \, (\Gamma' + + f \, \text{venv}) \)
using \( \langle \text{closed-venv} \, \Gamma' \rangle \)
(\( \text{closed-venv} \, \text{venv} \) )
by blast
next
show \( \text{vconstructor-value-env} \, (\Gamma' + + f \, \text{venv}) \)
using \( \langle \text{vconstructor-value-env} \, \Gamma' \rangle \)
(\( \text{vconstructor-value-env} \, \text{venv} \) )
by blast
next
show \( \text{wellformed} \, \text{rhs} \)
using \( \langle \text{(pat, rhs)} \in \text{set} \, \text{cs} \rangle \)
(\( \text{vwellformed} \, (\text{Vabs} \, \text{cs} \, \Gamma') \) )
by (\( \text{fastforce simp: list-all-iff} \) )
qed

next
\[ \text{have} \, \text{fmdrop-fset} \, \text{fmdom} \, \text{venv} \, \Gamma' \subseteq f \, \Gamma' + + f \, \text{venv} \]  
including \( \text{fmap: lifting fset.lifting} \)
by \( \text{transfer'} \)  
\( \text{(auto simp: map-drop-set-def map-filter-def map-le-def map-add-def split: if-splits)} \)  
\[ \text{thus} \, \text{fmdrop-fset} \, \text{(frees pat)} \, \text{fmmap} \, \text{value-to-sterm} \, \Gamma' \subseteq f \, \text{fmmap} \, \text{value-to-sterm} \]  
(\( \Gamma' + + f \, \text{venv} \) )
\[ \text{unfolding} \, \text{(fmdom venv = frees pat);} \]  
by (\( \text{metis fmdrop-fset-fmmap fmmap-subset} \) )
next
show \( \text{closed-env} \, \text{(fmmap} \, \text{value-to-sterm} \, \Gamma' + + f \, \text{venv}) \)
apply auto
apply rule
apply \( \text{(rule vclosed.value-to-sterm-env, fact)+} \)
done
next
show \( \text{value-env} \, \text{(fmmap} \, \text{value-to-sterm} \, \Gamma' + + f \, \text{venv}) \)
apply auto
apply rule
apply \( \text{(rule vconstructor-value.value-to-sterm-env, fact)+} \)
done
qed

show \( \text{?case} \)
proof \( \text{(rule seval.comb)} \)
have \( \text{rs', fmmap} \, \text{value-to-sterm} \, \Gamma \vdash_s \, t \downarrow \text{value-to-sterm} \)  
(\( \text{Vabs} \, \text{cs} \, \Gamma') \)
using \( \text{comb by (auto simp: closed-except-def)} \)  
thus \( \text{rs', fmmap} \, \text{value-to-sterm} \, \Gamma \vdash_s \, t \downarrow \text{Sabs} \, (\lambda(pat, t). \, (\text{pat, subst pat t})) \, \text{cs}) \)  
by \( \text{simp} \)
next
show \( \text{rs', fmmap} \, \text{value-to-sterm} \, \Gamma \vdash_s \, u \downarrow \text{value-to-sterm} \, u' \)
using \( \text{comb by (simp add: closed-except-def)} \)
next

show \( r's', \text{fmmap value-to-sterm } \Gamma \overset{+f}{\mapsto} \text{env} \vdash \text{?subst pat rhs} \downarrow \text{value-to-sterm val} \)

proof (rule seval-agree-eq)

show \( r's', \text{fmmap value-to-sterm } \Gamma' \overset{+f}{\mapsto} \text{fmmap value-to-sterm venv} \vdash \text{?subst pat rhs} \downarrow \text{value-to-sterm val} \)

using \( r's', \text{fmmap value-to-sterm } (\Gamma' \overset{+f}{\mapsto} \text{venv}) \vdash \text{?subst pat rhs} \downarrow \text{value-to-sterm val} \); by simp

next

show \( \text{frarestrict-fset } (\text{frees pat}) \text{ (fmmap value-to-sterm } \Gamma' \overset{+f}{\mapsto} \text{env}) = \)

\( \text{frarestrict-fset } (\text{frees pat}) \text{ (fmmap value-to-sterm } \Gamma \overset{+f}{\mapsto} \text{env}) \)

proof

unfolding \( \text{env} = \cdots \)

apply \( \text{subt } \text{fndom env} = \text{frees pat} \text{[symmetric]} \)

apply \( \text{subt } \text{fndom-map}[\text{symmetric}] \)

apply \( \text{subt } \text{fnadd-restrict-right-dom} \)

apply \( \text{subt } \text{fmdom-map}[\text{symmetric}] \)

apply \( \text{subt } \text{fnadd-restrict-right-dom} \)

apply simp

done

next

have \( \text{closed } (\text{value-to-sterm } (Vabs \text{ cs } \Gamma')) \)

using \( \text{vclosed } (Vabs \text{ cs } \Gamma') \)

by \( \text{rule vclosed.value-to-sterm} \)

thus \( \text{closed-except } (\text{subt rhs } (\text{fndrop-fset } (\text{frees pat}) \text{ (fmmap value-to-sterm } \Gamma')) ) \)

\( (\text{frees pat}) \)

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

by \( \text{auto simp: Sterm.closed-except-simps list-all-iff} \)

next

show \( \text{closed-env } \text{fmmap value-to-sterm } \Gamma' \overset{+f}{\mapsto} \text{fmmap value-to-sterm venv} \)

using \( \text{closed-venv } \Gamma' \text{ (closed-venv venv)} \)

by \( \text{auto intro: vclosed.value-to-sterm-venv} \)

next

show \( \text{frees pat } \subseteq \text{ fmdom } \text{fmmap value-to-sterm } \Gamma' \overset{+f}{\mapsto} \text{fmmap value-to-sterm venv} \)

using \( \text{fmdom venv } = \text{frees pat} \)

by \text{fastforce}

next

show \( \text{closed-srules } r's' \)

using \text{all-rules}

unfolding \( r's'-\text{def list-all-iff} \)

by \( \text{fastforce intro: vclosed.value-to-sterm} \)

qed

next

show \( \text{find-match } (\text{map } (\lambda(p\text{at, t}). (p\text{at, ?subst pat t}) ) \text{ cs}) \text{ (value-to-sterm } u') = \text{ Some } (\text{env, pat, ?subst pat rhs}) \)

using \( \text{find-match } - - = - - \)

by \( \text{auto simp: find-match-map} \)
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 o fst) cs
wellformed-venv Γ'
  by (auto simp: list-all-iff split-beta)

have rel-option match-related (vfind-match cs u') (find-match cs (value-to-sterm u'))
  apply (rule find-match-eq)
  fact
  apply (rule veval-constructor-value)
  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 = fnmap 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 $t$ ($\text{fmdom } \Gamma$) wellformed $t$
  using rec-comb by (simp add: closed-except-def)+
have vclosed ($V_{\text{recabs css name } \Gamma}$)
  by (rule veval-closed) fact+
hence vclosed ($V_{\text{abs cs } \Gamma}$)
  using rec-comb by auto

have vconstructor-value $u'$
  by (rule veval-constructor-value) fact+
have vconstructor-value ($V_{\text{recabs css name } \Gamma}$)
  by (rule veval-constructor-value) fact+
hence vconstructor-value-env $\Gamma'$
  by simp
have vconstructor-value-env $venv$
  by (rule veval-constructor-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 vclosed. vmatch-env) fact+

have closed-except $u$ ($\text{fmdom } \Gamma$)
  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-$venv$ $\Gamma'$
  using (vclosed ($V_{\text{abs cs } \Gamma}$)): by simp

let $?\text{subst} = \lambda \text{pat} \ t. \ \text{subst} \ t$ ($\text{fmdrop-fset } (\text{frees pat})$ ($\text{fnmap value-to-sterm } \Gamma'$))
have $rs'$, $\text{fnmap value-to-sterm } (\Gamma' + + f \ venv) \vdash \ ?\text{subst \ pat \ rhs} \downarrow \text{value-to-sterm val}$
proof (rule ssubst-eval)
  show $rs'$, $\text{fnmap value-to-sterm } (\Gamma' + + f \ venv) \vdash \？\text{pat \ rhs} \downarrow \text{value-to-sterm val}$
  proof (rule rec-comb)
    have linear $\text{pat}$ closed-except rhs ($\text{fmdom } \Gamma' | | f \text{ dom venv}$)
      using $(\text{pat, rhs}) \in \text{set cs}$ ($\text{vwellformed } (V_{\text{abs cs } \Gamma} z)$): (vclosed ($V_{\text{abs cs } \Gamma}$)): by (auto simp: list-all-iff)
hence closed-except rhs ($\text{fmdom } \Gamma' | | f \text{ dom venv}$)
  using vmatch ($\text{mk-pat pat} \ u' = \text{Some venv}$)
  by (auto simp: mk-pat-frees vmatch-dom)
thus closed-except rhs ($\text{fmdom } (\Gamma' + + f \ venv)$)
  by simp
proof (rule seval.comb)
have rs', fnmap value-to-term Γ |- s t \downarrow value-to-term (Vrecabs cs name Γ')
  using rec-comb by (auto simp: closed-except-def)
thus rs', fnmap value-to-term Γ |- s t \downarrow Sabs (map (\lambda(pat, t). (pat, ?subst pat t)) cs)
  unfolding \langle cs = - \rangle by simp
qed
next
  show \(rs', \text{fmmap} \text{value-to-sterm} \Gamma \vdash_s u \downarrow \text{value-to-sterm} u'\)
  using rec-comb by (simp add: closed-except-def)
next
  show \(rs', \text{fmmap} \text{value-to-sterm} \Gamma' \downarrow f \text{fmmap} \text{value-to-sterm} venv \vdash_s \text{?subst pat rhs} \downarrow \text{value-to-sterm val}\)
proof (rule seval-agree-eq)
    show \(rs', \text{fmmap} \text{value-to-sterm} (\Gamma' \downarrow f \text{venv}) \vdash_s \text{?subst pat rhs} \downarrow \text{value-to-sterm val}\) by simp
next
  show \(\text{fmarestrict-fset} (\text{frees pat}) (\text{fmmap} \text{value-to-sterm} \Gamma' \downarrow f \text{fmmap value-to-sterm venv}) = \text{fmarestrict-fset} (\text{frees pat}) (\text{fmmap} \text{value-to-sterm} \Gamma' \downarrow f \text{venv})\)
  unfolding (venv = -)
  apply (subst (fdom venv = frees pat)[symmetric])+
  apply (subst fdom-map[symmetric])
  apply (subst fnadd-restrict-right-dom)
  apply (subst fdom-map[symmetric])
  apply (subst fnadd-restrict-right-dom)
  apply simp
  done
next
  have \(\text{closed} (\text{value-to-sterm} (\text{Vabs cs} \Gamma'))\)
  using (rule vclosed (Vabs cs \Gamma'))
  by (rule vclosed.value-to-sterm)
  thus \(\text{closed-except} (\text{subst rhs} (\text{fndrop-fset} (\text{frees pat}) (\text{fmmap} \text{value-to-sterm} \Gamma'))) (\text{frees pat})\)
  using \((\text{pat}, \text{rhs}) \in \text{set cs}\)
  by (auto simp: Sterm.closed-except-simps list-all-iff)
next
  show \(\text{closed-venv} (\text{fmmap} \text{value-to-sterm} \Gamma' \downarrow f \text{fmmap value-to-sterm venv})\)
  using \((\text{closed-venv} \Gamma') (\text{closed-venv} \text{venv})\)
  by (auto intro: vclosed.value-to-sterm-venv)
next
  show \(\text{frees pat} \subseteq \text{fdom} (\text{fmmap} \text{value-to-sterm} \Gamma' \downarrow f \text{fmmap value-to-sterm venv})\)
  using \((\text{fdom venv = frees pat})\)
  by fastforce
next
  show \(\text{closed-srules} rs'\)
  using \(\text{all-rules}\)
  unfolding \(rs'\)-def list-all-iff
  by (fastforce intro: vclosed.value-to-sterm)
qed
next
  show \(\text{find-match} (\text{map} (\lambda(\text{pat}, t). (\text{pat}, \text{?subst pat t}) \text{ cs}) \text{ (value-to-sterm u')}\)
= 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) ∈ Γ by force
next
  case abs
  show ?case
    by (auto simp del: fmdrop-fset-fmmap intro: seval.abs)
qed (auto intro: seval.var seval.abs)

lemma veval-correct:
  assumes rs, fmempty ⊢ v t ↓ v wellformed t closed t
  shows rs′, fmempty ⊢ s t ↓ value-to-sterm v
proof
  have rs′, fmmap value-to-sterm fmempty ⊢ s t ↓ value-to-sterm v
  using assms
  by (auto intro: veval-correct0 simp del: fmmap-empty)
  thus ?thesis
  by simp
qed

end end

3.5.3 Big-step semantics with conflation of constants and
variables

theory Big-Step-Value-ML
imports Big-Step-Value
begin

definition mk-rec-env :: (name, sclauses) fmap ⇒ (name, value) fmap ⇒ (name, value) fmap where
mk-rec-env css Γ′ = fmmap-keys (λname cs. Vrecabs css name Γ′) css

context special-constants begin

inductive veval′ :: (name, value) fmap ⇒ sterm ⇒ value ⇒ bool ⊢ v s t - v - s t - [50,0,50] 50) where
const: name |∅| C ⇒ fmlookup Γ name = Some val ⇒ Γ ′ ⊢ Sconst name s val
var: fmlookup Γ name = Some val ⇒ Γ ′ ⊢ Svar name v val

end

179
abs: $\Gamma \vdash_v Sabs\ cs \downarrow Vabs\ cs\ \Gamma$

comb:
- $\Gamma \vdash_v t \downarrow Vabs\ cs\ \Gamma' \implies \Gamma \vdash_v u \downarrow u' \implies$
- $vfind-match\ cs\ u' = Some\ (env,\ -,\ rhs) \implies$
- $\Gamma' + + f\ env \vdash_v rhs\ \downarrow val \implies$
- $\Gamma \vdash_v t\ s\ u \downarrow val$

rec-comb:
- $\Gamma \vdash_v t \downarrow Vrecabs\ css\ name\ \Gamma' \implies$
- $fmlookup\ css\ name = Some\ cs \implies$
- $\Gamma' + + f\ mk-rec-env\ css\ \Gamma' + + f\ env\ \vdash_v rhs\ \downarrow val \implies$
- $\Gamma \vdash_v t\ s\ u \downarrow val$

constr: $name\ |\in|\ C \implies\ list-all2\ (veval\ '\Gamma)\ ts\ us \implies\ \Gamma \vdash_v name\ \$\ ts\ \downarrow Vconstr\ name\ us$

lemma veval'-sabs-svarE:
- assumes $\Gamma \vdash_v Sabs\ cs\ s\ Svar\ n \downarrow v$
- obtains $u'\ env\ pat\ rhs$
  - where $fmlookup\ \Gamma\ n = Some\ u'$
  - $vfind-match\ cs\ u' = Some\ (env,\ -,\ rhs)$
  - $\Gamma' + + f\ env \vdash_v rhs\ \downarrow val$

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 $(comb\ cs'\ \Gamma'\ u'\ env\ pat\ rhs)$
- moreover have $fmlookup\ \Gamma\ n = Some\ u'$
  - using $\Gamma \vdash_v Svar\ n \downarrow u'$
  - proof cases
    - case $(constr\ name\ ts)$
      - hence False
        - by $(fold\ free-sterm-def)$ simp
      - thus $?thesis$ by simp
  - qed auto

moreover have $cs = cs'\ \Gamma = \Gamma'$
- using $\Gamma \vdash_v Sabs\ cs \downarrow Vabs\ cs'\ \Gamma'$
- by $(cases;\ auto)+
ultimately show \(?thesis
using that by auto
qed

lemma \(\text{vwell'}\text{vwellformed}:\)
assumes \(\Gamma \vdash v t \downarrow v \text{wellformed } t \text{ wellformed-venv } \Gamma\)
shows \(v\text{wellformed } v\)
using \(\text{assms proof induction}\)
case \(\text{comb}\)
show \(?\text{case}\)
apply (rule \(\text{comb}\))
using \(\text{comb by (auto simp: list-all-iff \(vfind-match-\text{elem intro: vwellformed.\text{venv}\))}\)
next
case (\(\text{rec-comb } \Gamma \ t \ \text{css name } \Gamma' \ \text{cs } u \ u' \ \text{env pat rhs val}\))
have \((\text{pat, rhs}) \in \text{set cs}\)
by (rule \(vfind-match-\text{elem}) \text{ fact})
show \(?\text{case}\)
proof (rule \(\text{rec-comb}\))
show \(\text{wellformed-venv} (\Gamma' \text{ ++f mk-rec-env css } \Gamma' \text{ ++f env})\)
proof (intro \(\text{fmprep-add}\))
show \(\text{wellformed-venv } \Gamma'\)
using \(\text{rec-comb by auto}\)
next
show \(\text{wellformed-venv env}\)
using \(\text{rec-comb by (auto dest: vfind-match-\text{elem intro: vwellformed.\text{venv})}\})
next
show \(\text{wellformed-venv} (\text{mk-rec-env css } \Gamma')\)
unfolding \(\text{mk-rec-env-def}\)
using \(\text{rec-comb by (auto intro: \(\text{fdomI})\}\})
qed
next
have \(\text{vwellformed} (\text{Vrecabs css name } \Gamma')\)
unfolding \(\text{mk-rec-env-def}\)
using \(\text{rec-comb by (auto intro: \(\text{fdomI})\})\)
thus \(\text{wellformed rhs}\)
using \((\text{pat, rhs}) \in \text{set cs}) \text{ rec-comb by (auto simp: list-all-iff)}\)
qed
next
case (\(\text{constr name } \Gamma \ ts \ us\))
have \(\text{list-all vwellformed us}\)
using \(\text{list-all2 - - \(\text{wellformed}\ (- \ $$ -)\)}\)
proof (induction \(\text{ts us as rule: list.rel-induct})
case (\(\text{Cons } v \ vs \ u \ us\))
thus \(?\text{case}\)
using \(\text{constr by (auto simp: app-term-def wellformed.list-comb)}\)
qed simp
thus \(?\text{case}\)
by (simp add: list-all-iff)

qed auto

lemma (in constants) veval′-shadows:
assumes \( \Gamma \vdash_v t \downarrow v \) not-shadows-vconst-env \( \Gamma \vdash \neg \text{shadows-consts } t \)
shows not-shadows-vconst-env 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-vconst-env)

next
case (rec-comb \( \Gamma \) t css name \( \Gamma' \) cs u u' env rhs val)
have (pat, rhs) \( \in \) set cs
by (rule vfind-match-elem) fact
show ?case
proof (rule rec-comb)
show not-shadows-vconst-env (\( \Gamma' \) ++f mk-rec-env css \( \Gamma' \) ++f env)
proof (intro fnpred-add)
show not-shadows-vconst-env env
using rec-comb by (auto dest: vfind-match-elem intro: not-shadows-vconst-env)
next
show not-shadows-vconst-env (mk-rec-env css \( \Gamma' \))
unfolding mk-rec-env-def
using rec-comb by (auto intro: fdomI)
next
show not-shadows-vconst-env \( \Gamma' \)
using rec-comb by auto
qed

next
have not-shadows-vconst (Vrecabs css name \( \Gamma' \))
using rec-comb by auto
thus \( \neg \) shadows-consts rhs
using :(pat, rhs) \( \in \) set cs rec-comb by (auto simp: list-all-iff)
qed

next
case (constr name \( \Gamma \) ts us)
have list-all (not-shadows-vconst) us
using \( \langle \text{list-all2 - - -} \rangle (\neg \text{shadows-consts } \langle \text{name } \S \S \text{ ts}\rangle)\)
proof (induction ts us rule: list.rel-induct)
case (Cons v v vs us)
thus ?case
using constr by (auto simp: shadows-list-comb app-term-def)
qed simp

thus ?case
by (simp add: list-all-iff)
qed (auto simp: list-all iff list-ex-iff)

lemma veval′-closed:
assumes $\Gamma \vdash_v t \downarrow v$ closed-except $t$ (fmdom $\Gamma$) closed-venv $\Gamma$
assumes wellformed $t$ wellformed-venv $\Gamma$
shows $v$ closed $v$
using assms proof induction
  case $(\text{comb} \Gamma t cs \Gamma' u u' \text{ env pat rhs val})$
hence $v$ closed $(Vabs cs \Gamma')$
    by $(\text{auto simp: closed-except-def})$

have $(\text{pat, rhs}) \in \text{set cs} \text{ vmatch (mk-pat pat) } u' = \text{ Some env}$
  by $(\text{rule vfind-match-elem; fact})$
hence $\text{fmdom env = patvars (mk-pat pat)}$
  by $(\text{simp add: vmatch-dom})$

have $v$wellformed $(Vabs cs \Gamma')$
  apply $(\text{rule veval'}$-wellformed)
  using comb by auto
hence $\text{linear pat}$
  using $(\text{pat, rhs}) \in \text{set cs}$
  by $(\text{auto simp: list-all-iff})$
hence $\text{fmdom env = frees pat}$
  unfolding $(\text{fmdom env = -})$
  by $(\text{simp add: mk-pat-frees})$

show $?\text{case}$
  proof $(\text{rule comb})$
    show wellformed rhs
      using $(\text{pat, rhs}) \in \text{set cs} \text{ } v$wellformed $(Vabs cs \Gamma')$
      by $(\text{auto simp: list-all-iff})$
  next
    show closed-venv $(\Gamma' + +_f \text{ env})$
      apply rule
      using $(\text{v}$closed $(Vabs cs \Gamma')$: apply auto)
      apply $(\text{rule vclosed.vmatch-env})$
      apply $(\text{rule vfind-match-elem})$
      using comb by $(\text{auto simp: closed-except-def})$
  next
    show closed-except rhs $(\text{fmdom (}\Gamma' + +_f \text{ env}))$
      using $(\text{v}$closed $(Vabs cs \Gamma')$: $\text{fmdom env = frees pat} \text{ } (\text{pat, rhs}) \in \text{set cs}$
      by $(\text{auto simp: list-all-iff})$
  next
    show wellformed-venv $(\Gamma' + +_f \text{ env})$
      apply rule
      using $(\text{v}$wellformed $(Vabs cs \Gamma')$: apply auto)
      apply $(\text{rule vwellformed.vmatch-env})$
      apply $(\text{rule vfind-match-elem})$
      apply fact
      apply $(\text{rule veval'}$-wellformed)
      using comb by auto
qed
next

case (rec-comb \(\Gamma t\) css name \(\Gamma' cs u u' env rhs\) val)

have \((pat, rhs) \in set cs\) \(vmatch (mk-pat pat) u' = Some env\)
  by (rule \(\forall find-match-elem\); fact)+

hence \(fdom env = \text{patvars} (mk-pat pat)\)
  by (simp add: \(vmatch-dom\))

have \(vwellformed (Vrecabs css name \(\Gamma'\))\)
  apply (rule \(\text{veval}'\)-wellformed)

hence \(\text{wellformed-clauses cs}\)
  using rec-comb by auto

hence \(\text{linear pat}\)
  using \(\langle (\text{pat}, rhs) \in set cs \rangle\)
  by (auto simp: \(\text{list-all-iff}\))

hence \(\text{fdom env} = \text{frees pat}\)
  unfolding \(\text{fdom env} = \cdot\)
  by (simp add: \(\text{mk-pat-frees}\))

show \(?case\)
  proof (rule rec-comb)
    show \(\text{closed-venv} (\(\Gamma' + + f\) \(\text{mk-rec-env css} \Gamma' + + f\) \(\text{env}\))\)
      proof (intro \(\text{fmpred-add}\))
        show \(\text{closed-venv} \(\Gamma'\)\)
          using rec-comb by (auto simp: \(\text{closed-except-def}\))

next
  show \(\text{closed-venv} env\)
    using rec-comb by (auto simp: \(\text{closed-except-def dest: \(\forall find-match-elem\}\}\)
      intro: \(\text{vclosed.vmatch-env}\))

next
  show \(\text{closed-venv} (\text{mk-rec-env css} \(\Gamma'\))\)
    unfolding \(\text{mk-rec-env-def}\)
    using rec-comb by (auto simp: \(\text{closed-except-def intro: \(\text{fdomI}\)}\))

qed

next

have \(\text{vclosed} (Vrecabs css name \(\Gamma'\))\)
  using \(\text{mk-rec-env-def}\)
  using rec-comb by (auto simp: \(\text{closed-except-def intro: \(\text{fdomI}\)}\))

hence \(\text{closed-except rhs} (\text{fdom} \(\Gamma' [\cup] \text{frees pat}\))\)
  apply simp
  apply (elim \(\text{conjE}\))
  apply (drule \(\text{fmpredD[where} m = css]\))
  apply (rule rec-comb)
  using \(\langle (\text{pat}, rhs) \in set cs \rangle\)
  unfolding \(\text{list-all-iff by auto}\)

thus \(\text{closed-except rhs} (\text{fdom} (\(\Gamma' + + f\) \(\text{mk-rec-env css} \Gamma' + + f\) \(\text{env}\)))\)
  unfolding \(\text{closed-except-def}\)
  using \(\text{fdom env} = \text{frees pat}\)
  by auto
next
  show wellformed rhs
  using (wellformed-clauses cs) ((\pat, \rhs) \in set cs)
  by (auto simp: list-all-iff)

next
  show wellformed-venv (\Gamma^' ++ f mk-rec-env css \Gamma^' ++ f env)
  proof (intro fnpred-add)
    show wellformed-venv \Gamma^'
      using (wellformed (Vrecabs css name \Gamma')) by auto
  next
    show wellformed-venv env
      using rec-comb by (auto dest: vfind-match-elem intro: veval^'-wellformed vwwellformed.vmatch-env)
  next
    show wellformed-venv (mk-rec-env css \Gamma^' ++ f env)
      unfolding mk-rec-env-def
      using (wellformed (Vrecabs css name \Gamma')) by (auto intro: fmdomI)
  qed
  qed

next
  case (constr name \Gamma 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: Stern.closed-except-simps)
  qed simp
  thus ?case
  by (simp add: list-all-iff)
  qed (auto simp: Stern.closed-except-simps)

primrec vwelldefined' :: value \Rightarrow bool where
vwelldefined' (Vconstr name vs) \longleftrightarrow list-all vwelldefined' vs |
vwelldefined' (Vabs cs \Gamma) \longleftrightarrow
  pred-fmap id (fmmap vwelldefined' \Gamma) \land
  list-all (\lambda(pat, t). const t \subseteq| \fmdom \Gamma \cup C) cs \land
  fdisjnt C (fmdom \Gamma) |
vwelldefined' (Vrecabs css name \Gamma) \longleftrightarrow
  pred-fmap id (fmmap vwelldefined' \Gamma) \land
  pred-fmap (\lambda cs.
    list-all (\lambda(pat, t). const t \subseteq| \fmdom \Gamma \cup (C \cup \fmdom css)) cs \land
    fdisjnt C (fmdom \Gamma) \cup css \land
    name \in| fmdom css \land
    fdisjnt C (fmdom css)

lemma vmatch-welldefined':
  assumes vmatch pat v = Some env vwelldefined' v
shows \( \text{fnpred} \ (\lambda. \text{vwelldefined'}) \ \text{env} \)

using assms proof (induction \( \text{pat} \ v \ \text{arbitrary}; \text{env rule: vmatch-induct} \))

\text{case (constr name ps name' vs)}

hence
map-option \( (\text{foldl} \ (++) \ \text{fmempty}) \ (\text{those} \ (\text{map2} \ \text{vmatch} \ \text{ps} \ \text{vs})) = \text{Some env} \)
name = name' length ps = length vs
by \((\text{auto split: if-splits})\)
then obtain \( \text{envs} \) where \( \text{env} = \text{foldl} \ (++) \ \text{fmempty} \ \text{envs} \ \text{map2} \ \text{vmatch} \ \text{ps} \ \text{vs} = \text{map} \ \text{Some} \ \text{envs} \)
by \((\text{blast dest: those-someD})\)

moreover have \( \text{fnpred} \ (\lambda. \text{vwelldefined'}) \ \text{env} \) if \( \text{env} \in \text{set} \ \text{envs} \) for \( \text{env} \)
proof
- from that have \( \text{Some env} \in \text{set} \ (\text{map2} \ \text{vmatch} \ \text{ps} \ \text{vs}) \)
- unfolding \((\text{map2} \ . \ldots = \cdot)\) by simp
then obtain \( p \ v \) where \( p \in \text{set ps} \ v \in \text{set vs} \ \text{vmatch} \ p \ v = \text{Some env} \)
by \((\text{auto elim: map2-elemE})\)

hence \( \text{vwelldefined'} \ v \)
using constr by \((\text{simp add: list-all-iff})\)

show \( \text{?thesis} \)
by \((\text{rule constr; safe?})\) fact+

qed

ultimately show \( \text{?case} \)
by auto

qed auto

lemma \( \text{sconsts-list-comb} \):
\begin{align*}
\text{consts} \ (\text{list-comb} \ f \ \text{xs}) |\subseteq| S & \longleftrightarrow \text{consts} f |\subseteq| S \land \text{list-all} \ (\lambda x. \text{consts} x |\subseteq| S) \ \\
\text{xs} & \end{align*}
by \((\text{induction} \ \text{xs} \ \text{arbitrary:} f)\) auto

lemma \( \text{sconsts-sabs} \):
\begin{align*}
\text{consts} \ (\text{Sabs} \ \text{cs}) |\subseteq| S & \longleftrightarrow \text{list-all} \ (\lambda (t, \text{consts} t |\subseteq| S) \ \text{cs} \\
& \end{align*}
apply \((\text{auto simp: list-all-iff ffUnion-alt-def dest: ffUnion-least-rev})\)
apply \((\text{subst (asm) list-all-iff-fset[symmetric]})\)
apply \((\text{auto simp: list-all-iff fset-of-list-elem})\)
done

lemma \( \text{(in constants) veval'-welldefined'} \):
assumes \( \Gamma \vdash_v t \downarrow v \text{fdisj} \ C \ (\text{fdom} \ \Gamma) \)
assumes \( \text{consts} t |\subseteq| \text{fdom} \ \Gamma \cup \ C \ \text{fnpred} \ (\lambda. \text{vwelldefined'}) \ \Gamma \)
assumes \( \text{wellformed} t \ \text{wellformed-veve} \ \Gamma \)
assumes \( \neg \text{shadows-consts} t \ \text{not-shadows-vconsts-env} \ \Gamma \)
shows \( \text{vwelldefined'} \ v \)
using assms proof induction
\text{case (abs} \ \Gamma \ \text{cs)}
\text{thus} \( \text{?case} \)

186
unfolding \texttt{sconsts-sabs}
by (auto simp: list-all-iff list-ex-iff)

next

\textbf{case} (\texttt{comb} \(\Gamma \ t \ cs \ \Gamma' \ u \ u' \ env \ pat \ rhs \ val))
\textbf{hence} \((\texttt{pat}, \texttt{rhs}) \in \text{set} \ cs) 
by (auto dest: \texttt{vfind-match-elem})

moreover \textbf{have} \texttt{vwelldefined'} (\texttt{Vabs} \ cs \ \Gamma')
using \texttt{comb} by auto

ultimately \textbf{have} \texttt{consts} rhs |\subseteq| fmdom \(\Gamma' \cup \ C)
by (auto simp: list-all-iff)

\textbf{have} \texttt{vwellformed} (\texttt{Vabs} \ cs \ \Gamma')
\textbf{apply} (rule veval'-wellformed)
using \texttt{comb} by auto
\textbf{hence} \texttt{linear} \texttt{pat}
using \langle \texttt{pat}, \texttt{rhs} \rangle \in \text{set} \ cs
by (auto simp: list-all-iff)

\textbf{have} \texttt{not-shadows-vconsts} (\texttt{Vabs} \ cs \ \Gamma')
\textbf{apply} (rule veval'-shadows)
using \texttt{comb} by auto

\textbf{show} \texttt{?case}
\textbf{proof} (rule \texttt{comb})
\textbf{show} \texttt{consts} rhs |\subseteq| fmdom \(\Gamma' + + \ f \ env) |\cup| \ C
using \langle \texttt{consts} rhs |\subseteq| fmdom \(\Gamma' \cup \ C) \rangle
by auto

\textbf{next}
\textbf{have} \texttt{vwelldefined'} \texttt{u} \texttt{'}
using \texttt{comb} by auto
\textbf{hence} \texttt{fmpred} (\lambda- \texttt{vwelldefined'}) \texttt{env}
using \texttt{comb}
by (auto intro: vmatch-welldefined' dest: vfind-match-elem)
\textbf{thus} \texttt{fmpred} (\lambda- \texttt{vwelldefined'}) (\texttt{\(\Gamma' + + \ f \ env)})
using \langle \texttt{vwelldefined'} (\texttt{Vabs} \ cs \ \Gamma') \rangle by auto

\textbf{next}
\textbf{have} \texttt{fdisjnt} \texttt{C} (\texttt{fmdom} \ \Gamma')
\textbf{using} \langle \texttt{vwelldefined'} (\texttt{Vabs} \ cs \ \Gamma') \rangle

187
by simp
moreover have \( \text{fdisjnt } C \ (\text{fdom env}) \)
unfolding \( \text{fdom env} = \text{frees pat} \)
using \((\text{pat, rhs}) \in \text{cs}\) \( \langle \text{not-shadows-vconsts} \ (\text{Vabs cs } \Gamma') \rangle \)
by (auto simp: list-all-iff all-consts-def fdisjnt-alt-def)
ultimately show \( \text{fdisjnt } C \ (\text{fmdom } (\Gamma' ++_f \text{ env})) \)
unfolding fdisjnt-alt-def by auto
next
show wellformed rhs
using \((\text{pat, rhs}) \in \text{cs}\) \( \langle \text{vwellformed} \ (\text{Vabs cs } \Gamma') \rangle \)
by (auto simp: list-all-iff)
next
have wellformed-venv \( \Gamma' \)
using \( \langle \text{vwellformed} \ (\text{Vabs cs } \Gamma') \rangle \)
by simp
moreover have wellformed-venv env
apply (rule vwellformed.vmatch-env)
apply (rule vfind-match-elem)
apply fact
apply (rule veval'.wellformed)
using comb by auto
ultimately show wellformed-venv \( (\Gamma' ++_f \text{ env}) \)
by blast
next
have not-shadows-vconsts-env \( \Gamma' \)
using \( \langle \text{not-shadows-vconsts} \ (\text{Vabs cs } \Gamma') \rangle \)
by simp
moreover have not-shadows-vconsts-env env
apply (rule not-shadows-vconsts.vmatch-env)
apply (rule vfind-match-elem)
apply fact
apply (rule veval'.shadows)
using comb by auto
ultimately show not-shadows-vconsts-env \( (\Gamma' ++_f \text{ env}) \)
by blast
next
show \( \neg \text{shadows-consts} \text{ rhs} \)
using \( \langle \text{not-shadows-vconsts} \ (\text{Vabs cs } \Gamma') \rangle \ (\text{pat, rhs}) \in \text{cs}\)
by (auto simp: list-all-iff)
qed
next
case \( \text{rec-comb} \ \Gamma \ t \ cs \ \text{name} \ \Gamma' \ cs \ u \ u' \ \text{env} \ \text{pat} \ \text{rhs} \ \text{val} \)
hence \((\text{pat, rhs}) \in \text{cs}\)
by (auto dest: vfind-match-elem)
moreover have \( \text{vwelldefined'} \ (\text{Vrecabs cs name } \Gamma') \)
using rec-comb by auto
ultimately have \( \text{consts} \ \text{rhs} \ |<| \ \text{fdom} \ \Gamma' |\cup| \ (C |\cup| \text{fdom} \ \text{css}) \)
using \( \langle \text{fmlookup } \text{css name} = \text{Some } \text{cs} \rangle \)
by (auto simp: list-all-iff dest!: fmpredD[where \( m = \text{css} \)])
have \( \text{vwellformed} \ (\text{Vrecabs cs name } \Gamma') \)

188
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 fnmpred (λ-. vwelldefined′) Γ'
    using ⟨vwelldefined′ (Vrecabs css name Γ′) ⟩ by auto
  moreover have fnmpred (λ-. vwelldefined′) (mk-rec-env css Γ′)
    unfolding mk-rec-env-def
    using rec-comb by (auto intro: fmdomI)
  moreover have fnmpred (λ-. vwelldefined′) env
    using rec-comb by (auto dest: vfind-match-elem intro: vmatch-welldefined′)
  ultimately show fnmpred (λ-. vwelldefined′) (Γ′ ++ f mk-rec-env css Γ′ ++ 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 fnlookup css name = Some cs ⟨(pat, rhs) ∈ set cs⟩ (not-shadows-vconsts (Vrecabs css name Γ′))
    apply auto
    apply (drule fnmpredD[where m = css])
    by (auto simp: list-all-iff all consts-def fdisjnt-alt-def)

moreover have \( \text{fdisjnt } C \ (\text{fdom } (\text{mk-rec-env css } \Gamma')) \)
unfolding \( \text{mk-rec-env-def} \)
using \( \langle \text{welldefined'} (Vrecabs css name \Gamma') \rangle \)
by \( \text{simp} \)
ultimately show \( \text{fdisjnt } C \ (\text{fdom } (\Gamma' + + f \text{ mk-rec-env css } \Gamma' + + f \text{ env})) \)
unfolding \( \text{fdisjnt-alt-def} \) by \( \text{auto} \)

next
show \( \text{wellformed } \text{rhs} \)
using \( \langle (\text{pat}, \text{rhs}) \in \text{set cs} \rangle \langle \text{wellformed-clauses cs} \rangle \)
by \( \text{(auto simp: list-all-iff)} \)

next
have \( \text{wellformed-env } \Gamma' \)
using \( \langle \text{wellformed } \ (Vrecabs css name \Gamma') \rangle \) by \( \text{simp} \)
moreover have \( \text{wellformed-env } (\text{mk-rec-env css } \Gamma') \)
unfolding \( \text{mk-rec-env-def} \)
using \( \langle \text{wellformed } \ (Vrecabs css name \Gamma') \rangle \)
by \( \text{(auto intro: fmdomI)} \)
moreover have \( \text{wellformed-env env} \)
apply \( \text{(rule vwellformed.vmatch-env)} \)
apply \( \text{(rule vfind-match-elem)} \)
apply \( \text{fact} \)
apply \( \text{(rule venvv.wellformed)} \)
using \( \text{rec-comb by auto} \)
ultimately show \( \text{wellformed-env } (\Gamma' + + f \text{ mk-rec-env css } \Gamma' + + f \text{ env}) \)
by \( \text{blast} \)

next
have \( \text{not-shadows-vconsts-env } \Gamma' \)
using \( \langle \text{not-shadows-vconsts } (Vrecabs css name \Gamma') \rangle \) by \( \text{simp} \)
moreover have \( \text{not-shadows-vconsts-env } (\text{mk-rec-env css } \Gamma') \)
unfolding \( \text{mk-rec-env-def} \)
using \( \langle \text{not-shadows-vconsts } (Vrecabs css name \Gamma') \rangle \)
by \( \text{(auto intro: fmdomI)} \)
moreover have \( \text{not-shadows-vconsts-env env} \)
apply \( \text{(rule not-shadows-vconsts.vmatch-env)} \)
apply \( \text{(rule vfind-match-elem)} \)
apply \( \text{fact} \)
apply \( \text{(rule venvv.shadows)} \)
using \( \text{rec-comb by auto} \)
ultimately show \( \text{not-shadows-vconsts-env } (\Gamma' + + f \text{ mk-rec-env css } \Gamma' + + f \text{ env}) \)
by \( \text{blast} \)

next
show \( \neg \text{shadows-consts rhs} \)
using \( \text{rec-comb } \langle \text{not-shadows-vconsts } (Vrecabs css name \Gamma') \rangle \langle (\text{pat}, \text{rhs}) \in \text{set cs} \rangle \)
by \( \text{(auto simp: list-all-iff)} \)
qed

next
\( \text{case } (\text{constr name } \Gamma \ ts us) \)
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 us 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 (⊢ v / - ≈ - [0, 50]) where
constr: list-all2 vrelated ts us ⇒ Vconstr name ts ≈ Vconstr name us |
abs: fmrel-on-fset (frees (Sabs cs)) vrelated Γ1 Γ2 ⇒
fmrel-on-fset (consts (Sabs cs)) vrelated (fmap-of-list rs) Γ2 ⇒
    ⊢v Vabs cs Γ1 ≈ Vabs cs Γ2 |
rec-abs:
    pred-fmap (λcs.
        fmrel-on-fset (frees (Sabs cs)) vrelated Γ1 Γ2 ∧
fmrel-on-fset (consts (Sabs cs)) vrelated (fmap-of-list rs) (Γ2 ++f mk-rec-env
css Γ2)) css ⇒
        name ∈ fmdom css ⇒
        ⊢v Vrecabs css name Γ1 ≈ Vrecabs css name Γ2

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 t1 t2

191
\begin{itemize}
\item \textbf{Assume} \( \Gamma \vdash v \, t_1 \approx t_2 \)
\item \textbf{Thus} veq-structure \( t_1, t_2 \)
\item \textbf{Apply} (\textit{induction} \( t_1 \) arbitrary; \( t_2 \))
\item \textbf{Apply} (\textit{erule vrelated.cases; auto})
\item \textbf{Apply} (\textit{erule list.rel-mono-strong})
\item \textbf{Apply} \textit{simp}
\item \textbf{Apply} (\textit{erule vrelated.cases; auto})
\item \textbf{Apply} (\textit{erule vrelated.cases; auto})
\item \textbf{Done}
\end{itemize}
\begin{itemize}
\item \textbf{Next}
\item \textbf{Fix} \( \text{name name'} \) and \( ts \) \text{ us} :: \text{value list}
\item \textbf{Show} \( \Gamma \vdash v \, \text{Vconstr name ts} \approx \text{Vconstr name'} \{text{ us}} \leftrightarrow (\text{name} = \text{name'} \land \text{list-all2} \text{ vrelated ts us}) \)
\item \textbf{Proof} \text{ safe}
\item \textbf{Assume} \( \Gamma \vdash v \, \text{Vconstr name ts} \approx \text{Vconstr name'} \text{ us} \)
\item \textbf{Thus} \text{name} = \text{name'} \land \text{list-all2} \text{ vrelated ts us}
\item \textbf{By} (\text{cases; auto}+)
\item \textbf{Qed} (\textit{auto intro: vrelated.intro})
\end{itemize}
\textit{Qed}

The technically involved relation \text{vrelated} implies a weaker, but more intuitive property: If \( \Gamma \vdash v \, t \approx u \) then \( t \) and \( u \) are equal after termification (i.e. conversion with \text{value-to-sterm}). In fact, if both terms are ground terms, it collapses to equality. This follows directly from the interpretation of \text{value-struct-rel}.

\textbf{Lemma} \textit{veval'-correct}:
\begin{itemize}
\item \textbf{Assumes} \( \Gamma_2 \vdash v \, t \downarrow v_2 \, \text{wellformed} \, t \, \text{wellformed-venv} \, \Gamma_2 \)
\item \textbf{Assumes} \( \sim \text{shadows-consts} \, t \, \text{not-shadows-vconsts-env} \, \Gamma_2 \)
\item \textbf{Assumes} \( \text{welldefined} \, t \)
\item \textbf{Assumes} \( \text{fmpred} \, (\lambda -, \text{vwelldefined}) \, \Gamma_1 \)
\item \textbf{Assumes} \( \text{fnrel-on-fset} \, (\text{frees} \, t) \, \text{vrelated} \, \Gamma_1 \Gamma_2 \)
\item \textbf{Assumes} \( \text{fnrel-on-fset} \, (\text{consts} \, t) \, \text{vrelated} \, (\text{fmap-of-list} \, rs) \, \Gamma_2 \)
\item \textbf{Obtains} \( v_1 \) \text{ where} \( rs, \Gamma_1 \vdash v \, t \downarrow v_1 \, \Gamma_1 \, v_1 \approx v_2 \)
\item \textbf{Apply} \textit{atomize-elim}
\item \textbf{Using} \textit{assms proof} (\textit{induction arbitrary:} \( \Gamma_1 \))
\item \textbf{Case} (\textit{const name} \( \Gamma_2 \, \text{val}_2 \))
\item \textbf{Hence} \text{fnrel-on-fset} \, \{\text{name}\} \, \text{vrelated} \, (\text{fmap-of-list} \, rs) \, \Gamma_2 \)
\item \textbf{By} \textit{simp}
\item \textbf{Have} \text{rel-option} \, \text{vrelated} \, (\text{fnlookup} \, (\text{fmap-of-list} \, rs) \, \text{name}) \, (\text{fnlookup} \, \Gamma_2 \, \text{name})
\item \textbf{Apply} (\textit{rule fnrel-on-fsetD[where} \, S = \{\text{name}\}])
\item \textbf{Apply} \textit{simp}
\item \textbf{Apply} \textit{fact}
\item \textbf{Done}
\end{itemize}
\begin{itemize}
\item \textbf{Then obtain} \text{val}_1 \, \text{where} \text{fnlookup} \, (\text{fmap-of-list} \, rs) \, \text{name} = \text{Some} \, \text{val}_1 \, \Gamma_1 \, \text{val}_1 \approx \text{val}_2 \)
\item \textbf{Using} \textit{const by} \textit{cases auto}
\item \textbf{Hence} (\text{name}, \text{val}_1) \in \text{set} \, rs
\item \textbf{By} (\textit{auto dest: fmap-of-list-SomeD})
\end{itemize}
show \( ?\text{case} \)
apply (intro conjI exI)
apply (rule veval.const)
apply fact+
done

next

case (var \( \Gamma_2 \) name \( \text{val}_2 \))
hence \( \text{fmrel-on-fset} \{\{\text{name}\}\} \) vrelated \( \Gamma_1 \) \( \Gamma_2 \)
by simp
have rel-option vrelated (fmlookup \( \Gamma_1 \) name) (fmlookup \( \Gamma_2 \) name)
apply (rule fmrel-on-fsetD[where \( S = \{\{\text{name}\}\}\) ])
apply simp
apply fact
done

then obtain \( \text{val}_1 \) where fmlookup \( \Gamma_1 \) name = \( \text{Some} \) \( \text{val}_1 \) \( \vdash \text{val}_1 \) \( \approx \) \( \text{val}_2 \)
using var by cases auto

show \( ?\text{case} \)
apply (intro conjI exI)
apply (rule veval.var)
apply fact+
done

next

case abs
thus \( ?\text{case} \)
by (auto intro!: veval.abs vrelated.abs)

next

case (comb \( \Gamma_2 \) \( t \) cs \( \Gamma_2' \) \( u \) \( u_2' \) \( \text{env}_2 \) \( \text{rhs} \) \( \text{val}_2 \))
hence \( \exists v. \text{rs}, \Gamma_1 \vdash v \downarrow v \wedge \vdash v \approx \text{Vabs} \) cs \( \Gamma_2' \)
by (auto intro: fmrel-on-fsubset)
then obtain \( v \) where \( \vdash v \approx \text{Vabs} \) cs \( \Gamma_2' \) \( \text{rs}, \Gamma_1 \vdash v \downarrow v \)
by blast
moreover then obtain \( \Gamma_1' \)
where \( v = \text{Vabs} \) cs \( \Gamma_1' \)
and \( \text{fmrel-on-fset} \) (frees \( (\text{Sabs} \) cs)) vrelated \( \Gamma_1' \) \( \Gamma_2' \)
and \( \text{fmrel-on-fset} \) (consts \( (\text{Sabs} \) cs)) vrelated \( (\text{fmap-of-list} \) \( \text{rs} \) \) \( \Gamma_2' \)
by cases auto
ultimately have \( \text{rs}, \Gamma_1 \vdash v \downarrow \text{Vabs} \) cs \( \Gamma_1' \)
by simp

have \( \exists u_1'. \text{rs}, \Gamma_1 \vdash u \downarrow u_1' \wedge \vdash u_1' \approx u_2' \)
using comb by (auto intro: fmrel-on-fsubset)
then obtain \( u_1' \) where \( \vdash u_1' \approx u_2' \) \( \text{rs}, \Gamma_1 \vdash u \downarrow u_1' \)
by blast

have rel-option (rel-prod \( (\text{fmrel} \) vrelated) \( (=) \) \( (\text{vfind-match} \) cs \( u_1' \) \) \) \( (\text{vfind-match} \) cs \( u_2') \))
by (rule vrelated.vfind-match-rel') fact
then obtain \( \text{env}_1 \) where \( \text{vfind-match} \) cs \( u_1' = \text{Some} \) \( (\text{env}_1, \text{pat}, \text{rhs}) \) \( \text{fmrel} \) vrelated \( \text{env}_1 \) \( \text{env}_2 \)
using \langle v\text{find-match} \ cs \ u' \ 2 = \ \rangle
by \ cases \ auto

have \ (pat, \ rhs) \in \ set \ cs
by \ (rule \ v\text{find-match-elem}) \ fact

have \ v\text{wellformed} \ (Vabs \ cs \ \Gamma' \ 2)
apply \ (rule \ v\text{eval'\-wellformed})
apply \ fact
using \ \langle \text{wellformed} \ (t \ \$ \ u) \ \rangle \ apply \ \text{simp}
apply \ fact+
done
hence \ v\text{wellformed\-env} \ \Gamma' \ 2
by \ \text{simp}

have \ v\text{welldefined} \ v
apply \ (rule \ v\text{eval-welldefined})
apply \ fact+
using \ \text{comb} \ by \ \text{simp}
hence \ v\text{welldefined} \ (Vabs \ cs \ \Gamma' \ 1)
unfolding \ \langle \ v = \ \rangle .

have \ \text{linear} \ pat
using \ \langle \ (pat, \ rhs) \in \ set \ cs \ \rangle \ (v\text{wellformed} \ (Vabs \ cs \ \Gamma' \ 2))
by \ (auto \ simp: \ list-all-iff)

have \ f\text{dom} \ env_1 = \ \text{patvars} \ (mk\text{-pat} \ pat)
apply \ (rule \ v\text{match-dom})
apply \ (rule \ v\text{find-match-elem})
apply \ fact
done
with \ \langle \text{linear} \ pat \rangle \ have \ f\text{dom} \ env_1 = \ \text{frees} \ pat
by \ (simp \ add: \ mk\text{-pat-frees})

have \ f\text{dom} \ env_2 = \ \text{patvars} \ (mk\text{-pat} \ pat)
apply \ (rule \ v\text{match-dom})
apply \ (rule \ v\text{find-match-elem})
apply \ fact
done
with \ \langle \text{linear} \ pat \rangle \ have \ f\text{dom} \ env_2 = \ \text{frees} \ pat
by \ (simp \ add: \ mk\text{-pat-frees})

note \ f\text{set-of-list-map}[simp \ del]
have \ \exists \ \text{val}_1, \ \text{rs}, \ \Gamma' \ 1 + + f \ \text{env}_1 \ \vdash _v \ \text{rhs} \ \downarrow \ \text{val}_1 \ \land \ \vdash _v \ \text{val}_1 \ \approx \ \text{val}_2
proof \ (rule \ \text{comb})
show \ f\text{rel-on-fset} \ (\text{frees} \ \text{rhs}) \ v\text{related} \ (\Gamma' \ 1 + + f \ \text{env}_1) \ (\Gamma' \ 2 + + f \ \text{env}_2)
proof
fix \ name
assume \ name \ \in \ \text{frees} \ \text{rhs}
show rel-option vrelated (fmlookup (Γ'1 ++f env1) name) (fmlookup (Γ'2 ++f env2) name)

proof (cases name \in| frees pat)
   case True
      hence name \in| fmdom env1 name \in| fmdom env2
         using (fmdom env1 = frees pat) (fmdom env2 = frees pat)
         by simp+
      hence fmlookup (Γ'1 ++f env1) name = fmlookup env1 name fmlookup (Γ'2 ++f env2) name = fmlookup env2 name
         by auto
      thus ?thesis
         using (fmrel vrelated env1 env2)
         by auto
   next
   case False
      hence name \notin| fmdom env1 name \notin| fmdom env2
         using (fmdom env1 = frees pat) (fmdom env2 = frees pat)
         by simp+
      hence fmlookup (Γ'1 ++f env1) name = fmlookup Γ'1 name fmlookup (Γ'2 ++f env2) name = fmlookup Γ'2 name
         by auto

   moreover have name \in| frees (Sabs cs)
      using False (name \in| frees rhs) ((pat, rhs) \in| set cs)
      by (auto simp: fffUnion-alt-def)
      apply (rule fBexf [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 \neg shadows-consts rhs
      using ((pat, rhs) \in| set cs)
      by (auto simp: list-all-iff)

show not-shadows-vconsts-env (Γ'2 ++f env2)
   apply rule
   using (not-shadows-vconsts (Vabs cs Γ'2); apply simp
   apply (rule not-shadows-vconsts-ematch-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) (\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 name)
    using (fmrel-on-fset (consts (Sabs cs)) vrelated (fmap-of-list rs) \Gamma_2)
    by (auto dest: fmrel-on-fsetD)
moreover have name \notin fmdom env_2
proof
  assume name \in\ fmdom env_2
  hence fmlookup env_2 name \neq None
    by (meson fmdom-notI)
  then obtain v where fmlookup env_2 name = Some v
    by blast
  hence name \in\ fmdom env_2
    by (auto intro: fmdomI)
  hence name \in\ frees pat
    using (fmdom env_2 = frees pat)
    by simp
have welldefined rhs
  using :welldefined (Vabs cs \Gamma_1) ((pat, rhs) \in set cs)
  by (auto simp: list-all-iff)
  hence name \in\ fst | | fset-of-list rs | | C
    using (name \in\ |consts rhs)
    by (auto simp: allconsts-def)
moreover have \neg shadowsconsts pat
  using :not-shadows-vconsts (Vabs cs \Gamma_2) ((pat, rhs) \in set cs)
  by (auto simp: list-all-iff shadowsconsts-def allconsts-def)
ultimately show False
  using :name \in\ frees pat
  unfolding shadowsconsts-def fdisjnt-alt-def allconsts-def
  by auto
qed
ultimately show rel-option vrelated (fmlookup (fmap-of-list rs) name)
  (fmlookup (\Gamma_2 ++ f env_2) name)
  by simp
qed
next
show wellformed rhs
using \((\text{pat}, \text{rhs}) \in \text{set cs}) \langle \text{vwellformed (Vabs cs } \Gamma_2')\rangle\)
by \(\text{(auto simp: list-all-iff)}\)

next
have \(\text{wellformed-venv } \Gamma_2'\)
by \(\text{fact}\)
moreover have \(\text{wellformed-venv } \text{env}_2\)
apply \(\text{(rule vwellformed.vmatch-env)}\)
apply \(\text{(rule vfind-match-elem)}\)
apply \(\text{fact}\)
apply \(\text{(rule veval'-wellformed)}\)
apply \(\text{fact}\)
using \(\langle \text{wellformed (t } \text{ s } u)\rangle\)
apply \(\text{simp}\)
done
ultimately show \(\text{wellformed-venv } (\Gamma_2' ++ f \text{ env}_2)\)
by \(\text{blast}\)

next
show \(\text{welldefined } \text{rhs}\)
using \(\langle \text{vwelldefined (Vabs cs } \Gamma_1') \rangle \langle (\text{pat}, \text{rhs}) \in \text{set cs} \rangle\)
by \(\text{(auto simp: list-all-iff)}\)

next
have \(\text{fmpred } (\lambda . \text{vwelldefined}) \Gamma_1'\)
using \(\langle \text{vwelldefined (Vabs cs } \Gamma_1') \rangle\)
by \(\text{simp}\)
moreover have \(\text{fmpred } (\lambda . \text{vwelldefined}) \text{ env}_1\)
apply \(\text{(rule vwelldefined.vmatch-env)}\)
apply \(\text{(rule vfind-match-elem)}\)
apply \(\text{fact}\)
apply \(\text{(rule veval-welldefined)}\)
apply \(\text{fact}\)
using \(\text{comb apply simp}\)
done
ultimately show \(\text{fmpred } (\lambda . \text{vwelldefined}) (\Gamma_1' ++ f \text{ env}_1)\)
by \(\text{blast}\)
qed

then obtain \(\text{val}_1\) where \(\text{rs, } \Gamma_1' ++ f \text{ env}_1 \vdash_v \text{rhs } \Downarrow \text{val}_1 \vdash_v \text{val}_1 \approx \text{val}_2\)
by \(\text{blast}\)

show \(?case\)
apply \(\text{(intro conjI exI)}\)
apply \(\text{(rule veval.comb)}\)
apply \(\text{fact+}\)
done
next
— Almost verbatim copy from \text{comb case}.
case \(\text{(rec-comb } \Gamma_2' t \text{ css name } \Gamma_2' \text{ cs } u \text{ } u'_2 \text{ env}_2 \text{ pat } \text{rhs } \text{val}_2)\)
hence \( \exists v, rs, \Gamma_1 \vdash_v t \downarrow v \wedge \Gamma_1 \vdash v \approx Vrecabs \ css \ name \ \Gamma'_2 \)
by (auto intro: fmrel-on-fsubset)

then obtain \( v \) where \( \Gamma_1 \vdash v \approx Vrecabs \ css \ name \ \Gamma'_2 \)
by blast

moreover then obtain \( \Gamma'_1 \)
where \( v = Vrecabs \ css \ name \ \Gamma'_1 \)
and \( \text{fmrel-on-fset \ (frees (Sabs cs)) \ vrelated} \ \Gamma'_1 \ \Gamma'_2 \)
and \( \text{fmrel-on-fset \ (consts (Sabs cs)) \ vrelated \ (fmap-of-list \ rs) \ (\Gamma'_2 ++ f \ mk-rec-env \ css \ name \ \Gamma'_2)} \)
using \( \text{fmlookup \ css \ name = Some \ cs} \)
by cases auto

ultimately have \( rs, \Gamma_1 \vdash_v t \downarrow Vrecabs \ css \ name \ \Gamma'_1 \)
by simp

have \( \exists u_1', rs, \Gamma_1 \vdash_v u \downarrow u_1' \wedge \Gamma_1 \vdash u_1' \approx u'_2 \)
using rec-comb by (auto intro: fmrel-on-fsubset)
then obtain \( u_1' \) where \( \Gamma_1 \vdash_v u_1' \approx u'_2 \)
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 \( \text{env}_1 \) where \( v\text{find-match cs u'}_1 = \text{Some \ (env}_1, \text{pat, rhs)} \)
fmrel vrelated \( \text{env}_1 \ \text{env}_2 \)
using \( v\text{find-match cs u'}_2 = 0 \)
by cases auto

have \( \text{(pat, rhs)} \in \text{set \ cs} \)
by (rule vfind-match-elem) fact

have \( v\text{wellformed \ (Vrecabs css name \ \Gamma'_2)} \)
apply (rule veval'-wellformed)
apply fact
using (wellformed (t $ s \ u)) apply simp
apply fact+
done
hence wellformed-venv \( \Gamma'_2 \ v\text{wellformed \ (Vabs cs \ \Gamma'_2)} \)
using rec-comb by auto

have \( v\text{welldefined \ v} \)
apply (rule veval-welldefined)
apply fact+
using rec-comb by simp
hence vwelldefined \( \text{(Vrecabs css name \ \Gamma'_1)} \)
unfolding \( v = 0 \)

hence \( v\text{welldefined \ (Vabs cs \ \Gamma'_1)} \)
using rec-comb by auto

have linear \( \text{pat} \)

198
using \((\text{pat}, \text{rhs}) \in \text{set cs}\) \((\text{wellformed (Vabs cs \(\Gamma'_2\))})\):
by \((\text{auto simp: list-all-iff})\)

have \(\text{fmdom env}_1 = \text{patvars} (\text{mk-pat pat})\)
apply \((\text{rule vmatch-dom})\)
apply \((\text{rule vfind-match-elem})\)
apply \(\text{fact}\)
done
with \(\langle \text{linear pat} \rangle\) have \(\text{fmdom env}_1 = \text{frees pat}\)
by \((\text{simp add: mk-pat-frees})\)

have \(\text{fmdom env}_2 = \text{patvars} (\text{mk-pat pat})\)
apply \((\text{rule vmatch-dom})\)
apply \((\text{rule vfind-match-elem})\)
apply \(\text{fact}\)
done
with \(\langle \text{linear pat} \rangle\) have \(\text{fmdom env}_2 = \text{frees pat}\)
by \((\text{simp add: mk-pat-frees})\)

note \(\text{fset-of-list-map\[simp del\]}\)
have \(\exists \text{val}_1, \text{rs}, \Gamma'_1 ++_f \text{env}_1 \vdash_v \text{rhs} \vdash_v \text{val}_1 \wedge \vdash_v \text{val}_1 \approx \text{val}_2\)
proof \((\text{rule rec-comb})\)
have \(\text{not-shadows-vconsts} (\text{Vrecabs css name \(\Gamma'_2\)})\)
apply \((\text{rule veval\^\prime-shadows})\)
apply \(\text{fact}\)
using rec-comb by \(\text{auto}\)
thus \(\neg \text{shadows-consts \(\text{rhs}\)}\)
using \(\langle \text{pat}, \text{rhs} \rangle \in \text{set cs}\) rec-comb
by \((\text{auto simp: list-all-iff})\)
hence \(\text{fdisjnt all-consts (frees \(\text{rhs}\)})}\)
by \((\text{rule shadows-consts-frees})\)

have \(\text{not-shadows-vconsts-env} \Gamma'_2\)
using \(\langle \text{not-shadows-vconsts (Vrecabs css name \(\Gamma'_2\)}\rangle\)
by \(\text{simp}\)

moreover have \(\text{not-shadows-vconsts-env} (\text{mk-rec-env css \(\Gamma'_2\)})\)
unfolding \(\text{mk-rec-env-def}\)
using \(\langle \text{not-shadows-vconsts (Vrecabs css name \(\Gamma'_2\)}\rangle\)
by \((\text{auto intro: fmdomI})\)

moreover have \(\text{not-shadows-vconsts-env env}_2\)
apply \((\text{rule not-shadows-vconsts.vmatch-env})\)
apply \((\text{rule vfind-match-elem})\)
apply \(\text{fact}\)
apply \(\text{rule veval\^\prime-shadows}\)
apply \(\text{fact}\)
apply \(\text{fact}\)
using rec-comb by \(\text{auto}\)
ultimately show \( \text{not-shadows-vconsts-env} (\Gamma_2' ++f \text{mk-rec-env css} \Gamma_2' ++f env_2) \)
by auto

have \( \text{not-shadows-vconsts} (Vabs cs \Gamma_2') \)
using \( \langle \text{not-shadows-vconsts} (Vrecabs - - -) \rangle \) rec-comb
by auto

show \( \text{fmrel-on-fset (frees rhs) vrelated} (\Gamma_1' ++f env_1) (\Gamma_2' ++f \text{mk-rec-env css} \Gamma_2' ++f env_2) \)
proof
fix name
assume name \( \in \) frees rhs
moreover have \( \text{fmdom css} \subseteq \text{all-consts} \)
using \( \langle \text{vwelldefined} (Vrecabs - - -) \rangle \) unfolding all-consts-def
by auto
ultimately have name \( \notin \) fmdom css
using \( \langle \text{fdisjnt - (frees rhs)} \rangle \)
unfolding fdisjnt-alt-def
by \( \langle \text{metis (full-types) fempty-iff finterI fset-rev-mp} \rangle \)

show \( \text{rel-option vrelated} (\text{fmlookup} (\Gamma_1' ++f env_1) \text{name}) (\text{fmlookup} (\Gamma_2' ++f \text{mk-rec-env css} \Gamma_2' ++f env_2) \text{name}) \)
proof (cases name \( \in \) frees pat)
case True
hence name \( \in \) fmdom env_1 name \( \in \) fmdom env_2
using \( \langle \text{fmdom env_1 = frees pat} \rangle \) \( \langle \text{fmdom env_2 = frees pat} \rangle \)
by simp+
hence \( \text{fmlookup} (\Gamma_1' ++f env_1) \text{name} = \text{fmlookup env_1 name} \)
\( \text{fmlookup} (\Gamma_2' ++f \text{mk-rec-env css} \Gamma_2' ++f env_2) \text{name} = \text{fmlookup env_2 name} \)
by auto
thus \( \theta \)thesis
using \( \langle \text{fmrel vrelated env_1 env_2} \rangle \)
by auto
next
case False
hence name \( \notin \) fmdom env_1 name \( \notin \) fmdom env_2
using \( \langle \text{fmdom env_1 = frees pat} \rangle \) \( \langle \text{fmdom env_2 = frees pat} \rangle \)
by simp+
hence \( \text{fmlookup} (\Gamma_1' ++f env_1) \text{name} = \text{fmlookup} \Gamma_1' \text{name} \)
\( \text{fmlookup} (\Gamma_2' ++f \text{mk-rec-env css} \Gamma_2' ++f env_2) \text{name} = \text{fmlookup} \Gamma_2' \text{name} \)
unfolding mk-rec-env-def using \( \langle \text{name \notin fmdom css} \rangle \)
by auto

200
moreover have \( \text{name} \in \text{frees (Sabs cs)} \)
using \( \text{False} \langle \text{name} \in \text{frees rhs} \rangle \langle \text{pat, rhs} \in \text{set cs} \rangle \)
apply \( \text{(auto simp: fUnion-alt-def)} \)
apply \( \text{(rule fBexI [where x = frees rhs |\ -| frees pat])} \)
apply \( \text{(auto simp: fset-of-list-elem)} \)
done

ultimately show \( \text{?thesis} \)
using \( \langle \text{fmrel-on-fset (frees (Sabs cs)) vrelated \( \Gamma'_1 \) \( \Gamma'_2 \) \rangle \)
by \( \text{(auto dest: fmrel-on-fsetD)} \)
qed

show \( \text{fmrel-on-fset (consts rhs) vrelated (fmap-of-list rs) (\( \Gamma'_2 \) ++f mk-rec-env css \( \Gamma'_2 \) ++f env2)} \)
proof
fix \( \text{name} \)
assume \( \text{name} \in \text{consts rhs} \)
then \( \text{have name \in const}\) cs
using \( \langle \text{(pat, rhs) \in set cs} \rangle \)
by \( \text{(auto intro!: fBexI simp: fset-of-list-elem fUnion-alt-def)} \)

moreover have \( \text{name \not\in \text{fmdom env2} } \)
proof
assume \( \text{name \in \text{fmdom env2} } \)
then \( \text{obtain v where fnlookup env2 name = Some v} \)
by \( \text{blast} \)

have \( \text{vwelldefined (Vabs cs \( \Gamma'_1 \))} \)
using \( \langle \text{vwelldefined (Vrecabs css - \( \Gamma'_1 \))} \rangle \)
using \( \text{rec-comb} \)
by \( \text{auto} \)

hence \( \text{welldefined rhs} \)
using \( \langle \text{(pat, rhs) \in set cs; rec-comb} \rangle \)
by \( \text{(auto simp: list-all-iff)} \)

hence \( \text{welldefined rhs} \)
using \( \langle \text{name \in \text{fset \ ' \ of-list rs \cup C}} \rangle \)
using \( \langle \text{name \in \text{consts rhs; all-consts-def} \rangle} \)

201
by blast
moreover have \( \neg \text{shadows-consts} \text{ pat} \)
using \( \langle \text{not-shadows-vconsts} \ (Vabs \ cs \ \Gamma') \rangle \langle \text{pat, rhs} \rangle \in \text{set cs} \)
by (auto simp: list-all-iff shadows-consts-def all-consts-def)
ultimately show False
using \( \langle \text{name} \mid \in \rangle \text{frees pat} \)
unfolding shadows-consts-def fdisjnt-alt-def all-consts-def
by auto
qed
ultimately show rel-option \( vrelated \) \( \langle \text{fmlookup} \ (\text{fmap-of-list} \ rs) \text{ name} \rangle \)
(\( \text{fmlookup} \ (\Gamma' + + f \ mk-rec-env \ css \ \Gamma' + + f \ \text{env} _2) \text{ name} \))
by simp
qed
next
show wellformed \( \text{rhs} \)
using \( \langle \text{pat, rhs} \rangle \in \text{set cs} \) \( \langle vwellformed \ (Vabs \ cs \ \Gamma') \rangle \)
by (auto simp: list-all-iff)
next
have wellformed-venv \( \Gamma' \)
by fact
moreover have wellformed-venv \( \text{env} _2 \)
apply (rule vwellformed.vmatch-env)
apply (rule vfind-match-elem)
apply fact
apply (rule veval’.wellformed)
apply fact
using \( \langle \text{wellformed} \ (t \ \$ \ u) \rangle \); apply simp
apply fact+
done
moreover have wellformed-venv \( \text{mk-rec-env} \ (\Gamma' + + f \ mk-rec-env \ css \ \Gamma' + + f \ \text{env} _2) \)
by blast
next
show welldefined \( \text{rhs} \)
using \( \langle vwelldefined \ (Vabs \ cs \ \Gamma' _1) \rangle \langle \text{pat, rhs} \rangle \in \text{set cs} \)
by (auto simp: list-all-iff)
next
have fnpred \( (\lambda -. vwelldefined) \) \( \Gamma' _1 \)
using \( \langle vwelldefined \ (Vabs \ cs \ \Gamma' _1) \rangle \)
by simp
moreover have fnpred \( (\lambda -. vwelldefined) \) \( \text{env} _1 \)
apply (rule vwelldefined.vmatch-env)
apply (rule vfind-match-elem)
apply fact
apply (rule veval-welldefined)
apply fact

202
apply fact
using rec-comb apply simp
done

ultimately show \( \text{fmpred} (\lambda\cdot \text{vwelldefined}) (\Gamma_1' +++ f env_1) \)
by blast
qed

then obtain \( \text{val}_1 \) where \( rs, \Gamma_1' +++ f env_1 \vdash _v \text{rhs} \downarrow \text{val}_1 \vdash _v \text{val}_1 \approx \text{val}_2 \)
by blast

show ?\( \text{case} \)
apply (intro conjI exI)
apply (rule veval.rec-comb)
apply fact+
done

next
case (constr name \( \Gamma_2 \) \( ts \) \( us_2 \))

have \( \text{list-all2} (\lambda u_2. (\exists u_1. rs, \Gamma_1 \vdash _v t \downarrow u_1 \land \vdash _v u_1 \approx u_2)) ts \) \( us_2 \)
using \( \langle \text{list-all2} - ts \) \( us_2 \rangle \)
proof (rule list.rel-mono-strong, elim conjE impE allE exE)
fix \( t \) \( u_2 \)
assume \( t \in \text{set ts} \) \( u_2 \in \text{set us}_2 \)
assume \( \Gamma_2 \vdash _v t \downarrow u_2 \)

show \( \text{wellformed} t \) \( \text{welldefined} t \) \( \sim \text{shadows-consts} \) \( t \)
using constr \( \langle t \in \text{set ts} \rangle \)
unfolding \( \text{welldefined.list-comb} \) \( \text{wellformed.list-comb} \) \( \text{shadows.list-comb} \)
by (auto simp: list-all-iff list-ex-iff)

show \( \text{wellformed-venv} \) \( \Gamma_2 \)
\( \text{not-shadows-vconsts-env} \) \( \Gamma_2 \)
\( \text{fmpred} (\lambda\cdot \text{vwelldefined}) \) \( \Gamma_1 \)
by fact+

have \( \text{consts} t \) \( \in \) \( \text{fset-of-list} (\map \text{consts} \) \( ts \) \( ) \)
using \( \langle t \in \text{set ts} \rangle \) by (simp add: fset-of-list-elem)

hence \( \text{consts} t \) \( \subseteq \) \( \text{consts} \) (name \( $$ \) \) \( ts \) \)

unfolding \( \text{consts-list-comb} \)
by (metis \( \text{ffUnion-subset-elem} \) \( \text{le-supI2} \))
thus \( \text{fmrel-on-fset} (\text{consts} \) \( t \) \) \( \text{vrelated} \) (fmap-of-list \( rs \) \) \( \Gamma_2 \)
using constr by (blast intro: fmrel-on-fsubset)

have \( \text{frees} t \) \( \in \) \( \text{fset-of-list} (\map \text{frees} \) \( ts \) \( ) \)
using \( \langle t \in \text{set ts} \rangle \) by (simp add: fset-of-list-elem)

hence \( \text{frees} t \) \( \subseteq \) \( \text{frees} \) (name \( $$ \) \) \( ts \) \)

unfolding \( \text{frees-list-comb} \) \( \text{const-stern-def} \) \( \text{freess-def} \)
by (auto intro!: ffUnion-subset-elem)

thus fmrel-on-fset (frees t) vrelated Γ_1 Γ_2
  using constr by (blast intro: fmrel-on-fsubset)
qed auto

then obtain us_1 where list-all2 (veval rs Γ_1) ts us_1 list-all2 vrelated us_1 us_2
by induction auto

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

lemma veval'-correct':
assumes Γ_2 ⊢ v t ↓ v fmrel-on-fset (consts t) vrelated (fmap-of-list rs) Γ_2
assumes closed t
assumes fmrel-on-fset (consts t) vrelated fmempty Γ_2
obtains v_1 where rs, fmempty ⊢ v_1 ⊢ v_1 ≈ v_2
proof (rule veval'-correct[where Γ_1 = fmempty])
  show fmpred (λ· vwelldefined) fmempty by simp
next
  show fmrel-on-fset (frees t) vrelated fmempty Γ_2
    using (closed t) unfolding closed-except-def by auto
qed (rule assms)+

end

Preservation of extensional equality

lemma (in constants) veval'-agree-eq:
assumes Γ ⊢ v t ↓ v fmrel-on-fset (ids t) vrelated Γ' Γ
assumes closed-venv Γ closed-except t (fmdom Γ)
assumes wellformed t wellformed-venv Γ fdisjnt C (fmdom Γ)
assumes consts t |⊆| fmdom Γ |∪| C fmpred (λ· vwelldefined') Γ
assumes ¬ shadows-consts t not-shadows-vconsts-env Γ
obtains v' where Γ' ⊢ v t ↓ v' v' ≈_e v
using assms proof (induction arbitrary: Γ' thesis)
case (const name Γ val)
hence name |∈| ids (Sconst name)
  unfolding ids-def by simp
with const have rel-option vrelated (fmlookup Γ' name) (fmlookup Γ name)
  by (auto dest: fmrel-on-fsetD)
then obtain val' where fmlookup Γ' name = Some val' val' ≈_e val
  using (fmlookup Γ name = Some val)
  by cases auto
thus case
  using const by (auto intro: veval'.const)

204
next
case (var \( \Gamma \) name val)
  hence name \( \in \) ids (Svar name)
  unfolding ids-def by simp
with var have rel-option erelated (flookup \( \Gamma' \) name) (flookup \( \Gamma' \) name)
  by (auto dest: fmrel-on-fsetD)
then obtain val' where flookup \( \Gamma' \) name = Some val' val' \( \approx_\varepsilon \) val
  using (flookup \( \Gamma' \) name = Some val)
  by cases auto
thus \?case
  using var by (auto intro: veval'.var)
next
case (abs \( \Gamma \) cs)
  hence Vabs cs \( \Gamma' \) \( \approx_\varepsilon \) Vabs cs \( \Gamma \)
  by (auto intro: erelated.abs)
  thus \?case
  using abs by (auto intro: veval'.abs)
next
case (comb \( \Gamma \) t cs \( \Gamma_\Lambda \) u v 2 env pat rhs val)
  have fmrel-on-fset (ids t) erelated \( \Gamma' \) \( \Gamma \)
    apply (rule fmrel-on-fsubset)
    apply fact
    unfolding ids-def by auto
  then obtain \( v_1' \) where \( \Gamma' \downarrow_\varepsilon t \downarrow_\varepsilon v_1' \approx_\varepsilon \) Vabs cs \( \Gamma_\Lambda \)
    using comb by (auto simp: closed-except-def)
  then obtain \( \Gamma_\Lambda' \) where \( v_1' = \) Vabs cs \( \Gamma_\Lambda' \) fmrel-on-fset (ids (Sabs cs)) erelated \( \Gamma_\Lambda' \Gamma_\Lambda \)
    by (auto elim: erelated.cases)
  have fmrel-on-fset (ids u) erelated \( \Gamma' \) \( \Gamma \)
    apply (rule fmrel-on-fsubset)
    apply fact
    unfolding ids-def by auto
  then obtain \( v_2' \) where \( \Gamma' \downarrow_\varepsilon u \downarrow_\varepsilon v_2' \approx_\varepsilon v_2 \)
    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 \( v_2' \approx_\varepsilon v_2 \) 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 \( \Gamma_\Lambda \))
    apply (rule veval'-closed)
    using comb by (auto simp: closed-except-def)
have \( \text{vclosed } v_2 \)
  apply (rule \text{veval'}-closed)
  using \text{comb} by (auto simp: closed-except-def)

have \( \text{closed-except } (Sabs \ cs) (\text{fmdom } \Gamma_\Lambda) \)
  using \( \text{vclosed } (\text{Vabs } cs \ \Gamma_\Lambda) \) by (auto simp: \text{Sterm} closed-except-simps)

have \( \text{fmdom } \Gamma_\Lambda \subseteq \text{fmdom } \Gamma_\Lambda \)
  unfolding closed-except-def.

have \( \text{vwellformed } (\text{Vabs } cs \ \Gamma_\Lambda) \)
  apply (rule \text{veval'}-wellformed)
  apply fact
  using \text{comb} by auto

have \((\text{pat}, \text{rhs}) \in \text{set } cs\)
  by (rule \text{vfind-match-elem} fact)

have \( \text{fmdom } \text{env} = \text{frees } \text{pat} \)
  apply simp
  apply (rule \text{vmatch-dom})
  apply (rule \text{vfind-match-elem})
  apply (rule \text{comb})
  done

have \( \text{vwelldefined'} (\text{Vabs } cs \ \Gamma_\Lambda) \)
  apply (rule \text{veval'}-welldefined')
  apply fact
  using \text{comb} by auto

have \( \text{not-shadows-vconsts } (\text{Vabs } cs \ \Gamma_\Lambda) \)
  apply (rule \text{veval'}-shadows)
  using \text{comb} by auto

obtain \( \text{val'} \) where \( \Gamma_\Lambda' ++_f \text{env'} \vdash_v \text{rhs} \downarrow \text{val'} \approx_v \text{val} \)
proof (erule \text{comb.IH})
  show \( \text{closed-venv } (\Gamma_\Lambda' ++_f \text{env}) \)
    apply rule
    using \( \text{vclosed } (\text{Vabs } cs \ \Gamma_\Lambda) \) apply simp
    apply (rule \text{vclosed} \text{.vmatch-env})
    apply (rule \text{vfind-match-elem})
    apply fact
    apply fact
done
next
show wellformed rhs
  using (\langle pat, rhs \rangle \in set cs \rangle \langle \text{wellformed} (Vabs cs \Gamma) \rangle)
  by (auto simp: list-all-iff)
next
show wellformed-venv (\Gamma \Lambda ++ f env)
  apply rule
  using (\langle \text{wellformed} (Vabs cs \Gamma) \rangle \langle \text{simp} \rangle)
  apply (rule vwellformed.envmatch-env)
  apply (rule \text{vfind-match-elem})
  apply (rule comb)
  apply (rule \text{veval'}-wellformed)
  apply fact
  using comb by auto
next
have \text{fmdom} env = \text{fmdom} env'
  using \langle (\langle pat, rhs \rangle \in set cs \rangle \langle \text{closed} (Vabs cs \Gamma) \rangle \langle \text{fmdom} env = \text{frees pat} \rangle \rangle
  by (auto simp: list-all-iff)
next
show \text{fmrel-on-fset} (ids rhs) \text{erelated} (\Gamma \Lambda' ++ f env') (\Gamma \Lambda ++ f env)
  proof
    fix id
    assume id \in| ids rhs
    thus rel-option \text{erelated} (\text{fmlookup} (\Gamma \Lambda' ++ f env') id) (\text{fmlookup} (\Gamma \Lambda ++ f env) id)
    unfolding ids-def
    proof (cases rule: funion-strictE)
      case A
      hence id \in| \text{fmdom} env |\cup| \text{fmdom} \Gamma
        using \langle \text{closed-except rhs} (\text{fmdom} (\Gamma \Lambda ++ f env)) \rangle
        unfolding closed-except-def
        by auto
      thus \?thesis
        proof (cases rule: funion-strictE)
          case A
          hence id \in| \text{fmdom} env'
            using \langle \text{fmdom} env = \text{frees pat} \rangle \langle \text{fmdom} env = \text{fmdom} env' \rangle
            by simp
          with A show \?thesis
            using \langle \text{fmrel erelated} env' env \rangle by auto
        next
        case B
      next
      case B
    next
    case B
hence $id \mid \notin \ fmdom \ env$ using \( fmdom \ env = \ fmdom \ env' \) by simp

hence $id \mid \in \ fmdom (Sabs \ cs)$
   apply auto
   unfolding \( \text{ffUnion-alt-def} \)
   apply simp
   apply (rule \( \text{fBexI[where} \ x = (\text{pat}, \ rhs)\])\)
   using \( \text{id} \mid \in \ \text{frees} \ \text{rhs}\)
   apply simp
   unfolding \( \text{fset-of-list-elem} \)
   apply (rule \( \text{(pat, rhs)} \in \ \text{set} \ cs\))
   done

hence $id \mid \in \ \text{ids} (Sabs \ cs)$
   unfolding \( \text{ids-def} \)
   by simp

have \( id \mid \notin \ \text{fmdom} \ env' \)
   using \( B \)
   unfolding \( \text{fmdom} \ env = \ fmdom \ env' \) by simp

thus \(? thesis \)
   using \( id \mid \notin \ \text{fmdom} \ env' \)
   apply simp
   apply (rule \( \text{fmrel-on-fsetD} \))
   apply (rule \( \text{id} \mid \in \ \text{ids} (Sabs \ cs)\))
   done

qed

next

case \( B \)

have \( id \mid \in \ \text{consts} \ (Sabs \ cs) \)
   apply auto
   unfolding \( \text{ffUnion-alt-def} \)
   apply simp
   apply (rule \( \text{fBexI[where} \ x = (\text{pat}, \ rhs)\])\)
   apply simp
   apply fact
   unfolding \( \text{fset-of-list-elem} \)
   apply (rule \( \text{(pat, rhs)} \in \ \text{set} \ cs\))
   done

hence \( \text{id} \mid \in \ \text{ids} \ (Sabs \ cs) \)
   unfolding \( \text{ids-def} \)
   by auto

show \(? thesis \)
   using \( \text{fmdom} \ env = \ fmdom \ env' \)
   apply auto
   apply (rule \( \text{fmrelD} \))
   apply (rule \( \text{fmrel erelated env' env} \))
   apply (rule \( \text{fmrel-on-fsetD} \))
   apply (rule \( \text{id} \mid \in \ \text{ids} \ (Sabs \ cs)\))
   apply (rule \( \text{fmrel-on-fset \ (ids} (Sabs \ cs)) \text{ erelated} \ \Gamma_{\Lambda'} \Gamma_{\Lambda'} \))
   done

qed

208
qed
next
show fmpred ($\lambda$. vwelldefined') ($\Gamma_\Lambda$ ++f env)
proof
  have vwelldefined' (Vabs cs $\Gamma_\Lambda$)
    apply (rule veval'-welldefined')
    apply fact
    using comb by auto
  thus fmpred ($\lambda$. vwelldefined') $\Gamma_\Lambda$
    by simp
next
  have vwelldefined' $v_2$
    apply (rule veval'-welldefined')
    apply fact
    using comb by auto

show fmpred ($\lambda$. vwelldefined') env
apply (rule vmatch-welldefined')
apply (rule vfind-match-elem)
apply fact
done
qed

next
have fdisjnt C (fmdom $\Gamma_\Lambda$)
  using ⟨vwelldefined' (Vabs cs $\Gamma_\Lambda$)⟩ by simp
moreover have fdisjnt C (fmdom env)
  unfolding fdisjnt-alt-def by auto
ultimately show fdisjnt C (fmdom ($\Gamma_\Lambda$ ++f env))
  unfolding fdisjnt-alt-def by auto
next
show $\neg$ shadows-consts rhs
  using ⟨(pat, rhs) ∈ set cs⟩ ⟨not-shadows-vconsts (Vabs cs $\Gamma_\Lambda$)⟩
  by (auto simp: list-all-iff)

next
have not-shadows-vconsts-env $\Gamma_\Lambda$
  using ⟨not-shadows-vconsts (Vabs cs $\Gamma_\Lambda$)⟩ by auto
moreover have not-shadows-vconsts-env env
  apply (rule not-shadows-vconsts.vmatch-env)
  apply (rule vfind-match-elem)
  apply fact
  apply (rule veval'.shadows)
  using comb by auto
ultimately show not-shadows-vconsts-env ($\Gamma_\Lambda$ ++f env)
  by blast
next
show consts rhs $|$⊆|$ fmdom ($\Gamma_\Lambda$ ++f env) $|$∪| C
  using ⟨consts rhs $|$⊆|$ ⟩
by auto

qed

moreover have \( \Gamma \vdash v \, t \, s \subseteq u \downarrow \text{val}' \)
proof (rule veval'.comb)
  show \( \Gamma \vdash t \, Vabs \, cs \, \Gamma \)'
  using \( \Gamma \vdash t \downarrow v_1' \)
  unfolding \( \langle v_1' = : \rangle \).
qed fact+

ultimately show \( ?\text{case} \)
using comb by metis

next

  case (rec-comb \( \Gamma \) \( t \) \( cs \) \( name \) \( \Gamma \) \( \Lambda \) cs u v_2 env \( pat \) \( rhs \) \text{val})

  have \( \text{fmrel-on-fset} (\text{ids} \, t) \, \text{erelated} \, \Gamma \, \Gamma \)'
  apply (rule \( \text{fmrel-on-fsubset} \))
  apply fact
  unfolding \text{ids-def} by auto
  then obtain \( v_1' \) where \( \Gamma \vdash t \downarrow v_1' \, v_1' \approx_e v_2 \)
  using rec-comb by (auto simp: closed-except-def)
  then obtain \( \Gamma \)\( \Lambda \)'\)
    where \( v_1' = Vrecabs \, cs \, \name \, \Gamma \)\( \Lambda \)
    and \( \text{pred-fmap} (\lambda \text{cs} \, \text{fmrel-on-fset} (\text{ids} \, (\text{Sabs} \, \text{cs})) \, \text{erelated} \, \Gamma \)\( \Lambda \)' \( \Gamma \)\( \Lambda \)) cs
    by (auto elim: \text{erelated.cases})

  have \( \text{fmrel-on-fset} (\text{ids} \, u) \, \text{erelated} \, \Gamma \, \Gamma \)'
  apply (rule \( \text{fmrel-on-fsubset} \))
  apply fact
  unfolding \text{ids-def} by auto
  then obtain \( v_2' \) where \( \Gamma \vdash u \downarrow v_2' \, v_2' \approx_e v_2 \)
  apply –
  apply (erule rec-comb.IH(2))
  using rec-comb by (auto simp: closed-except-def)

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

  have \( \text{vclosed} \, (Vrecabs \, cs \, \name \, \Gamma \)\( \Lambda \))
  apply (rule \( \text{veval'}-\text{closed} \))
  using rec-comb by (auto simp: closed-except-def)
  hence \( \text{vclosed} \, (Vabs \, cs \, \Gamma \)\( \Lambda \))
  using rec-comb by (auto simp: closed-except-def)
  have \( \text{vclosed} \, v_2 \)
  apply (rule \( \text{veval'}-\text{closed} \))
using rec-comb by (auto simp: closed-except-def)

have closed-except (Sabs cs) \( f\text{ndom} \; \Gamma_\Lambda \)
using (\text{closed} (Vabs cs \; \Gamma_\Lambda)) by (auto simp: Stem.closed-except-simps)
hence frees (Sabs cs) \( \subseteq \) \( f\text{ndom} \; \Gamma_\Lambda \)

unfolding closed-except-def .

have \( v\text{wellformed} \) (Vrecabs css name \( \Gamma_\Lambda \))
apply (rule veval'\text{-wellformed})
apply fact
using rec-comb by auto

hence \( v\text{wellformed} \) (Vabs cs \( \Gamma_\Lambda \))
using rec-comb by (auto simp: closed-except-def)

have \((\text{pat}, \text{rhs}) \in \text{set} \; \text{cs})\)
by (rule vfind-match-elem) fact

hence linear \( \text{pat} \)
using (Vwellformed (Vabs cs \( \Gamma_\Lambda \)))
by (auto simp: list-all-iff)

hence frees \( \text{pat} = \text{patvars} \) (mk-pat \( \text{pat} \))
by (simp add: mk-pat-frees)

hence \( \text{fndom} \; \text{env} = \text{frees} \; \text{pat} \)
apply simp
apply (rule vmatch-dom)
apply (rule vfind-match-elem)
apply (rule rec-comb)
done

have \( v\text{welldefined}' \) (Vrecabs css name \( \Gamma_\Lambda \))
apply (rule veval'\text{-welldefined}')
apply fact
using rec-comb by auto

hence \( \text{consts} \; \text{rhs} \subseteq \text{fndom} \; \Gamma_\Lambda \cup (C \cup \text{fndom} \; \text{css}) \) \( \text{fdisjnt} \; C \) (\( \text{fndom} \; \Gamma_\Lambda \))
using (\text{pat}, \text{rhs}) \in \text{set} \; \text{cs}) (\text{fmlookup} \; \text{css} \; \text{name} = \text{Some} \; \text{cs})
by (auto simp: list-all-iff dest: fmpredD[where \( m = \text{css} \)])

have not-shadows-vconsts (Vrecabs css name \( \Gamma_\Lambda \))
apply (rule veval'\text{-shadows})
using rec-comb by auto

hence not-shadows-vconsts (Vabs cs \( \Gamma_\Lambda \))
using rec-comb by auto

obtain \text{val}' \; \text{where} \; \Gamma_\Lambda' ++ \_ \; \text{mk-rec-env} \; \text{css} \; \Gamma_\Lambda' ++ \_ \; \text{env}' \; \vdash \; \text{rhs} \; \downarrow \; \text{val}' \; \text{val}' \approx_e \; \text{val}

proof (erule rec-comb.IH)
show \text{closed-venv} (\Gamma_\Lambda ++ \_ \; \text{mk-rec-env} \; \text{css} \; \Gamma_\Lambda ++ \_ \; \text{env})
apply rule
apply rule
using (\text{vclosed} (Vabs cs \; \Gamma_\Lambda)) apply simp

211
unfolding mk-rec-env-def

using (\closed (Vrecabs css name \Gamma_\Lambda) )
apply (auto intro: fmdomI) 
apply (rule vclosed.vmatch-env)
apply (rule vclosed.vmatch-env)
apply fact
apply fact
done

next
show wellformed rhs
using ((\pat, rhs) \in \set cs) (vwellformed (Vabs cs \Gamma_\Lambda))
by (auto simp: list-all-iff)

next
show wellformed-eenv (\Gamma_\Lambda ++f mk-rec-env css \Gamma_\Lambda ++f env)
apply rule
apply rule
using (vwellformed (Vabs cs \Gamma_\Lambda))
apply simp
unfolding mk-rec-env-def
using (vwellformed (Vrecabs css name \Gamma_\Lambda))
apply (auto intro: fmdomI) 
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 ++f env))
using ((\pat, rhs) \in \set cs) (\closed (Vabs cs \Gamma_\Lambda): fmdom env = "frees \pat"
by (auto simp: list-all-iff closed-except-def)
thus closed-except rhs (fmdom (\Gamma_\Lambda ++f mk-rec-env css \Gamma_\Lambda ++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 \Gamma_\Lambda)
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 (\\lambda cs. fmrel-on-fset (ids (Sabs cs)) erelated \Gamma_\Lambda' \Gamma_\Lambda) css))
done

have fmrel-on-fset (ids (Sabs cs)) erelated \Gamma_\Lambda' \Gamma_\Lambda
using \pred-fmap (\\lambda cs. fmrel-on-fset (ids (Sabs cs)) erelated \Gamma_\Lambda' \Gamma_\Lambda) css)
rec-comb

by auto

have \( \text{fmdom} (\text{mk-rec-env } css \ \Gamma \Lambda) = \text{fmdom} (\text{mk-rec-env } css \ \Gamma \Lambda') \)

unfolding \( \text{mk-rec-env-def} \) by auto

show \( \text{fmrel-on-fset} (\text{ids } rhs) \) erelated \( (\Gamma \Lambda' + + f \ \text{mk-rec-env } css \ \Gamma \Lambda' + + f \ env') \)

(\( \Gamma \Lambda + + f \ \text{mk-rec-env } css \ \Gamma \Lambda + + f \ env \))

proof

fix id

assume id \( \in \) ids rhs

thus \( \text{rel-option erelated} (\text{fmlookup} (\Gamma \Lambda' + + f \ \text{mk-rec-env } css \ \Gamma \Lambda' + + f \ env') id) \)

(\( \text{fmlookup} (\Gamma \Lambda + + f \ \text{mk-rec-env } css \ \Gamma \Lambda + + f \ env) \) id)

unfolding ids-def

proof (cases rule: funion-strictE)

case A

hence id \( \in \) \( \text{fmdom} \) env \( \dot{\cup} \) \( \text{fmdom} \) \( \Gamma \Lambda \)

using \( \dot{\text{closed-except rhs}} \) \( (\text{fmdom} (\Gamma \Lambda + + f \ env)) \)

unfolding closed-except-def

by auto

thus \( \text{?thesis} \)

proof (cases rule: funion-strictE)

case A

hence id \( \in \) \( \text{fmdom} \) env'

using \( \text{?fmdom env = frees pat : fmdom env = fmdom env'} \) by simp

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

using \( \text{fmrel erelated env'} env \) by auto

next

case B

hence id \( \notin \) \( \text{frees pat} \)

using \( \text{fmdom env = frees pat} \) by simp

hence id \( \in \) \( \text{frees} \) \( (\text{Sabs cs}) \)

apply auto

unfolding fUnion-altn-def

apply simp

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

using id \( \in \) \( \text{frees rhs} \) apply simp

unfolding fset-of-list-elem

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

done

hence id \( \in \) \( \text{ids} \) \( (\text{Sabs cs}) \)

unfolding ids-def by simp

have id \( \notin \) \( \text{fmdom env'} \)

using B unfolding \( \text{fmdom env = fmdom env'} \) by simp

thus \( \text{?thesis} \)

213
using \( \langle \text{id} \mid \notin \rangle \text{fmdom env} \) \( \text{fmdom} (\text{mk-rec-env css } \Gamma) = \text{fmdom} \)

\[ (\text{mk-rec-env css } \Gamma') \]

apply auto
apply (rule fmrel-on-fsetD)
apply (rule \( \langle \text{id} \mid \notin \rangle \text{ids rhs} \))
apply (rule \( \langle \text{fmrel-on-fset} (\text{ids rhs}) \rangle \text{erelated} (\text{mk-rec-env css } \Gamma') \))
apply (rule fmrel-on-fsetD)
apply (rule \( \langle \text{id} \mid \notin \rangle \text{ids} (Sabs cs) \))
done

qed

next

case \( B \)

have \( \text{id} \mid \notin \) \( \text{consts} \) (Sabs cs)
apply auto
unfolding ffUnion-alt-def
apply simp
apply (rule fBexI)
apply simp
apply fact
unfolding fset-of-list-elem
apply (rule \( \langle \text{id} \mid \notin \rangle \text{ids} (Sabs cs) \))
done
hence \( \text{id} \mid \notin \) \( \text{ids} \) (Sabs cs)
unfolding ids-def by auto

show \(?\text{thesis}\)
using \( \langle \text{fmdom env} = \text{fmdom env}' \rangle \text{fmdom} (\text{mk-rec-env css } \Gamma) = \text{fmdom} \)

\[ (\text{mk-rec-env css } \Gamma') \]

apply auto
apply (rule fmrelD)
apply (rule \( \langle \text{fmrel erelated env'} env \rangle \))
apply (rule fmrel-on-fsetD)
apply (rule \( \langle \text{id} \mid \notin \rangle \text{ids rhs} \))
apply (rule \( \langle \text{fmrel-on-fset} (\text{ids rhs}) \rangle \text{erelated} (\text{mk-rec-env css } \Gamma') \))
apply (rule fmrelD)
apply (rule \( \langle \text{fmrel erelated env'} env \rangle \))
apply (rule fmrel-on-fsetD)
apply (rule \( \langle \text{id} \mid \notin \rangle \text{ids} (Sabs cs) \))
apply (rule \( \langle \text{fmrel-on-fset} (\text{ids} (Sabs cs)) \rangle \text{erelated} \Gamma' \Gamma \))
done

qed

next

show \( \text{fmpred} (\lambda-. \text{vwelldefined'}) (\Gamma_\Lambda + + f \text{mk-rec-env css } \Gamma \text{+} + f \text{env}) \)
proof (intro fmpred-add)
have \text{vwelldefined'} (Vrecabs css name \( \Gamma_\Lambda \))
apply (rule veval'-welldefined')
apply fact
using rec-comb by auto
thus \( \text{fmpred} (\lambda-. \text{vwelldefined}') \Gamma \) \( \text{fmpred} (\lambda-. \text{vwelldefined}') \) (mk-rec-env css \( \Gamma \))

unfolding mk-rec-env-def
by (auto intro: fmdomI)

next
have vwelldefined' \( v_2 \)
apply (rule veval'-welldefined')
apply fact
using rec-comb by auto

show \( \text{fmpred} (\lambda-. \text{vwelldefined}') \) \( \text{env} \)
apply (rule vmatch-welldefined')
apply (rule vfind-match-elem)
apply fact+
done
qed

next
have fdisjnt C (fmdom \( \text{env} \))
unfolding \( \text{fmdom \( \text{env} \)} = \cdot \)
using \( (\text{pat}, \text{rhs}) \in \text{set \( cs \)} \) \( \text{not-shadows-vconsts} (\text{Vabs \( cs \)} \Gamma \))
by (auto simp: list-all-iff allconsts-def fdisjnt-alt-def)

moreover have fdisjnt C (fmdom css)
using \( \text{vwelldefined}' (\text{Vrecabs css name} \Gamma \))
by simp

ultimately show fdisjnt C (fmdom (\( \Gamma \)++f mk-rec-env css \( \Gamma \)++f \( \text{env} \)))
using \( \text{fdisjnt C} (\text{fmdom} \Gamma \))
unfolding fdisjnt-alt-def mk-rec-env-def by auto

next
show \( \neg \text{shadows-consts} \text{rhs} \)
using \( (\text{pat}, \text{rhs}) \in \text{set \( cs \)} \) \( \text{not-shadows-vconsts} (\text{Vabs \( cs \)} \Gamma \))
by (auto simp: list-all-iff)

next
have not-shadows-vconsts-env \( \Gamma \)
using \( \text{not-shadows-vconsts} (\text{Vabs \( cs \)} \Gamma \))
by auto

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

moreover have not-shadows-vconsts-env (mk-rec-env css \( \Gamma \))
unfolding mk-rec-env-def
using \( \text{not-shadows-vconsts} (\text{Vrecabs css name} \Gamma \))
by (auto intro: fmdomI)

ultimately show not-shadows-vconsts-env \( \Gamma \)++f mk-rec-env css \( \Gamma \)++f env
by blast

215
next
  show consts rhs $\subseteq$ fmdom (Γ, ++f mk-rec-env css Γ, ++f env) $\cup$ C
  using ⟨consts rhs $\subseteq$ ⟩ unfolding mk-rec-env-def
  by auto
qed

moreover have Γ ′ |- v $\$ u $\downarrow$ val'
proof (rule veval'.rec-comb)
  show Γ ′ |- v $\downarrow$ Vrecabs css name Γ ′
  using ⟨Γ ′ |- v $\downarrow$ v ′⟩
  unfolding ⟨v ′ = ⟩.
qed fact+

ultimately show ?case
  using rec-comb by metis
next
  case (constr name Γ ts us)
  have list-all (λ t. fmrel-on-fset (ids t) erelated Γ ′ Γ ∧ closed-except t (fmdom Γ)
  ∧ wellformed t ∧ consts t $\subseteq$ fmdom Γ $\cup$ C ∧ ¬ 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 $$ ts) $\subseteq$ fmdom Γ $\cup$ C⟩
      unfolding consts-list-comb
      by (metis Ball-set constr.prems(8) special-constants.sconsts-list-comb)
    subgoal
      using ⟨¬ shadows-consts (name $$ ts)⟩
      unfolding shadows.list-comb by (auto simp: list-ex-iff)
    done
  done

obtain us' where list-all3 (λt u u'. Γ ′ |- v t $\downarrow$ u' ∧ u' $\approx_e$ 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 \( u' \) where \( \text{list-all3} \ (\lambda t u u'. \Gamma' \vdash_v t \downarrow u' \land u' \approx_e u) \) \( ts us us' \)
by \( \text{auto} \)

have
\[
\text{fmrel-on-fset} \ (\text{ids} \ t) \ \text{erelated} \ \Gamma' \ \Gamma \ \text{closed-except} \ t \ (\text{fdom} \ \Gamma)
\text{wellformed} \ t \ \text{consts} \ t \ [\subseteq] \ \text{fdom} \ \Gamma \ [\cup] \ C \sim \text{shadows-consts} \ t
\text{using} \ \text{Cons} \ by \ \text{auto}
\]

then obtain \( u' \) where \( \Gamma' \vdash_v t \downarrow u' u' \approx_e u \)
using \( \text{closed-venv} \ \Gamma \) \( \text{wellformed-venv} \ \Gamma \) \( \text{fdisjnt} \ C \ (\text{fdom} \ \Gamma) \) \( \text{fmpred} \ (\lambda- \ \text{vwelldefined}') \ \Gamma) \)
using \( \text{not-shadows-vconsts-env} \ \Gamma \) \( \text{Cons.hyps} \)
by \( \text{blast} \)

show \( ?\text{case} \)
apply \( \text{rule} \ \text{Cons.prems} \)
apply \( \text{rule} \ \text{list-all3-cons} \)
apply \( \text{fact} \)
apply \( \text{rule} \ \text{conjl} \)
apply \( \text{fact}+ \)
done
\qedsymbol \ \text{auto}

show \( ?\text{case} \)
apply \( \text{rule} \ \text{constr.prems} \)
apply \( \text{rule} \ \text{veval'\:constr\[where \ us = us'\]} \)
apply \( \text{fact} \)
using \( \text{list-all3} - ts us \ us' \)
apply \( \text{induct}; \ \text{auto} \)
apply \( \text{rule} \ \text{erelated.intros} \)
using \( \text{list-all3} - ts us \ us' \)
apply \( \text{induct}; \ \text{auto} \)
done
\qedsymbol

end


Chapter 4

Preprocessing of code equations

theory Doc-Preproc
imports Main
begin
end

4.1 A type class for correspondence between HOL expressions and terms

theory Eval-Class
imports
../Rewriting/Rewriting-Term
../Utils/ML-Utils
Deriving.Derive-Manager
Dict-Construction.Dict-Construction
begin

no-notation Mpat-Antiquot.mpaq-App (infixl §§ 900)
hide-const (open) Strong-Term.wellformed
declare Strong-Term.wellformed-term-def[simp del]

class evaluate =
  fixes eval :: rule fset ⇒ term ⇒ 'a ⇒ bool (-/ ⊢/ (- ≈/ -) [50,0,50] 50)
  assumes eval-wellformed: rs ⊢ t ≈ a ⇒ wellformed t
begin

definition eval' :: rule fset ⇒ term ⇒ 'a ⇒ bool (-/ ⊢/ (- ↓/ -) [50,0,50] 50)
where
  rs ⊢ t ↓ a ←→ wellformed t ∧ (∃ t'. rs ⊢ t →∗ t' ∧ rs ⊢ t' ≈ a)

lemma eval'I[intro]:
assumes wellformed $t \vdash t \rightarrow^* t' \vdash t' \approx a$
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 $t \vdash t \rightarrow^* t' \vdash t' \approx a$
using assms unfolding eval'-def by auto

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

lemma eval-compose:
assumes wellformed $t \vdash t \rightarrow^* t' \vdash t' \downarrow a$
shows $rs \vdash t \downarrow a$
proof -
  from $(rs \vdash t' \downarrow a)$ obtain $t''$ where $rs \vdash t' \rightarrow^* t'' \vdash t'' \approx a$
  by blast
moreover hence $rs \vdash t \rightarrow^* t''$
  using assms by auto
ultimately show $rs \vdash t \downarrow a$
  using assms by auto
qed

end

instance fun :: (evaluate, evaluate) evaluate begin

  definition eval-fun where
  eval-fun $rs t a \longleftrightarrow wellformed t \land (\forall x t_x. rs \vdash t_x \downarrow x \rightarrow rs \vdash t \Downarrow t_x \downarrow a x)$
  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 $rs \vdash t \Downarrow f x$
  using assms unfolding eval-fun-def by blast

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

219
moreover have \( rs \vdash t \xrightarrow{\ast} t' \xrightarrow{\ast} t \)
  using \( (rs \vdash t \xrightarrow{\ast} t') \) by (rule rewrite.rt-fun[unfolded app-term-def])
moreover have \( rs \vdash t' \xrightarrow{\ast} t_x \)
  using \( (rs \vdash t' \approx f) \) by (rule eval-funD)
ultimately show \( rs \vdash t \xrightarrow{\ast} t_x \)
  by (rule eval-compose)

qed

lemma eval-ext:
  assumes wellformed \( f \land x t \). rs \vdash t \downarrow x \implies rs \vdash f \xrightarrow{\ast} t \downarrow a \)
  shows \( rs \vdash f \approx a \)
  using assms unfolding eval-fun-def by blast

lemma eval'-ext:
  assumes wellformed \( f \land x t \). rs \vdash t \downarrow x \implies rs \vdash f \xrightarrow{\ast} t \downarrow a \)
  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 \( f :: 'a::evaluate \Rightarrow 'b::evaluate \)
  assumes wellformed' \( 1 t \land u x \). rs \vdash u \downarrow x \implies rs \vdash t \[u] \downarrow f \)
  shows \( rs \vdash \Lambda t \downarrow f \)
  proof (rule eval'-ext)
    show wellformed \( \Lambda t \)
      using assms by simp
  next
    fix \( x :: 'a \) and \( u \)
    assume rs \vdash u \downarrow x
    show rs \vdash \Lambda t \xrightarrow{\ast} u \downarrow f \)
      proof (rule eval-compose)
        show wellformed \( \Lambda t \xrightarrow{\ast} u \)
          using assms (rs \vdash u \downarrow x) by auto
      next
        show rs \vdash \Lambda t \xrightarrow{\ast} u \downarrow \[u] \beta \)
          using (rs \vdash u \downarrow x) by (auto intro: rewrite.beta)
      next
        show rs \vdash \[u] \beta \downarrow f \)
          using assms (rs \vdash u \downarrow x) by auto
    qed
  qed

lemma eval-impl-wellformed[dest]: \( rs \vdash t \approx a \implies wellformed' \land t \)
by (auto dest: wellformed-inc eval-wellformed[unfolded wellformed-term-def])

lemma eval'-impl-wellformed[dest]: \( rs \vdash t \downarrow a \implies wellformed' \land t \)
unfolding eval'-def by (auto dest: wellformed-inc)
lemma wellformed-unpack:
  wellformed’ n (t $ u) ⇒ wellformed’ n t
  wellformed’ n (t $ u) ⇒ wellformed’ n u
  wellformed’ n (λ t) ⇒ wellformed’ (Suc n) t
by auto

lemma replace-bound-aux:
n < 0 ←→ False
Suc n < Suc m ←→ n < m
0 < Suc n ←→ True
((0::nat) = 0) ←→ True
(0 = Suc m) ←→ False
(Suc m = Suc n) ←→ n = m
(Suc m = 0) ←→ 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-step-term :: term × term ⇒ term ⇒ term option
where
  rewrite-step-term = rewrite-step

private lemmas rewrite-rt-fun = rewrite.rt-fun[unfolded app-term-def]
private lemmas rewrite-rt-arg = rewrite.rt-arg[unfolded app-term-def]

ML-file tactics.ML
end

method-setup wellformed = ⟨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

221
Eval-Class

**keywords** embed :: thy-decl

**begin**

fun non-overlapping' :: term ⇒ term ⇒ bool where
non-overlapping' (Const x) (Const y) ⟷ x ≠ y |
non-overlapping' (Const -) (- $ -) ⟷ True |
non-overlapping' (- $ -) (Const -) ⟷ True |
non-overlapping'(t₁ $ t₂) (u₁ $ u₂) ⟷ non-overlapping' t₁ u₁ ∨ non-overlapping'
  t₂ u₂ |
non-overlapping' - - ⟷ False

lemma non-overlapping-approx:
  assumes non-overlapping' t u
  shows non-overlapping t u
  using assms
  by (induct t u rule: non-overlapping'.induct) fastforce+

fun pattern-compatible' :: term ⇒ term ⇒ bool where
pattern-compatible' (t₁ $ t₂) (u₁ $ u₂) ⟷ pattern-compatible' t₁ u₁ ∧ (t₁ = u₁
  → pattern-compatible' t₂ u₂) |
pattern-compatible' t u ⟷ t = u ∨ non-overlapping' t u

lemma pattern-compatible-approx:
  assumes pattern-compatible' t u
  shows pattern-compatible t u
  using assms
  proof (induction t u rule: pattern-compatible.induct)
    case 2-1
    thus ?case
      by (force simp: non-overlapping-approx)
  next
    case 2-5
    thus ?case
      by (force simp: non-overlapping-approx)
  qed auto

abbreviation pattern-compatibles' :: (term × 'a) fset ⇒ bool where
pattern-compatibles' ≡ fpairwise (λ(lhs₁, -) (lhs₂, -). pattern-compatible' lhs₁ lhs₂)

definition rules' :: C-info ⇒ rule fset ⇒ bool where
rules' C-info rs ⟷
  fBall rs rule ∧
  arity-compatibles rs ∧
  is-fmap rs ∧
  pattern-compatibles' rs ∧
  rs ≠ ||| ∧
  fBall rs (λ(lhs, -). ¬ pre-constants.shadowsconsts C-info (heads-of rs) lhs) ∧
  fdisjnt (heads-of rs) (constructors.C 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 (\(\lambda (lhs, \cdot)\). pre-constants.shadows-consts C-info (heads-of rs) lhs)
  and fBall rs (\(\lambda (\cdot, rhs)\). 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: \(f = g \Rightarrow f \ x = g \ x\)
by auto

ML-file embed.ML
confs lift-term :: 'a ⇒ term (\(\cdot\))

setup:
  let
    fun embed ((Const (@{const-name lift-term}, \(\cdot\))) \$ t) = HOL-Term.mk-term false t
    | embed (t \$ u) = embed t \$ embed u
    | embed t = t
    in Context.theory-map (Syntax-Phases.term-check 99 lift (K (map embed))) end

end

4.3 Default instances

theory Eval-Instances
imports Embed
begin

ML-file eval-instances.ML
setup (Eval-Instances.setup)
derive evaluate nat bool list unit prod sum option char num name term

223
end
Chapter 5

Final stage: Translation to CakeML

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 = fnempty
unfolding nsEmpty-def by simp

context begin

private lemma build-rec-env-fmap0:
fmap-of-ns (foldr (λ (f, x, e). nsBind f (Recclosure envA funs' f)) funs env) =
fmap-of-ns env ++ f fmap-of-list (map (λ f, (Name f, Recclosure envA 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 \textit{build-rec-env-fmap}:
\[
\text{fmap-of-ns (build-rec-env \textit{funs} env}_{A}\text{ env}) = \text{fmap-of-ns env} + +\_I \text{ cake-mk-rec-env}
\]
unfolding \textit{build-rec-env-def} \textit{cake-mk-rec-env-def}
by (rule \textit{build-rec-env-fmap0})
end

5.2 Constructors according to CakeML

\textbf{definition} \textit{cake-tctor} :: \textit{string} ⇒ \textit{tctor} \textit{where}
\textit{cake-tctor} name = (if name = "fun" then \textit{Ast.TC-fn} else \textit{Ast.TC-name} (\textit{Short name}))

\textbf{primrec} \textit{typ-to-t} :: \textit{typ} ⇒ \textit{Ast.t} \textit{where}
\textit{typ-to-t} (TVar name) = \textit{Ast.Tvar} (as-string name) |
\textit{typ-to-t} (TApp name args) = \textit{Ast.Tapp} (map \textit{typ-to-t} args) (\textit{cake-tctor} (as-string name))

\textbf{context} constructors \textbf{begin}

\textbf{definition} \textit{as-static-cenv} :: \textit{c-ns} \textit{where}
\textit{as-static-cenv} = \textit{Bind} (rev (map (map-prod id (map-prod id (TypeId ◦ \textit{Short})))) (\text{flat-C-info})) []

\textbf{lemma} \textit{as-static-cenv-cakeml-static-env} :: \textit{cakeml-static-env} \textit{as-static-cenv}
unfolding \textit{cakeml-static-env-def} \textit{as-static-cenv-def}
by (auto simp: list.pred-map split: prod.splits)

\textbf{sublocale} \textit{cake-static-env?} :: \textit{cakeml-static-env} \textit{as-static-cenv}
by (rule \textit{as-static-cenv-cakeml-static-env})

\textbf{definition} \textit{as-cake-type-def} :: \textit{Ast.type-def} \textit{where}
\textit{as-cake-type-def} =
\textit{map} (\lambda (name, dt-def). (\textit{as-string} (\textit{tparams} dt-def), \textit{as-string} name, 
\textit{map} (\lambda (C, params). (\textit{as-string} C, map \textit{typ-to-t} params))
\textit{(sorted-list-of-fmap (constructors dt-def))}))
\textit{(sorted-list-of-fmap C-info)}

\textbf{definition} \textit{cake-dt-prelude} :: \textit{Ast.dec} \textit{where}
\textit{cake-dt-prelude} = \textit{Ast.Dtype empty-locs \textit{as-cake-type-def}}

\textbf{definition} \textit{cake-all-types} :: \textit{tid-or-exn set} \textit{where}
\textit{cake-all-types} = (TypeId ◦ \textit{Short} ◦ \textit{as-string}) ' fset all-tdefs

\textbf{definition} \textit{empty-state-with-types} :: unit SemanticPrimitives.state \textit{where}
\textit{empty-state-with-types} =
\{ clock = 0, refs = [], ffi = \textit{empty-ffi-state}, \textit{defined-types} = \textit{cake-all-types},

227
defined-mods = {} |

lemma empty-state-with-types-alt-def:
empty-state-with-types = empty-state (| defined-types := cake-all-types |)
ungfolding 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
ungfolding 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
ungfolding as-cake-type-def-def as-static-cenv-def build-tdefs-def alist-to-ns-def flat-C-info-def
apply (auto simp: List.bind-def map-concat)
apply (rule arg-cong[where f = concat])
by (auto simp: map-concat comp-def split-beta)

private lemmas evaluate-type =
evaluate-dec.dtype1 |
where new-tdecs = cake-all-types and s = empty-state and mn = [] and tds = as-cake-type-def,
unfolded state-types-update env-types-update,
folded empty-sem-env-def]

private lemma type-defs-to-new-tdecs:
type-defs-to-new-tdecs [] as-cake-type-def =
set (map (λname. TypeId (Short (as-string name))) (sorted-list-of-fset (fdom C-info)))
ungfolding 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 (λx 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 <-> distinct (tds => (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 = λx 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, Real 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 (λx. case x of (tvs, tn, ctors) => tn) as-cake-type-def)
    unfolding all-distinct-alt-def as-cake-type-def-def all-constructor-def
    by (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
Computability

\texttt{declare constructors.as-static-cenv\[code\]}
\texttt{declare constructors.as-cake-type-def-def\[code\]}
\texttt{declare constructors.cake-dt-prelude-def\[code\]}

\texttt{export-code constructors.as-static-cenv constructors.cake-dt-prelude}
\texttt{checking Scala}

end

5.3 CakeML backend

\texttt{theory CakeML-Backend}
\texttt{imports}
\texttt{ CakeML-Setup}
\texttt{ ../Terms/Value}
\texttt{ ../Rewriting/Rewriting-Sterm}
\texttt{begin}

5.3.1 Compilation

\texttt{fun mk-ml-pat :: pat \to Ast.pat where}
\texttt{mk-ml-pat (Patvar s) = Ast.Pvar (as-string s) |}
\texttt{mk-ml-pat (Patconstr s args) = Ast.Pcon (Some (Short (as-string s))) (map mk-ml-pat args)}

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

\texttt{context begin}

\texttt{private fun frees\' :: term \to name list where}
\texttt{frees\' (Free x) = [x] |}
\texttt{frees\' (t_1 \& t_2) = frees\' t_2 @ frees\' t_1 |}
\texttt{frees\' (\Lambda \; t) = frees\' t |}
\texttt{frees\' - = []}

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

\texttt{private lemma frees\'-list-comb: frees\' (list-comb f xs) = concat (rev (map frees\' xs)) @ frees\' f
by (induction xs arbitrary: f) (auto simp: app-term-def)}

\texttt{private lemma frees\'-distinct: linear pat \to distinct (frees\' pat)}

230
proof (induction pat)
case (App t u)
  hence distinct (frees' u @ frees' t)
    by (fastforce intro; distinct-append-fset fdisjnt-swap)
thus ?case
  by simp
qed auto

private fun pat-bindings' :: Ast.pat ⇒ name list where
 pat-bindings' (Ast.Pvar n) = [Name n] |
 pat-bindings' (Ast.Pcon - ps) = concat (rev (map pat-bindings' ps)) |
 pat-bindings' (Ast.Pref p) = pat-bindings' p |
 pat-bindings' (Ast.Pannot p -) = pat-bindings' p |
 pat-bindings' - = []

private lemma pat-bindings'-eq:
  map Name (pats-bindings ps xs) = concat (rev (map pat-bindings' ps)) @
  map Name xs

  map Name (pat-bindings p xs) = pat-bindings' p @ map Name xs
by (induction ps xs and p xs rule: pats-bindings-pat-bindings.induct) (auto simp: ac-simps)

private lemma pat-bindings'-empty-eq: map Name (pat-bindings p []) = pat-bindings' p
by (simp add: pat-bindings'-eq)

private lemma pat-bindings'-eq-frees: linear p ⇒ pat-bindings' (mk-ml-pat (mk-pat p)) = frees' p
proof (induction rule: mk-pat.induct)
case (1 t)
  show ?case
    using (linear t) proof (cases rule: linear-strip-comb-cases)
      case (comb s args)
      have map (pat-bindings' o mk-ml-pat o mk-pat) args = map frees' args
        proof (rule list.map-cong0, unfold comp-apply)
          fix x
          assume x ∈ set args
          moreover hence linear x
            using 1 comb by (metis linears-linear linears-strip-comb snd-conv)
          ultimately show pat-bindings' (mk-ml-pat (mk-pat x)) = frees' x
            using 1 comb by auto
        qed
    hence concat (rev (map (pat-bindings' o mk-ml-pat o mk-pat) args)) = concat (rev (map frees' args))
      by metis
    with comb show ?thesis
      apply (fold const-term-def)
apply (auto simp: strip-list-comb-const frees' list-comb comp-assoc)
apply (unfold const-term-def)
apply simp
done
qed auto

lemma mk-pat-distinct: linear pat \Rightarrow\ 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 \in\ 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 \in\ 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 (\lambda(pat, t). (mk-ml-pat (mk-pat pat), mk-con (frees pat \cup S) t)) cs

context begin

private lemma mk-exp-cupcake0:
  wellformed t \Rightarrow\ is-cupcake-exp (mk-exp S t)
  wellformed-clauses cs \Rightarrow\ cupcake-clauses (mk-clauses S cs) \land cake-linear-clauses
  (mk-clauses S cs)
  wellformed t \Rightarrow\ 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

232
proof
  from prems have $t = c$$ args
  apply (fold const-sterm-def)
  by (metis fst-conv list-strip-comb snd-conv)
show ?thesis
  apply (auto simp: list-all-iff simp del: mk-con.simps)
  apply (rule 5(1))
  apply (rule prems(1)[symmetric])
  apply (rule refl)
  apply (rule prems)
  apply assumption
  using ⟨wellformed t⟩ ⟨t = -⟩
  apply (auto simp: wellformed list-comb list-all-iff)
  done
qed
  using 5 by (auto split: prod.splits sterm.splits)
qed (auto simp: Let-def list-all-iff intro: mk-pat-distinct)
declare mk-con.simps[simp del]

lemma mk-exp-cupcake:
  wellformed t ⇒ 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) |
\textbf{app} related-exp \( t_1 \ u_1 \implies \text{related-exp} \ t_2 \ u_2 \implies \text{related-exp} \ (t_1 \ $ t_2) \ (\text{Ast.App})
\begin{align*}
\text{Ast.Opppp} [u_1, u_2] \mid \\
\text{fun: list-all2 (rel-prod related-pat related-exp) cs ml-cs} \implies \\
\ n \ [\notin S \ (\text{Subs cs})] \implies n \ [\notin \text{all-consts} \implies \\
\text{related-exp} \ (\text{Subs cs}) \ (\text{Ast.Fun} \ (\text{as-string n}) \ (\text{Ast.Mat} \ (\text{Ast.Var} \ \text{(Short}} \\
(\text{as-string n}))) \ ml-cs)) \mid \\
\text{mat: list-all2 (rel-prod related-pat related-exp) cs ml-cs} \implies \\
\text{related-exp} \ \text{scr ml-scr} \implies \\
\text{related-exp} \ (\text{Subs cs} \ $ \ \text{scr}) \ (\text{Ast.Mat} \ ml-scr \ ml-cs)
\end{align*}

\textbf{lemma} \text{related-exp-is-cupcake:}
\begin{itemize}
  \item \text{assumes} related-exp \( t \ e \) wellformed \( t \)
  \item \text{shows} is-cupcake-exp \( e \)
\end{itemize}
\textbf{using} \text{assms} \textbf{proof} induction
\begin{itemize}
  \item \text{case} (\text{fun cs ml-cs n})
    \item \text{hence} list-all (\lambda (p, t). linear pat \land wellformed t) cs \text{ by simp}
  \item \text{moreover have} cupcake-clauses ml-cs \land cake-linear-clauses ml-cs
    \item \text{using} (\text{list-all2 - cs ml-cs}) (\text{list-all - cs})
  \item \text{proof} induction
    \item \text{case} (\text{Cons c cs ml-c ml-cs})
      \item \text{obtain} ml-p ml-e \text{ where ml-c = (ml-p, ml-e) by fastforce}
      \item \text{obtain} p t \text{ where c = (p, t) by fastforce}
      \item \text{have} ml-p = \text{mk-ml-pat} (\text{mk-pat p})
        \item \text{using Cons unfolding} (ml-c = \cdot \langle c = \cdot \rangle \text{ by simp}
        \item \text{thus} ?\text{case}
          \item \text{using Cons unfolding} (ml-c = \cdot \langle c = \cdot \rangle \text{ by (auto intro: mk-pat-distinct)}
      \end{itemize}
\textbf{qed simp}
ultimately show ?\text{case}
\begin{itemize}
  \item \text{by auto}
\end{itemize}
\textbf{next}
\begin{itemize}
  \item \text{case} (\text{mat cs ml-cs scr ml-scr})
    \item \text{hence} list-all (\lambda (p, t). linear pat \land wellformed t) cs \text{ by simp}
    \item \text{moreover have} cupcake-clauses ml-cs \land cake-linear-clauses ml-cs
      \item \text{using} (\text{list-all2 - cs ml-cs}) (\text{list-all - cs})
    \item \text{proof} induction
      \item \text{case} (\text{Cons c cs ml-c ml-cs})
        \item \text{obtain} ml-p ml-e \text{ where ml-c = (ml-p, ml-e) by fastforce}
        \item \text{obtain} p t \text{ where c = (p, t) by fastforce}
        \item \text{have} ml-p = \text{mk-ml-pat} (\text{mk-pat p})
          \item \text{using Cons unfolding} (ml-c = \cdot \langle c = \cdot \rangle \text{ by simp}
          \item \text{thus} ?\text{case}
            \item \text{using Cons unfolding} (ml-c = \cdot \langle c = \cdot \rangle \text{ by (auto intro: mk-pat-distinct)}
        \end{itemize}
\textbf{qed simp}
ultimately show ?\text{case}
\begin{itemize}
  \item \text{using mat by auto}
\end{itemize}
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
doesn't 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.split id0.split)

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:

\( \text{list-all2 related-v us vs } \rightarrow \text{related-v (Vconstr name us)} (\text{Conv (Some (as-string name, -)) vs}) \) |

\( \text{related-fun cs n e } \rightarrow \text{fmrel-on-fset (ids (Sabs cs)) related-v } \Gamma (\text{fmap-of-ns (sem-env.v env)}) \rightarrow \text{related-v (Vabs cs } \Gamma \text{)} (\text{Closure env (as-string n) e}) |

\( \text{rec-closure:} \)

\( \text{fmrel-on-fset (fbind (fmran css) (ids \circ Sabs)) related-v } \Gamma (\text{fmap-of-ns (sem-env.v env)}) \rightarrow \text{related-v } \Gamma (\text{fmap-of-list (map (map-prod Name (map-prod Name id) exps))}) \rightarrow \text{related-v (Vrecabs css name } \Gamma \text{)} (\text{Reclosure env exps (as-string name)}) \)

abbreviation \( \text{var-env :: (name, value) fmap } \Rightarrow \text{((string \times v) list } \Rightarrow \text{bool where}} \)

\( \text{var-env } \Gamma \text{ ns } \equiv \text{fmrel related-v } \Gamma (\text{fmap-of-list (map (map-prod Name id) ns)}) \)

lemma \( \text{related-v-ext:} \)

assumes \( \text{related-v v ml-v} \)

assumes \( v' \approx_e v \)

shows \( \text{related-v v' ml-v} \)

using \( \text{assms} \) proof (induction arbitrary: \( v' \))

case (conv us ml-us name)

obtain ts where \( v' = \text{Vconstr name ts list-all2 erelated ts us} \)

by cases auto

have \( \text{list-all2 related-v ts ml-us} \)

by (rule list-all2-trans (OF - \{ \text{list-all2 erelated ts us}; conv(1) \})) auto

thus \( \text{case} \)

using conv unfolding \( (v' = -) \)

by (auto intro: related-v.conv)

next

case (\( \text{closure cs n e } \Gamma_2 \) env)

obtain \( \Gamma_1 \) where \( v' = \text{Vabs cs } \Gamma_1 \text{ fmrel-on-fset (ids (Sabs cs)) erelated } \Gamma_1 \text{ } \Gamma_2 \)

using \( (v' \approx_e -) \)

by cases auto

have \( \text{fmrel-on-fset (ids (Sabs cs)) related-v } \Gamma_1 (\text{fmap-of-ns (sem-env.v env)}) \)

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 \( \Gamma_1 \) \( \Gamma_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 \(\Gamma_2\) env exps name)
obtain \(\Gamma_1\) where \(v' = \text{Vrecabs css name} \ \Gamma_1\) pred-fmap (\(\lambda cs. \text{fmrel-on-fset (ids (Sabs cs)) elated} \ \Gamma_1 \ \Gamma_2\)) css
using \(\langle v' \approx e \ -\rangle\)
by cases auto

have fmrel-on-fset (fbind (fmran css) (ids o Sabs)) related-v \(\Gamma_1\) (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 \langle pred-fmap - css\rangle])
unfolding comp-apply
apply (drule fmrel-on-fsetD)
apply assumption
apply (metis (mono-tags, lifting) option.rel-sel)
done
done

done
done

show ?case
unfolding \(\langle v' = -\rangle\)
by (rule related-v.rec-closure) fact+

qed

context begin

private inductive match-result-related :: \((\text{string} \times v)\) list \(\Rightarrow\) \((\text{string} \times v)\) list
match-result \(\Rightarrow\) \((\text{name}, \text{value})\) \text{fmap} \text{option} \(\Rightarrow\) \text{bool for eenv where}
no-match: match-result-related eenv No-match None \mid
type-error: match-result-related eenv Match-type-error - \mid
match: var-env \(\Gamma\) eenv-m \(\Rightarrow\) match-result-related eenv (Match (eenv-m @ eenv)) (Some \(\Gamma\))

private corollary match-result-related-empty: match-result-related eenv (Match
proof

have \text{match-result-related} eenv (\text{Match} (\text{[]} \circ eenv)) (\text{Some \text{fnempty}})
  by (rule match-result-related.match) auto
thus \text{thesis}
  by simp
qed

private fun is-Match :: 'a match-result \Rightarrow \text{bool}
where
is-Match (\text{Match} \text{-}) \leftarrow \text{True}
is-Match \text{-} \leftarrow \text{False}

lemma cupcake-pmatch-related:
  assumes related-v v ml-v
  shows \text{match-result-related} eenv (\text{cupcake-pmatch as-static-cenv (mk-ml-pat pat) ml-v eenv}) (\text{vmatch pat v})
using assms proof (induction pat arbitrary: v ml-v eenv)
case (\text{Patvar} name)
  have \text{var-env} (fmap-of-list [(\text{name}, v)]) [(\text{as-string name}, ml-v)]
    using Patvar by auto
  hence \text{match-result-related} eenv (\text{Match} [(\text{as-string name}, ml-v)] \circ eenv)) (\text{Some (fmap-of-list [(\text{name}, v)])})
    by (rule match-result-related.match)
  thus \text{case}
    by simp
next
case (\text{Patconstr} name ps v0)
show \text{case}
  using Patconstr.prems
proof (cases rule: related-v.cases)
  case (\text{conv us vs name'} -)
  define f where
f p v m =
  (case m of
    \text{Match env} \Rightarrow \text{cupcake-pmatch as-static-cenv p v env}
  | m \Rightarrow m) \text{ for } p v m
{
  assume name = name'

  assume length ps = length us
  hence list-all2 (\lambda x. \text{True}) us ps
    by (induct rule: list-induct2) auto
  hence list-all3 (\lambda p v. \text{related-v t v}) us ps vs
    using list-all2 related-v us vs:
    by (rule list-all3-from-list-all2s) auto
hence \( *: \text{match-result-related eenv} \)
\[
(Matching\_fold2 f \text{Match-type-error} (\text{map mk-ml-pat ps}) \ vs (\text{Match (eenv-m \&@ eenv)}))
\]
\[
(\text{map-option (foldl (++f) \(\Gamma\)) (those (map2 vmatch ps us)))}
\]
\(\text{is} \)
\(\text{rel} \)
\]
\(\text{if} \ \var-env \ \Gamma \ eenv-m \)
\(\text{for} \ eenv-m \ \Gamma \)
\(\text{using} \ \text{Patconstr.IH} (\text{related-v \(\v0\ ml-v\)) that} \)
\(\text{proof} \ (\text{induction us ps vs arbitrary:} \ \Gamma \ eenv-m \ \text{rule: list-all3-induct}) \)
\(\text{case} \ (\text{Cons} t \ us \ p \ ps \ v \ vs) \)
\(\text{have} \ \text{match-result-related} (\text{eenv-m \&@ eenv}) (\text{cupcake-pmatch as-static-eenv (mk-ml-pat p) v (eenv-m \&@ eenv)}) (vmatch \ p \ t) \)
\(\text{using} \ \text{Cons by simp} \)
\(\text{thus} \ ?\text{case} \)
\(\text{proof} \ \text{cases} \)
\(\text{case} \ \text{no-match} \)
\(\text{thus} \ ?\text{thesis} \)
\(\text{unfolding} \ f\text{-def} \)
\(\text{apply} \ (\text{cases length} (\text{map mk-ml-pat ps}) = \text{length} \ vs) \)
\(\text{by} \ (\text{fastforce intro: match-result-related.intros simp:cup-pmatch-list-nomatch \(\text{cup-pmatch-list-length-neq})+) \)
\(\text{next} \)
\(\text{case} \ \text{error} \)
\(\text{thus} \ ?\text{thesis} \)
\(\text{unfolding} \ f\text{-def} \)
\(\text{apply} \ (\text{cases length} (\text{map mk-ml-pat ps}) = \text{length} \ vs) \)
\(\text{by} \ (\text{fastforce intro: match-result-related.intros simp:cup-pmatch-list-typerr \(\text{cup-pmatch-list-length-neq})+) \)
\(\text{next} \)
\(\text{case} \ (\text{match} \ \Gamma' \ eenv-m') \)
\(\text{have} \ \text{match-result-related eenv} \)
\(\[
(Matching\_fold2 f \text{Match-type-error} (\text{map mk-ml-pat ps}) \ vs (\text{Match ((eenv-m' \&@ eenv-m') \&@ eenv}))
\]
\(\text{(map-option (foldl (++f) (\(\Gamma + + f \Gamma') \text{(those (map2 vmatch ps us)))}) \)
\(\text{proof} \ (\text{rule Cons, rule Cons}) \)
\(\text{show} \ \var-env \ (\(\Gamma + + f \Gamma') (\text{eenv-m'} \&@ eenv-m) \)
\(\text{using} \ (\text{var-env} \ \Gamma \ eenv-m' \text{match} \)
\(\text{by} \ \text{force} \)
\(\text{qed} \ (\text{simp | fact})+) \)
\(\text{thus} \ ?\text{thesis} \)
\(\text{using} \ \text{match} \)
\(\text{unfolding} \ f\text{-def} \)
\(\text{by} \ (\text{auto simp: map-option.compositionality comp-def}) \)
\(\text{qed} \)

240
qed (auto intro: match-result-related.match)

moreover have var-env fnempty []
  by force

ultimately have ?rel [] fnempty
  by fastforce

hence ?thesis
  using conv (length ps = length us)
  unfolding f-def (name = name')
  by (auto intro: match-result-related.intros split: option.splits elim: static-cenv-lookup)
}
moreover
{
  assume name ≠ name'
  with conv have ?thesis
  by (auto intro: match-result-related.intros split: option.splits elim: same-ctor.elims simp: name.expand)
}
moreover
{
  let ?fold = Matching.fold2 f Match-type-error (map mk-ml-pat ps) vs (Match eenv)

  assume *: length ps ≠ length us
  moreover have length us = length vs
    using (list-all2 related-v us vs) by (rule list-all2-lengthD)
  ultimately have length ps ≠ length vs
    by simp

  moreover have ¬ is-Match (Matching.fold2 f err xs ys init)
    if ¬ is-Match err and length xs ≠ length ys for init err xs ys
    using that f-def
    by (induct f err xs ys init rule: fold2.induct) (auto split: match-result.splits)
  ultimately have ¬ is-Match ?fold
    by simp
  hence ?fold = Match-type-error ∨ ?fold = No-match
    by (cases ?fold) auto

  with * have ?thesis
    unfolding ml-v = → (v0 = →) f-def
    by (auto intro: match-result-related.intros split: option.splits)
}

ultimately show ?thesis
  by auto
qed (auto intro: match-result-related.intros)

qed
lemma match-all-related:
assumes list-all2 (rel-prod related-pat related-exp) cs ml-cs
assumes list-all (\(\lambda\) (pat, -), linear pat) cs
assumes related-v v ml-v
assumes cupcake-match-result as-static-cenv ml-v ml-cs Bindv = Rval (ml-rhs, ml-pat, eenv)
obtains rhs pat \(\Gamma\) where
\(\text{ml-pat} = \text{mk-ml-pat} (\text{mk-pat pat})\)
\(\text{related-exp rhs ml-rhs}\)
\(\text{vfind-match cs v} = \text{Some} (\Gamma, \text{pat}, \text{rhs})\)
\(\text{var-env } \Gamma\) eenv
using assms
proof (induction cs ml-cs arbitrary: thesis ml-pat ml-rhs rule: list-all2-induct)
case (Cons c cs ml-c ml-cs)
moreover obtain pat0 rhs0 where \(c = (\text{pat0}, \text{rhs0})\) by fastforce
moreover obtain ml-pat0 ml-rhs0 where \(\text{ml-c} = (\text{ml-pat0}, \text{ml-rhs0})\) by fastforce
ultimately have \(\text{ml-pat0} = \text{mk-ml-pat} (\text{mk-pat pat0})\) 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 (\text{mk-ml-pat} (\text{mk-pat pat0})) ml-v []) (vmatch (\text{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 \(\text{ml-c} = \_\)
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 \(\text{ml-c} = \_\)
by (simp split: if-splits)
next
fix pat rhs \(\Gamma\)
assume ml-pat = mk-ml-pat (mk-pat pat)
assume related-exp rhs ml-rhs
assume vfind-match cs v = Some (\(\Gamma\), pat, rhs)
assume var-env \(\Gamma\) eenv

242
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
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 (λcs (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 o Sabs)) related-v Γ_Λ (fmap-of-ns (sem-env.v env_Λ))
  shows fmrel related-v (mk-rec-env css Γ_Λ) (cake-mk-rec-env funs env_Λ)
proof (rule fmrelI)
  fix name
  have rel-option (λcs (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 (λcs (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_Λ funs (as-string name))
    using assms by (auto intro: related-v.recclosure)
  show rel-option related-v (fmlookup (mk-rec-env css Γ_Λ) name) (fmlookup (cake-mk-rec-env funs env_Λ) 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 fact
    apply (rule *
    done
qed

lemma mk-exp-correctness:
  ids t ⩽ S ⇒ all-consts ⩽ S ⇒ ¬ shadows-consts t ⇒ related-exp t (mk-exp S t)
  ids (Sabs cs) ⩽ S ⇒ all-consts ⩽ S ⇒ ¬ shadows-consts (Sabs cs) ⇒
  list-all2 (rel-prod related-pat related-exp) cs (mk-clauses S cs)

244
\[ \text{ids } t \subseteq S \implies \text{all-consts } \subseteq S \implies \neg \text{shadows-consts } t \implies \text{related-exp } t \quad (\text{mk-con } S t) \]

**proof** (induction rule: mk-exp-mk-clauses-mk-con.induct)

- **case** (2 S name)
  - **show** ?case
    - **proof** (cases name \( \in \) C)
      - **case** True
        - hence related-exp (name \$ []\$) \( (\text{mk-exp } S (\text{Sconst name})) \)
          - by (auto intro: related-exp.intros simp del: list-comb.simps)
        - **thus** ?thesis
          - by (simp add: const-sterm-def)
    - **qed** (auto intro: related-exp.intros)

- **next**
  - **case** (4 S cs)
    - have fresh-fNext \((S \cup \text{all-consts}) \not\in S \cup \text{all-consts}\)
      - by (rule fNext-not-member)
    - hence fresh-fNext S \(\not\in S \cup \text{all-consts}\)
      - using \((\text{all-consts } \subseteq S)\) by (simp add: sup-absorb1)
    - hence fresh-fNext S \(\not\in \text{ids } (\text{Sabs cs}) \cup \text{all-consts}\)
      - using 4 by auto
    - **show** ?case
      - apply (simp add: Let-def)
      - apply (rule related-exp.fun)
        - apply (rule 4.IH[unfolded mk-clauses.simps])
          - apply (rule refl)
          - apply fact
          - using (fresh-fNext S \(\not\in \text{ids } (\text{Sabs cs}) \cup \text{all-consts}\) by auto)
    - **next**
      - **case** (5 S t)
        - **show** ?case
          - apply (simp add: mk-con.simps split!: prod.splits sterm.splits if-splits)
          - **subgoal premises** prems for args c
            - **proof**
              - from prems have \( t = c \$ args \)
                - 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 \( t \subseteq S \)) unfolding (\( t = \_ \))
apply (auto simp: ids-list-comb)
by (meson \( f\text{Union-subset-elem} \ f\text{image-eqI} \ f\text{set-of-list-elem} \ f\text{set-rev-mp} \))
subgoal by (rule 5)

subgoal
using (\( \neg \text{shadows-consts} t \)) unfolding (\( t = \_ \))
unfolding shadows.list-comb
by (auto simp: list-ex-iff)
done

qed
using 5 by (auto split: prod.splits stem.splits)

next

have list-all2 (\( \lambda x \ y. \ \text{rel-prod} \ \text{related-pat} \ \text{related-exp} \ x \) (case y of (pat, t) \( \Rightarrow \) (mk-ml-pat (mk-pat pat), mk-con (frees pat \( \cup \) S) t)) cs cs
proof (rule list.rel-refl-strong, safe intro!: rel-prod.intros)
fix pat rhs
assume (pat, rhs) \( \in \) set cs

have \( \text{consts} \ \text{rhs} \ \| \subseteq \ S \)
using (ids \( \text{Sabs} \ cs \) \| \subseteq \ S\))
unfolding \( \text{ids-def} \)
apply auto
apply (drule \( f\text{Union-least-rev} \))+
apply (auto simp: \( f\text{set-of-list-elem} \) elim \( f\text{BallE} \))
done

have frees rhs \| \subseteq \ frees pat \( \cup \) S
using (ids \( \text{Sabs} \ cs \) \| \subseteq \ S\) \( ; (\text{pat}, \ \text{rhs}) \in \text{set} \ cs \))
unfolding \( \text{ids-def} \)
apply auto
apply (drule \( f\text{Union-least-rev} \))+
apply (auto simp: \( f\text{set-of-list-elem} \) elim \( f\text{BallE} \))
done

have \( \neg \text{shadows-consts} \ \text{rhs} \)
using (\( \text{pat}, \ \text{rhs} \) \( \in \) set cs) 6
by (auto simp: list-ex-iff)

show related-exp rhs (mk-con (frees pat \( \cup \) S) rhs)
apply (rule 6)
apply fact
subgoal by simp
subgoal
unfolding \( \text{ids-def} \)
using (\( \text{consts} \ \text{rhs} \ \| \subseteq \ S \) \( ; (\text{frees} \ \text{rhs} \ \| \subseteq \ frees \ \text{pat} \ \| \cup \ S) \) \( ; \) \( \text{by} \) auto
subgoal using \(6(3)\) by fast

subgoal by fact
done

qed

thus \(?\)case
  by (simp add: list.rel-map)
qed (auto intro: related-exp.intros simp: ids-def fdisjnt-alt-def)

context begin

private lemma semantic-correctness0:
  fixes exp
  assumes cupcake-evaluate-single env exp r is-cupcake-all-env env
  assumes fmrel-on-fset (ids t) related-v \(\Gamma\) (fmmap-of-ns (sem-env.v env))
  assumes related-exp t exp
  assumes wellformed t wellformed-venv \(\Gamma\)
  assumes closed-venv \(\Gamma\)
  assumes \(\Gamma\) closed-except t (fmdom \(\Gamma\))
  assumes \(\lambda\text{-}welldefined\(\)' \(\Gamma\) \(\Gamma\) consts t \(\subseteq\) fmdom \(\Gamma\) \(\cup\) \(C\)
  assumes fdisjnt \(C\) (fmdom \(\Gamma\))
  assumes \(\neg\text{ shadows-consts} t\) \(\neg\text{not-shadows-vconsts-env} \Gamma\)
  shows if-rval \((\lambda\text{ml-v}. \exists v. \Gamma \vdash v \downarrow v \land \text{related-v} v \text{ml-v})\) \(r\)
using assms proof (induction arbitrary: \(\Gamma\) t)
  case (con1 env cn es ress ml-vs ml-v')
    obtain name ts where cn = Some (Short (as-string name)) name \(\in\) \(C\) t =
      name \(\subseteq\) 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 = \langle\text{list-all2-shortcircuit \(-\mbox{-}\)}\rangle\)
  unfolded \(\langle\text{ress = \(-\)}\rangle\) list-all2-shortcircuit-real list-all2-rev1,
  folded ml-vs',def]
moreover have
  list-all wellformed ts list-all (\(\lambda\text{t.} \neg\text{shadows-consts} t\) ts
  list-all (\(\lambda\text{t.} \text{consts} t \subseteq\) fmdom \(\Gamma\) \(\cup\) \(C\)) ts list-all (\(\lambda\text{t.} \text{closed-except} t\) (fmdom
  \(\Gamma\))) ts
  subgoal using wellformed t; unfolding \(t = \neg\) wellformed.list-comb by simp
  subgoal
    using \(\neg\text{shadows-consts} t\); unfolding \(t = \neg\) 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 $S$ ts) cons1.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 (λt'. fnrel-on-fset (ids t') related-v Γ (fmap-of-ns (sem-env.v env))) ts
proof (standard, rule fnrel-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 fnrel-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 Γ Γv t ↓ u related-v u ml-v
proof
show is-cupcake-all-env env related-exp t e wellformed-env Γ closed-env Γ
  fnpred (λv. vwelldefined') Γ fdisjnt C (fdom Γ)
  not-shadows-vconsts-env Γ
  by fact+
next
show wellformed t ¬ shadows-cons ts closed-except t (fmdom Γ)
  consts t ⊆ fmdom Γ |∪| C fnrel-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) \(t = Sconst name name | \notin C\)
  unfolding \(id = -\)
  apply (cases t)
  using name.expand by blast+
  thus ?case
  proof cases
    case var
    hence name \(\in\) ids t
    unfolding ids-def by simp

    have rel-option related-v (fmlookup \(\Gamma\) name) (cupcake-nsLookup (sem-env.v env) (as-string name))
    using \(\text{fmrel-on-fset} (ids t) \; - \; -\)
    apply -
    apply (drule fmrel-on-fsetD[OF \(\langle name \; |\; \in\; ids\; t\rangle\)])
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 \( \text{fmap-of-ns} (\text{sem-env.v env}) = \text{fmap-of-list} ns \)
apply (cases env)
apply simp
subgoal for \( v \) by (cases \( v \)) simp
done

show \(?case\)
apply simp
apply (rule exI)
apply (rule conjI)
unfolding \( \langle t = \_ \rangle \)
apply (rule veval′.abs)
unfolding \( \langle n = \_ \rangle \)
apply (rule related-v.closure)
unfolding \( \langle u = \_ \rangle \)
apply (rule subst related-fun-alt-def; rule conjI)
apply fact
apply (rule conjI; fact)
using \( \langle \text{fmrel-on-fset} (\text{ids} \ t \ - \ - \ -) \rangle \)
unfolding \( \langle t = \_ \rangle \langle \text{fmap-of-ns} (\text{sem-env.v env}) = \_ \rangle \)
bysimp

next
case (\( \text{app1 env exps ress ml-vs env′ exp′ bv} \))
from \( \langle \text{related-exp} t \ - \ - \ \rangle \) obtain \( \text{exp1 exp2 t1 t2} \)
where \( \text{rev exps} = [\text{exp2, exp1}] \)
\( \text{exps} = [\text{exp1, exp2}] \)
\( t = t1 \; $s, t2 \)
and \( \text{related-exp} t1 \; \text{exp1 related-exp} t2 \; \text{exp2} \)
by cases auto
moreover from \( \text{app1} \) have \( \text{ress} = \text{map Real ml-vs} \)
by auto
ultimately obtain \( \text{ml-v1 ml-v2} \) where \( \text{ml-vs} = [\text{ml-v2, ml-v1}] \)
using \( \text{app1(1)} \)
by (smt list-all2-shortcircuit-real list-all2-Cons1 list-all2-Nil)

have \( \text{is-cupcake-exp} \; \text{exp1 is-cupcake-exp} \; \text{exp2} \)
using \( \text{app1 unfolding} \; \text{exps} = \_ \) by (auto dest: related-exp-is-cupcake)
moreover have \( \text{fmrel-on-fset} (\text{ids} \ t1) \) related-v \( \Gamma \) \( \langle \text{fmap-of-ns} (\text{sem-env.v env}) \rangle \)
using \( \text{app1 unfolding} \; \text{ids-def} \; \langle t = \_ \rangle \)
by (auto intro: fmrel-on-fsubset)
moreover have \( \text{fmrel-on-fset} (\text{ids} \ t2) \) related-v \( \Gamma \) \( \langle \text{fmap-of-ns} (\text{sem-env.v env}) \rangle \)
using \( \text{app1 unfolding} \; \text{ids-def} \; \langle t = \_ \rangle \)
by (auto intro: fmrel-on-fsubset)
ultimately have
cupcake-evaluate-single env \( \text{exp1} \) \( \langle \text{Real ml-v1} \rangle \)
cupcake-evaluate-single env \( \text{exp2} \) \( \langle \text{Real ml-v2} \rangle \) and
\( \exists t1'. \; \Gamma \vdash t1 \downarrow t1' \) \( \wedge \) related-\( v \) \( t1' \; \text{ml-v1} \; \exists t2'. \; \Gamma \vdash t2 \downarrow t2' \) \( \wedge \) related-\( v \) \( t2' \; \text{ml-v2} \)
using \( \text{app1 (related-exp} t1 \; \text{exp1} \); (related-exp \( t2 \; \text{exp2} \))
unfolding \( \langle \text{ress} = \_ \rangle \langle \text{exps} = \_ \rangle \langle \text{ml-vs} = \_ \rangle \langle t = \_ \rangle \)
by (auto simp: closed-except-def)
then obtain \(v_1, v_2\)

where \(\Gamma \vdash t_1 \downarrow v_1 \ldots \downarrow v_1\) related-v \(v_1\) ml-v_1

and \(\Gamma \vdash t_2 \downarrow v_2 \ldots \downarrow v_2\) 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 \(\langle ml-vs = \cdot \rangle\) by simp

hence is-cupcake-exp \(\exp'\) is-cupcake-all-env env'

using \(\langle\text{do-opapp} \cdot = \cdot\rangle\) by (metis cupcake-opapp-preserve)+

have vclosed v_1

proof (rule veval'-closed)

show closed-except \(t_1\) (fmdom \(\Gamma\))

using \(\langle\text{closed-except} - (\text{fmdom} \ \Gamma)\rangle\)

unfolding \(\langle t = \cdot \rangle\)

by (simp add: closed-except-def)

next

show wellformed \(t_1\)

using \(\langle \text{wellformed} \ t \rangle\) unfolding \(\langle t = \cdot \rangle\)

by simp

ded fact+

have vclosed v_2

apply (rule veval'-closed)

apply fact

using app1 unfolding \(\langle t = \cdot \rangle\) by (auto simp: closed-except-def)

have vwellformed v_1

apply (rule veval'-wellformed)

apply fact

using app1 unfolding \(\langle t = \cdot \rangle\) by auto

have vwellformed v_2

apply (rule veval'-wellformed)

apply fact

using app1 unfolding \(\langle t = \cdot \rangle\) by auto

have vwelldefined' v_1

apply (rule veval'-welldefined')

apply fact

using app1 unfolding \(\langle t = \cdot \rangle\) by auto

have vwelldefined' v_2

apply (rule veval'-welldefined')

apply fact

using app1 unfolding \(\langle t = \cdot \rangle\) by auto
have not-shadows-vconsts v₁
apply (rule veval'-shadows)
apply fact
using app₁ unfolding \( t = \cdot \) by auto
have not-shadows-vconsts v₂
apply (rule veval'-shadows)
apply fact
using app₁ unfolding \( t = \cdot \) by auto

show ?case
proof (rule if-rvalI)
fix ml-v
assume \( bv = \text{Real } ml-v \)
show \( \exists v. \Gamma \vdash v \downarrow v' \wedge \text{related } v \text{ ml-v} \)
using (do-opapp - = \cdot)
proof (cases rule: do-opapp-cases)
case (closure env_Λ n)
then have closure':
\[ \text{ml-v}_1 = \text{Closure } env_\Lambda (\text{as-string } (\text{Name } n)) \text{ exp'} \]
\[ \text{env'} = \text{update-v } (\lambda_. \text{nsBind } (\text{as-string } (\text{Name } n))) \text{ ml-v}_2 (\text{sem-env.v env}_\Lambda) \]
unfolding (ml-vs = \cdot)
by auto
obtain \( \Gamma_\Lambda cs \)
where \( v_1 = \text{Vabs } cs \Gamma_\Lambda \text{ related-fun } cs \text{ (Name } n) \text{ exp'} \)
and \( \text{fmrel-on-fset } (\text{ids } (\text{Sabs } cs)) \text{ related } \Gamma_\Lambda (\text{fnmap-of-ns } (\text{sem-env.v env}_\Lambda)) \)
using (related-v v₁ ml-v₁) unfolding (ml-v₁ = \cdot)
by cases auto

then obtain ml-cs
where \( \text{exp'} = \text{Mat } (\text{Var } (\text{Short } (\text{as-string } (\text{Name } n)))) \text{ ml-cs Name n} \)
|\( \notin \) | ids (Sabs cs) Name n |\( \notin \) | all-consts
and \( \text{list-all2 } (\text{rel-prod } \text{related-pat } \text{related-exp} ) \text{ cs ml-cs} \)
by (auto elim: related-funE)

hence \( \text{cupcake-evaluate-single } \text{env'} (\text{Mat } (\text{Var } (\text{Short } (\text{as-string } (\text{Name } n)))) \text{ ml-cs } (\text{Real } ml-v) \)
using (cupcake-evaluate-single env' exp' bv)
unfolding (bv = \cdot)
by simp

then obtain m-env v' ml-rhs ml-pat
where \( \text{cupcake-evaluate-single } \text{env'} (\text{Var } (\text{Short } (\text{as-string } (\text{Name } n)))) \text{ ml-cs } (\text{Real } ml-v) \)
and \( \text{cupcake-match-result } (\text{sem-env.c env'} ) v' \text{ ml-cs } \text{Bindv } = \text{Real } (\text{ml-rhs, ml-pat, m-env}) \)
and \( \text{cupcake-evaluate-single } (\text{env' } []) \text{ sem-env.v := nsAppend } (\text{alist-to-ns m-env}) (\text{sem-env.v env'} []) \text{ ml-rhs } (\text{Real } ml-v) \)
by cases auto

have
closed-venv (fmupd (Name n) v2 ΓΑ) wellformed-venv (fmupd (Name n) v2 ΓΑ) not-shadows-vcons-teq (fnmpred (λx. vwelldefined′) (fmupd (Name n) v2 ΓΑ)) using (vclosed v1) (vclosed v2) using (vwellformed v1) (vwellformed v2) using (not-shadows-vconsts v1) (not-shadows-vconsts v2) using (vwelldefined′ v1) (vwelldefined′ v2) unfolding ⟨v1 = -⟩
by auto

have closed-except (Sabs cs) (fmdom (fmupd (Name n) v2 ΓΑ))
using (vclosed v1) unfolding ⟨v1 = -⟩
apply (auto simp: Sterm.closed-except-simps list-all-iff)
apply (auto simp: closed-except-def)
done

have cons-teq (Sabs cs) ℒ| fmdom (fmupd (Name n) v2 ΓΑ) ℒ| C
using (vwelldefined′ v1) unfolding ⟨v1 = -⟩
unfolding scnscts-sabs
by (auto simp: list-all-iff)

have ¬ shadows-consts (Sabs cs)
using (not-shadows-vconsts v1) unfolding ⟨v1 = -⟩
by (auto simp: list-all-iff list-ex-iff)

have fdisjnt C (fmdom ΓΑ)
using (vwelldefined′ v1) unfolding ⟨v1 = -⟩
by simp

have if-rval (λml-v. ∃v. fmupd (Name n) v2 ΓΑ ⊢v Sabs cs $a Svar (Name n) \downarrow v ∧ related-v v ml-v) bv
proof (rule app1 (2))
show fnrel-on-fset (ids (Sabs cs $a Svar (Name n))) related-v (fmupd (Name n) v2 ΓΑ) (fnmap-of-ns (sem-env.v env'))
unfolding closure'
apply (simp del: frees-sterm.simps(3) const-s-sterm.simps(3) name.sel add: ids-def split!: sem-env.splits)
apply (rule fnrel-on-fset-updateI)
apply (fold ids-def)
using fnrel-on-fset (ids (Sabs cs)) related-v ΓΑ - apply simp
apply (rule related-v v2 ml-v2)
done
next
show wellformed (Sabs cs $a Svar (Name n))
using :vwellformed v1) unfolding ⟨v1 = -⟩

254
by simp
next
  show related-exp \(S\sabs \, cs \, \$, \, S\var \, (\, \text{Name} \, n)\) \(\, \text{exp}'\) unfolding \(\text{\langle exp' = \sim \rangle}\) using \(\\text{\langle list-all2 \, (rel-prod \, related-pat \, related-exp) \, cs \, ml-cs \rangle}\) by \(\text{auto \ intro:related-exp.intro \ simp \ del: name.sel}\)
next
  show closed-except \(S\sabs \, cs \, \$, \, S\var \, (\, \text{Name} \, n)\) \(\, \text{\langle fmdom \, (fmupd \, (\, \text{Name} \, n) \, \, v \, 2 \, \Gamma \, \Lambda)\rangle}\) by \(\text{simp add: closed-except-def}\)
next
  show \(\neg \, \text{shadows-consts} \, (\, \text{Name} \, n)\) using \(\text{\langle \neg \, \text{shadows-consts} \, (\, \text{Name} \, n)\rangle}\) \(\text{\langle Name \, n \, |\ / \in| \, all-consts\rangle}\) by simp
next
  show \(\text{consts} \, (\, \text{Name} \, n) \, |\subseteq| \, \text{fmdom} \, (\, \text{fmupd} \, (\, \text{Name} \, n) \, \, v \, 2 \, \Gamma \, \Lambda)\, |\cup| \, C\) using \(\text{consts} \, (\, \text{Name} \, n) \, |\subseteq| \, \text{fmdom} \, (\, \text{fmupd} \, (\, \text{Name} \, n) \, \, v \, 2 \, \Gamma \, \Lambda)\, |\cup| \, C\) by simp
next
  show \(\text{fdisjnt} \, C \, (\, \text{fmdom} \, (\, \text{fmupd} \, (\, \text{Name} \, n) \, \, v \, 2 \, \Gamma \, \Lambda)\)\) using \(\text{\langle Name \, n \, |\ / \in| \, all-consts\rangle}\) \(\text{\langle fdisjnt} \, C \, (\, \text{fmdom} \, \Gamma \, \Lambda)\rangle\) unfolding \(\text{fdisjnt-alt-def \ all-consts-def}\) by auto
qed fact+
then obtain \(v\) where \(\text{fmupd} \, (\, \text{Name} \, n) \, \, v \, 2 \, \Gamma \, \Lambda \, \vdash_v \, \text{Sabs cs \, \$, \, S\var \, (\, \text{Name} \, n)\, \downarrow} \, \text{\langle related-v \, v \, ml-v\rangle}\) unfolding \(\text{\langle bv = \sim \rangle}\) by auto
then obtain \(v\) where \(\text{vfind-match cs \, v \, 2} \, \text{\langle Some \, (env, \, pat, \, rhs)\rangle}\) and \(\text{fmupd} \, (\, \text{Name} \, n) \, \, v \, 2 \, \Gamma \, \Lambda \, \vdash_v \, \text{env \, \vdash \, \text{\text{v}} \, \text{rhs} \, \downarrow} \, \text{\langle v \, related-v \, v \, ml-v\rangle}\) unfolding \(\text{\langle bv = \sim \rangle}\) by auto
hence \(\text{\langle \, pat, \, rhs\rangle} \in \, \text{set \, cs \, \text{vmatch \, (mk-pat \, pat) \, v \, 2} \, \text{\langle Some \, env\rangle}\rangle\) by \(\text{metis \, vfind-match-elem}\)+
hence linear pat wellformed rhs using \(\text{\langle wellformed \, v \, 1\rangle}\) unfolding \(\text{\langle v \, 1 = \sim \rangle}\) by \(\text{auto \ simp: list-all-iff}\)
hence frees pat = patvars \(\text{(mk-pat-pat)}\) by \(\text{simp add: mk-pat-frees}\)
hence \(\text{fmdom \, env = \, frees \, pat}\) apply simp apply \(\text{\langle rule \, vmatch\,-\,dom\rangle}\) apply \(\text{\langle rule \, vmatch \, (mk-pat-pat) \, v \, 2} \, \text{\langle Some \, env\rangle}\) done
obtain \(v'\) where \(\Gamma \, \vdash_{\, v} \, \text{env} \, \vdash_{\, v} \, \, \text{rhs} \, \downarrow \, \text{\langle v' \, v' \, \approx_{\, v}\rangle}\) proof \(\text{\langle rule \, veval\,-\,agree-eq\rangle}\) show \(\text{fmupd} \, (\, \text{Name} \, n) \, \, v \, 2 \, \Gamma \, \Lambda \, \vdash_v \, \text{env} \, \vdash_{\, v} \, \text{rhs} \, \downarrow \, \text{\langle v \, by \, fact}\)
next
have $s$: Name $n$ $\notin$ ids $\text{rhs}$ if Name $n$ $\notin$ fmdom $\text{env}$
proof
assume Name $n$ $\in$ ids $\text{rhs}$
thus $\text{False}$
unfolding ids-def
proof (cases rule: funion-strictE)
case $A$
moreover have Name $n$ $\notin$ frees $\text{pat}$
using that unfolding (fmdom $\text{env}$ = frees $\text{pat}$) .
ultimately have Name $n$ $\in$ frees (Sabs $\text{cs}$)
apply auto
unfolding fUnion-alt-def
apply simp
apply (rule fBexI[where $x$ = (pat, rhs)])
apply (auto simp: fset-of-list-elem)
apply (rule (pat, rhs) $\in$ set $\text{cs}$)
done
thus $?\text{thesis}$
using :Name $n$ $\notin$ ids (Sabs $\text{cs}$): unfolding ids-def
by blast
next
case $B$
hence Name $n$ $\in$ consts (Sabs $\text{cs}$)
apply auto
unfolding fUnion-alt-def
apply simp
apply (rule fBexI[where $x$ = (pat, rhs)])
apply (auto simp: fset-of-list-elem)
apply (rule (pat, rhs) $\in$ set $\text{cs}$)
done
thus $?\text{thesis}$
using :Name $n$ $\notin$ ids (Sabs $\text{cs}$): unfolding ids-def
by blast
qed
qed

show fnrel-on-fset (ids $\text{rhs}$) erelated (Γ $\Lambda$ ++ $\text{f}$ $\text{env}$) (fmupd (Name $n$) $v_2$ Γ $\Lambda$ ++ $\text{f}$ $\text{env}$)
apply rule
apply auto
apply (rule option.rel-refl; rule erelated-refl)
using $*$ apply auto[]
apply (rule option.rel-refl; rule erelated-refl)+
done
next
show closed-venv (fmupd (Name $n$) $v_2$ Γ $\Lambda$ ++ $\text{f}$ $\text{env}$)
apply rule
apply fact
apply (rule vclosed.vmatch-env)
apply fact
apply fact
done
next
show wellformed-venv (fmupd (Name n) v₂ Γₐ ++f env)
apply rule
apply fact
apply (rule vuwellformed.vmatch-env)
apply fact
apply fact
done
next
show closed-except rhs (fmdom (fmupd (Name n) v₂ Γₐ ++f env))
using (fmdom env = frees pat) ((pat, rhs) ∈ set cs)
using :closed-except (Sabs cs) (fmdom (fmupd (Name n) v₂ Γₐ))
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 Γₐ ++f env) |∪| C
using :consts (Sabs cs) |⊆| fmdom (fmupd (Name n) v₂ Γₐ) |∪| 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 Γₐ ++f env))
using :Name n |∉| all-consts (fdisjnt C (fmdom Γₐ))
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 Γₐ ++f env)
using :not-shadows-vconsts-env (fmupd (Name n) v₂ Γₐ) by blast
next
have fmpred (λ-. vuwelldefined’) env
by (rule vmatch-welldefined’) fact+
thus fmpred (λ-. vuwelldefined’) (fmupd (Name n) v₂ Γₐ ++f env)
using :fmpred (λ-. vuwelldefined’) (fmupd (Name n) v₂ Γₐ) by blast
qed blast
show thesis
apply (intro exI conjI)
unfolding (t = →)
apply (rule veval'.comb)
using |ΓΓv t1 v1| unfolding (v1 = →)
apply blast
apply fact
apply fact+
apply (rule related-v-ext)
apply fact+
done
next
case (reclosure env funs name n)
with reclosure have reclosure':
ml-v1 = Reclosure env funs name
env' = update-v (λ-. nsBind (as-string (Name n)) ml-v2 (build-rec-env funs env funs (sem-env.v env v))) env
unfolding (ml-vs = →) by auto
obtain ΓΛ css
where v1 = Vrecabs css (Name name) ΓΛ
and fmrel-on-fset (bind (fmran css) (ids o Sabs)) related-v ΓΛ
(fmmap-of-ns (sem-env.v envΛ))
and fmrel (λcs (n, e). related-fun cs n e) css
(fmmap-of-list (map (map-prod Name (map-prod Name id)) funs))
using (related-v v1 ml-v1) unfolding (ml-v1 = →)
by cases auto
from fmrel - - have rel-option (λcs (n, e). related-fun cs (Name n) e)
(fmlookup css (Name name)) (find-recfun name funs)
apply −
apply (subst option.rel-set)
apply auto
apply (drule fmrel-fmdom-eq)
apply (drule fmdom-notI)
using (v1 = Vrecabs css (Name name) ΓΛ; vwellformed v1) apply auto[1]
using reclosure(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

258
where \(\text{exp}' = \text{Mat} (\text{Var} (\text{Short} (\text{as-string} (\text{Name} \ n))))\) ml-cs Name \ n

\[ |\not\in| \text{ids}(\text{Subs} \ cs) \text{Name} \ n \ |\not\in| \text{all-consts} \]

\[\text{and \ list-all2 (rel-prod related-pat related-exp) \ cs \ ml-cs}\]

by (auto elim: related-funE)

hence \(\text{cupcake-evaluate-single} \ \text{env}' (\text{Mat} (\text{Var} (\text{Short} \ n)))\) ml-cs

using \(\langle\text{cupcake-evaluate-single} \ \text{env}' \ \text{exp}' \ \text{bv}\rangle\)

unfolding \(\langle\text{bv} = \cdot\rangle\)

by simp

then obtain \(m\text{-env} \ v' \ \text{ml-rhs} \ \text{ml-pat}\)

where \(\text{cupcake-evaluate-single} \ \text{env}' (\text{Var} (\text{Short} \ n))\) (Real \(\text{v}'\))

and \(\text{cupcake-match-result} (\text{sem-env} \ . \ c \ \text{env}' \ v') \ \text{ml-cs} \ \text{Bindv} = \text{Real} (\text{ml-rhs}, \ \text{ml-pat}, \ \text{m-env})\)

and \(\text{cupcake-evaluate-single} \ (\text{env}' [\text{sem-env} . v := \text{nsAppend} (\text{alist-to-ns} \ \text{m-env})]) \ \text{ml-rhs} \ (\text{Real} \ \text{ml-v})\)

by cases auto

have closed-venv \(\langle\text{fmupd} (\text{Name} \ n) \ v_2 \ (\Gamma_\Lambda \ ++f \ \text{mk-rec-env} \ \text{css} \ \Gamma_\Lambda)\rangle\)

using \(\langle\text{velosed} \ v_1 \rangle \langle\text{velosed} \ v_2\rangle\)

using \(\langle\text{fmlookup} \ \text{css} (\text{Name} \ \text{name}) = \text{Some} \ cs\rangle\)

unfolding \(\langle v_1 = \cdot \rangle \ \text{mk-rec-env-def}\)

apply auto

apply rule

apply rule

apply \(\langle\text{auto intro: fmdomI}\rangle\)

done

have wellformed-venv \(\langle\text{fmupd} (\text{Name} \ n) \ v_2 \ (\Gamma_\Lambda \ ++f \ \text{mk-rec-env} \ \text{css} \ \Gamma_\Lambda)\rangle\)

using \(\langle\text{wellformed} \ v_1 \rangle \langle\text{wellformed} \ v_2\rangle\)

using \(\langle\text{fmlookup} \ \text{css} (\text{Name} \ \text{name}) = \text{Some} \ cs\rangle\)

unfolding \(\langle v_1 = \cdot \rangle \ \text{mk-rec-env-def}\)

apply auto

apply rule

apply rule

apply \(\langle\text{auto intro: fmdomI}\rangle\)

done

have not-shadows-vconsts-env \(\langle\text{fmupd} (\text{Name} \ n) \ v_2 \ (\Gamma_\Lambda \ ++f \ \text{mk-rec-env} \ \text{css} \ \Gamma_\Lambda)\rangle\)

using \(\langle\text{not-shadows-vconsts} \ v_1 \rangle \langle\text{not-shadows-vconsts} \ v_2\rangle\)

using \(\langle\text{fmlookup} \ \text{css} (\text{Name} \ \text{name}) = \text{Some} \ cs\rangle\)

unfolding \(\langle v_1 = \cdot \rangle \ \text{mk-rec-env-def}\)

apply auto

apply rule

apply rule

apply \(\langle\text{auto intro: fmdomI}\rangle\)

done

have \(\text{fmpred (\lambda. \ \text{welldefined}') (fmupd (\text{Name} \ n) \ v_2 \ (\Gamma_\Lambda \ ++f \ \text{mk-rec-env} \ \text{css} \ \Gamma_\Lambda)}\)

259
using \( (\text{welldefined}' \ v_1) \ (\text{welldefined}' \ v_2) \)
using \( \langle \text{fmlookup css} \ (\text{Name name}) = \text{Some cs} \rangle \)
unfolding \( (v_1 \rightarrow \text{mk-rec-env-def}) \)
apply auto
apply rule
apply rule
apply \( (\text{auto intro: \text{fdomI}}) \)
done

have \( \text{closed-except} \ (\text{Sabs cs}) \ (\text{fdomm} \ (\text{fmupd} \ (\text{Name n}) \ v_2 \ \Gamma \Lambda)) \)
using \( (\text{vclosed} \ v_1) \ \text{unfolding} \ (v_1 = \_0) \)
apply \( (\text{auto simp: \text{Sterm.closed-except-simps list-all-iff}}) \)
using \( \langle \text{fmlookup css} \ (\text{Name name}) = \text{Some cs} \rangle \)
apply \( (\text{auto simp: \text{closed-except-def dest \_1: \text{fmpredD[where m = css]}}}) \)
done

have \( \text{consts} \ (\text{Sabs cs}) \ |\subseteq| \ \text{fdomm} \ (\text{fmupd} \ (\text{Name n}) \ v_2 \ \Gamma \Lambda) \ |\cup| \ (C |\cup| \ \text{fdom css}) \)
using \( (\text{vwelldefined}' \ v_1) \ \text{unfolding} \ (v_1 = \_0) \)
unfolding \( \text{seconsts-sabs} \)
using \( \langle \text{fmlookup css} \ (\text{Name name}) = \text{Some cs} \rangle \)
by \( (\text{auto simp: list-all-iff list-ex-iff}) \)

have \( \neg \text{shadows-consts} \ (\text{Sabs cs}) \)
using \( (\text{not-shadows-vcnsts} \ v_1) \ \text{unfolding} \ (v_1 = \_0) \)
using \( \langle \text{fmlookup css} \ (\text{Name name}) = \text{Some cs} \rangle \)
by \( (\text{auto simp: list-all-iff list-ex-iff}) \)

have \( \text{fdisjnt} \ C \ (\text{fdomm} \ \Gamma \Lambda) \)
using \( (\text{vwelldefined}' \ v_1) \ \text{unfolding} \ (v_1 = \_0) \)
using \( \langle \text{fmlookup css} \ (\text{Name name}) = \text{Some cs} \rangle \)
by auto

have \( \text{if-rval} \ (\lambda ml-v. \ \exists v. \ \text{fmupd} \ (\text{Name n}) \ v_2 \ (\Gamma \Lambda ++ f \ \text{mk-rec-env css} \ \Gamma \Lambda)) \)
\(+v \ \text{Sabs cs} \ \_0 \ \text{Svar} \ (\text{Name n}) \ \_1 \ v \ \land \ \text{related-v v ml-v} \) \( \text{bv} \)
proof \( (\text{rule \text{appI(2)}}) \)
have \( \text{fmrel-on-fset} \ (\text{ids} \ (\text{Sabs cs})) \ \text{related-v} \ \Gamma \Lambda \ (\text{fmap-of-ns} \ (\text{sem-env.v env}_\Lambda)) \)
apply \( (\text{rule \text{fmrel-on-fsubset}}) \)
apply fact
apply \( (\text{subst \text{union-image-bind-eq}[symmetric]}) \)
apply \( (\text{rule \text{ffUnion-subset-elem}}) \)
apply \( (\text{subst \text{fimage-iff}}) \)
apply \( (\text{rule \text{fBexI}}) \)
apply simp
apply \( (\text{rule \text{fmranI}}) \)
apply fact
done
have \( \text{fmrel-on-fset} \ (\text{ids} \ (\text{Sabs} \ cs)) \ \text{related-v} \ (\text{mk-rec-env} \ css \ \Gamma_A) \)
\((\text{cake-mk-rec-env} \ \text{funs} \ \text{env}_A)\)

apply rule
apply (rule \( \text{mk-rec-env-related} \ (\text{THEN} \ \text{fmrelD}) \))
apply (rule \( \text{fmrel} \ - \ css \ - \))
apply (rule \( \text{fmrel-on-fset} \ (\text{fbind} \ - \ -) \ \text{related-v} \ \Gamma_A \ - \))
done

show \( \text{fmrel-on-fset} \ (\text{ids} \ (\text{Sabs} \ cs \ S) S S \ \text{Svar} \ (\text{Name} \ n)) \)
\(\text{related-v} \ (\text{fmupd} \ (\text{Name} \ n) \ v_2 \ (\Gamma_A ++ f \ \text{mk-rec-env} \ css \ \Gamma_A)) \ (\text{fmap-of-ns} \ (\text{sem-env.} \ v \ \text{env}')) \)

unfolding \( \text{reclosure'} \)
apply (simp del: \( \text{frees-sterm}. \ \text{simps}(\text{3}) \ \text{consts-sterm}. \ \text{simps}(\text{3}) \ \text{name}. \ \text{sel} \))

add: \( \text{ids-def split}: \ \text{sem-env}. \ \text{splits} \)

apply (rule \( \text{fmrel-on-fset-updateI} \))
unfolding \( \text{build-rec-env-fmap} \)
apply (rule \( \text{fmrel-on-fset-addI} \))
apply (fold \( \text{ids-def} \))
subgoal
using \( \text{fmrel-on-fset} \ (\text{ids} \ (\text{Sabs} \ cs)) \ \text{related-v} \ \Gamma_A \ - \) by simp
subgoal
using \( \text{fmrel-on-fset} \ (\text{ids} \ (\text{Sabs} \ cs)) \ \text{related-v} \ (\text{mk-rec-env} \ \text{css} \ \Gamma_A) \ - \) by simp

apply (rule \( \text{related-v} \ v_2 \ ml-v_2') \)
done

next
show \( \text{wellformed} \ (\text{Sabs} \ cs \ S S S \ \text{Svar} \ (\text{Name} \ n)) \)
using : \( \text{wellformed} \ v_1 \); unfolding \( v_1 = - \)
using \( \text{fmlookup css} \ (\text{Name} \ name) = \text{Some cs} \)
by auto

next
show \( \text{related-exp} \ (\text{Sabs} \ cs \ S S S \ \text{Svar} \ (\text{Name} \ n)) \ \text{exp}' \)
unfolding \( \text{exp}' = - \)
apply (rule \( \text{related-exp}. \ \text{intros} \))
apply fact
apply (rule \( \text{related-exp}. \ \text{intros} \))
done

next
show \( \text{closed-except} \ (\text{Sabs} \ cs \ S S S \ \text{Svar} \ (\text{Name} \ n)) \ \text{fdom} \ (\text{fmupd} \ (\text{Name} \ n) \ v_2 \ (\Gamma_A ++ f \ \text{mk-rec-env} \ css \ \Gamma_A)) \)
using : \( \text{closed-except} \ (\text{Sabs} \ cs) \ (\text{fdom} \ (\text{fmupd} \ (\text{Name} \ n) \ v_2 \ \Gamma_A)) \)
by \( \text{auto simp}: \ \text{list-all-iff closed-except-def} \)

next
show \( \neg \ \text{shadows-consts} \ (\text{Sabs} \ cs \ S S S \ \text{Svar} \ (\text{Name} \ n)) \)
using \( \neg \ \text{shadows-consts} \ (\text{Sabs} \ cs): \ (\text{Name} \ n \ | \mathcal{E}| \ all-consts) \ \text{by simp} \)

next
show \( \text{consts} \ (\text{Sabs} \ cs \ S S S \ \text{Svar} \ (\text{Name} \ n)) \ \subseteq \ \text{fdom} \ (\text{fmupd} \ (\text{Name} \ n) \ v_2 \ (\Gamma_A ++ f \ \text{mk-rec-env} \ css \ \Gamma_A)) \ | \mathcal{U} | \ C \)
using : \( \text{consts} \ (\text{Sabs} \ cs) \ \subseteq \ - \) unfolding \( \text{mk-rec-env-def} \)
by auto
next
  show fdisjnt C (fmdom (fmupd (Name n) v2 (ΓLambda ++ f mk-rec-env css ΓLambda))) using ⟨Name n |{感兴趣}⟩ all-consts) (fdisjnt C (fmdom ΓLambda) ⟨vwelldefined′⟩)
  unfolding mk-rec-env-def ⟨v1 = -⟩
  by (auto simp: fdisjnt-alt-def all-consts-def)
qed fact+
then obtain v
where fmupd (Name n) v2 (ΓLambda ++ f mk-rec-env css ΓLambda) ⊢ v Sabs cs $ Svar (Name n) ↓ v related-v v ml-v
unfolding ⟨bv = -⟩
by auto
then obtain env pat rhs
where vfind-match cs v2 = Some (env, pat, rhs)
and fmupd (Name n) v2 (ΓLambda ++ f mk-rec-env css ΓLambda) ++ f env ⊢ rhs
↓ v
by (auto elim: veval′-sabs-svarE)
hence (pat, rhs) ∈ set cs vmatch (mk-pat pat) v2 = Some env
by (metis vfind-match-elem)
linear pat wellformed rhs
using ⟨vwellformed v1⟩ unfolding ⟨v1 = -⟩
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) v2 = Some env)
done
obtain v' where ΓLambda ++ f mk-rec-env css ΓLambda ++ f env ⊢ v' rhs ↓ v' v' ≈c v
proof (rule veval′-agree-eq)
  show fmupd (Name n) v2 (ΓLambda ++ f mk-rec-env css ΓLambda) ++ f env ⊢ v
  rhs ↓ v by fact
next
have *: Name n |{感兴趣}⟩ ids rhs if Name n |{感兴趣}⟩ fmdom env
proof
  assume Name n |{感兴趣}⟩ ids rhs
  thus False
  unfolding ids-def
  proof (cases rule: funion-strictE)
    case A
    moreover have Name n |{感兴趣}⟩ frees pat
      using that unfolding (fmdom env = frees pat).
    ultimately have Name n |{感兴趣}⟩ frees (Sabs cs)
  qed
qed

262
apply auto
unfolding ffUnion-alt-def
apply simp
apply (rule fBexI[where \(x = (\text{pat}, \text{rhs})\)])
apply (auto simp: fset-of-list-elem)
apply (rule \((\text{pat}, \text{rhs}) \in \text{set cs}\))
done
thus \(?thesis\)
  using \((\text{Name \(n\)} \mid |) \text{ids} (\text{Sabs \(cs\)})\): unfolding ids-def
by blast

next
case B
hence Name \(n\) \(\mid \in\) \(\text{consts} (\text{Sabs \(cs\)})\)
  apply auto
unfolding ffUnion-alt-def
apply simp
apply (rule fBexI[where \(x = (\text{pat}, \text{rhs})\)])
apply (auto simp: fset-of-list-elem)
apply (rule \((\text{pat}, \text{rhs}) \in \text{set cs}\))
done
thus \(?thesis\)
  using \((\text{Name \(n\)} \mid |) \text{ids} (\text{Sabs \(cs\)})\): unfolding ids-def
by blast

qed

show fmrel-on-fset \((\text{ids} \text{rhs})\) erelated \((\Gamma_\Lambda \text{++f mk-rec-env} \text{css} \Gamma_\Lambda) \text{++f env}\)
apply rule
apply auto
apply (rule \text{option.rel-refl}; \text{rule erelated-refl})
using * apply auto]
apply (rule \text{option.rel-refl}; \text{rule erelated-refl}+)
using * apply auto]
apply (rule \text{option.rel-refl}; \text{rule erelated-refl}+)
done
next
show closed-venv \((\text{fmupd} \text{Name \(n\)} \text{v} \text{2} (\Gamma_\Lambda \text{++f mk-rec-env} \text{css} \Gamma_\Lambda) \text{++f env})\)
apply rule
apply fact
apply \text{rule \text{vclosed.vmatch-env}}
apply fact
apply fact
done
next
show wellformed-venv \((\text{fmupd} \text{Name \(n\)} \text{v} \text{2} (\Gamma_\Lambda \text{++f mk-rec-env} \text{css} \Gamma_\Lambda) \text{++f env})\)
apply rule
apply fact
apply (rule vwellformed.ematch-env)
apply fact
apply fact
done

next
  show closed-except rhs (fndom (fmupd (Name n) v2 (ΓΛ ++f
mk-rec-env css ΓΛ)) ++f env))
  using (fndom env = frees pat) (pat, rhs) ∈ set cs
  using (closed-except (Sabs cs) (fmupd (Name n) v2 ΓΛ))
  apply (auto simp: Stern.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 |⊆| fndom (fmupd (Name n) v2 (ΓΛ ++f mk-rec-env
css ΓΛ)) ++f env) |∪| C
  using :consts (Sabs cs) |⊆| (pat, rhs) ∈ set cs
  unfolding sconsts-sabs mk-rec-env-def
  by (auto simp: list-all-iff)

next
  have fdisjnt C (fndom env)
  using (pat, rhs) ∈ set cs; (~ shadows-consts (Sabs cs))
  unfolding (fndom env = frees pat)
  by (auto simp: list-ex-iff all-consts-def fdisjnt-alt-def)
  moreover have fdisjnt C (fndom css)
  using :vwelldefined' v1 unfolding (v1 = -)
  by simp
  ultimately show fdisjnt C (fndom (fmupd (Name n) v2 (ΓΛ ++f
mk-rec-env css ΓΛ)) ++f env))
  using :Name n |∅| all-consts; fdisjnt C (fndom ΓΛ);
  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-veconsts-env env
  by (rule not-shadows-veconsts.ematch-env) fact+
  thus not-shadows-veconsts-env (fmupd (Name n) v2 (ΓΛ ++f mk-rec-env
css ΓΛ) ++f env)
  using (not-shadows-veconsts-env (fmupd (Name n) v2 (ΓΛ ++f
mk-rec-env css ΓΛ))); by blast

next
  have fnpred (λ-. vwelldefined') env
  by (rule ematch-welldefined') fact+
thus \( \text{fmpred} (\lambda. \text{vwelldefined}') (\text{fmapd} (\text{Name} n) v_2 (\Gamma \Lambda + + f \text{mk-rec-env css} \Gamma \Lambda)) + + f \text{ env} \)

using \( \text{fmpred} (\lambda. \text{vwelldefined}') (\text{fmapd} (\text{Name} n) v_2 (\Gamma \Lambda + + f \text{mk-rec-env css} \Gamma \Lambda)) \) by blast

qed simp

show ?thesis
apply (intro exI conjI)
unfolding \( t = - \)
apply (rule \( \text{veval',rec-comb} \))
using \( \Gamma \vdash v_1 \downarrow v_1 \) unfolding \( v_1 = - \) apply blast
apply fact+
apply (rule related-v-ext)
apply fact+
done

qed

next

case \( \text{mat1 env ml-scr ml-scr'} ml-cs ml-rhs ml-pat env' ml-res' \)

obtain scr cs
where \( t = \text{Subs cs \$}, \text{scr related-exp scr ml-scr} \)
and list-all2 (rel-prod related-pat related-exp) cs ml-cs
using (related-exp \( t (\text{Mat ml-scr ml-cs}) \))
by cases

have \( \text{sem-env.e env} = \text{as-static-cenv} \)
using (is-cupcake-all-env env)
by (auto elim: is-cupcake-all-envE)

obtain scr' where \( \Gamma \vdash v \downarrow \text{scr'} \text{ related-v scr'} ml-scr' \)
using mat1(4) unfolding if-real.simps

proof
show \( \text{is-cupcake-all-env env related-exp scr ml-scr wellformed-venv \( \Gamma \)} \)
closed-venv \( \Gamma \) \( \text{fmpred} (\lambda. \text{vwelldefined}') \) \( \Gamma \) \( \text{fdisjnt} \) \( C \) \( \text{fdom} \) \( \Gamma \)
not-shadows-consts-venv \( \Gamma \)
by fact+

next
show \( \text{fmrel-on-fset} (\text{ids scr}) \text{ related-v \( \Gamma \)} \) \( \text{fmap-of-ns} (\text{sem-env.v env}) \)
apply (rule fmrel-on-fsubset)
apply fact
unfolding \( t = - \) ids-def
apply auto
done

next
show \( \text{wellformed scr cons} \text{ const} \text{ scr} |\subseteq| \text{fdom} \) \( \Gamma |\cup| C \)
closed-except scr \( \text{fdom} \) \( \Gamma \) \( \neg \text{shadows-consts scr} \)
using mat1 unfolding \( t = \cdot \) by \((\text{auto simp: closed-except-def})\)

qed blast

have list-all \((\lambda(pat, \cdot). \text{linear pat})\) cs
  using mat1 unfolding \( t = \cdot \) by \((\text{auto simp: list-all-iff})\)

obtain rhs\(\cdot\) pat \(\Gamma'\)
  where ml-pat = mk-ml-pat \((\text{mk-pat pat})\) related-exp rhs ml-rhs
  and vfind-match cs scr' = Some \((\Gamma',\text{pat},\text{rhs})\)
  and var-env \(\Gamma'\) env'
  using \((\text{list-all2 - cs ml-cs})\) \((\text{list-all - cs})\) \((\text{related-v ser'}\cdot\text{ml-ser'}\cdot\text{mat1(2)})\)
  unfolding \((\text{sem-env.c env} = \text{as-static-cenv})\)
  by \((\text{auto elim: match-all-related})\)

hence vmatch \((\text{mk-pat pat})\) scr' = Some \(\Gamma'\)
  by \((\text{auto dest: vfind-match-elem})\)

hence patvars \((\text{mk-pat pat})\) = \(\text{fmdom} \Gamma'\)
  by \((\text{auto simp: vmatch-dom})\)

have \((\text{pat, rhs})\) \(\in\) set cs
  by \((\text{rule vfind-match-elem})\) \text{fact}

have linear pat
  using \((\text{pat, rhs})\) \(\in\) set cs \(\langle\text{wellformed}\rangle\) unfolding \(t = \cdot\)
  by \((\text{auto simp: list-all-iff})\)

hence fmdom \(\Gamma'\) = \(\text{frees pat}\)
  by \((\text{simp add: mk-pat-frees})\)

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

obtain rhs' where \(\Gamma \vdash_{v} \Gamma' \vdash_{v} \text{rhs} \downarrow \text{rhs}' \text{related-v \hspace{1em} rhs'} \text{ml-rhs}'\)
  using mat1(5) unfolding \(\langle\text{ml-res = \cdot}\rangle\) \(\text{if-rval.simps}\)
proof
  show is-cupcake-all-env \((\text{env} \parallel \text{sem-env.v} := \text{nsAppend (alist-to-ns env')})\)
    \((\text{sem-env.v env}) \parallel\)
  proof (rule cupcake-v-update-preserve)
    have list-all \((\text{is-cupcake-value o snd})\) env'
      proof (rule match-all-preserve)
        show cupcake-c-ns \((\text{sem-env.c env})\)
          unfolding \((\text{sem-env.c env}) = \cdot\) by \((\text{rule static-cenv})\)
      next
      have is-cupcake-exp \((\text{Mat ml-scr ml-cs})\)
        apply \((\text{rule related-exp-is-cupcake})\)
        using mat1 by auto
      thus cupcake-clauses ml-cs
    
  qed

266
by simp

show is-cupcake-value ml-scr'
  apply (rule cupcake-single-preserve)
  apply (rule mat1)
  apply (rule mat1)
  using (is-cupcake-exp (Mat ml-scr ml-cs)) by simp
qed fact+

hence is-cupcake-ns (alist-to-ns env')
  by (rule cupcake-alist-to-ns-preserve)
moreover have is-cupcake-ns (sem-env.v env)
  by (rule is-cupcake-all-envD) fact
ultimately show is-cupcake-ns (nsAppend (alist-to-ns env') (sem-env.v env))
  by (rule cupcake-nsAppend-preserve)
qed fact

next
show related-exp rhs ml-rhs
  by fact
next

have *: fmdom (fmap-of-list (map (map-prod Name id) env')) = fmdom Γ'
  using (var-env Γ' env')
  by (metis fnrel-fmdom-eq)

have **: id |∈| ids t if id |∈| ids rhs id |∉| fmdom Γ' for id
  using (id |∈| ids rhs) unfolding ids-def
proof (cases rule: funion-strictE)
  case A
  from that have id |∉| frees pat
    unfolding (fmdom Γ' = frees pat) by simp
  hence id |∈| frees t
    using (pat, rhs) ∈ set cs
    unfolding (t = -)
    apply auto
    apply (subst ffUnion-alt-def)
    apply simp
    apply (rule fBexI[where x = (pat, rhs)])
    using A apply (auto simp: fset-of-list-elem)
    done
  thus id |∈| frees t |∪| consts t by simp
next
  case B
  hence id |∈| consts t
    using (pat, rhs) ∈ set cs
    unfolding (t = -)
    apply auto
    apply (subst ffUnion-alt-def)
    apply simp

267
apply (rule fBexI[where \( x = (\text{pat, rhs}) \)])
apply (auto simp: fset-of-list-elem)
done
thus \( id \mid \in \mid \frees t \mid \cup \mid \consts t \) by simp
qed

have \( \text{fmrel-on-fset } (\text{ids rhs}) \) related-v \((\Gamma ++_f \Gamma') \) \((\text{fmap-of-ns } (\text{sem-env.v env}) ++_f \text{fmap-of-list } (\text{map } (\text{map-prod Name id}) \text{ env}' )) \)
apply rule
apply simp
apply safe
subgoal
apply (rule fmrelD)
apply (rule \( \langle \text{var-env \Gamma' env' } \rangle \))
done
subgoal using * by simp
subgoal using *
by (metis \( \text{no-types, hide-lams} \) \( \text{comp-def fimageI fdom-fmap-of-list fset-of-list-map fst-comp-map-prod} \))
subgoal using **
by (metis \( \text{fmlookup-ns fmrel-on-fsetD mat1.prems(2)} \))
done
thus \( \text{fmrel-on-fset } (\text{ids rhs}) \) related-v \((\Gamma ++_f \Gamma') \) \((\text{fmap-of-ns } (\text{sem-env.v env}) \) \((\text{env } \) \text{sem-env.v := nsAppend } (\text{alist-to-ns env' }) \text{ (sem-env.v env' }) \)\)\)
by (auto split: \( \text{sem-env.splits} \))
next
show \( \text{wellformed-venv } (\Gamma ++_f \Gamma') \)
apply rule
apply fact
apply (rule \( \text{vwellformed.vmatch-env} \))
apply fact
apply (rule \( \text{veval'-wellformed} \))
apply fact
using mat1 unfolding \( t = \cdot \) by auto
next
show \( \text{closed-venv } (\Gamma ++_f \Gamma') \)
apply rule
apply fact
apply (rule \( \text{vclosed.vmatch-env} \))
apply fact
apply (rule \( \text{veval'-closed} \))
apply fact
using mat1 unfolding \( t = \cdot \) by (auto simp: \( \text{closed-except-def} \))
next
show \( \text{fmpred } (\lambda -. \text{vwelldefined'}) \) \((\Gamma ++_f \Gamma') \)
apply rule
apply fact
apply (rule \( \text{vmatch-welldefined'} \))
apply \textit{fact}
apply (\textit{rule veval'-welldefined'})
apply \textit{fact}

\textbf{using \textit{mat1 unfolding}} \langle t = \_ \rangle \textbf{by \textit{auto}}

next
show \textit{not-shadows-vconsts-env} \((\Gamma ++ f \Gamma')\)
apply \textit{rule}
apply \textit{fact}
apply \textit{rule not-shadows-vconsts.vmatch-env}
apply \textit{fact}
apply \textit{rule veval'-shadows}
apply \textit{fact}

\textbf{using \textit{mat1 unfolding}} \langle t = \_ \rangle \textbf{by \textit{auto}}

next
show \textit{wellformed rhs}
using \langle (\textit{pat}, \textit{rhs}) \in \textit{set cs} \rangle \langle \textit{wellformed t} \rangle
unfolding \langle t = \_ \rangle
by (\textit{auto simp: list-all-iff})

next
show \textit{closed-except rhs} \((\textit{fdom} \ (\Gamma ++ f \Gamma'))\)
apply \textit{simp}
unfolding \langle \textit{fdom} \ (\Gamma' = \textit{frees pat}) \rangle
using \langle (\textit{pat}, \textit{rhs}) \in \textit{set cs} \rangle \langle \textit{closed-except t} \ (\textit{fdom} \ \Gamma) \rangle
unfolding \langle t = \_ \rangle
by (\textit{auto simp: Sterm.closed-except-simps list-all-iff})

next
have \textit{consts} \((\textit{Sabs \ cs}) \subseteq \textit{fdom} \ (\Gamma' \cup C)\)
using \textit{mat1 unfolding} \langle t = \_ \rangle \textbf{by \textit{auto}}

next
show \textit{consts rhs} \((\textit{fdom} \ (\Gamma ++ f \Gamma')) \cup C\)
apply \textit{simp}
unfolding \langle \textit{fdom} \ (\Gamma' = \textit{frees pat}) \rangle
using \langle (\textit{pat}, \textit{rhs}) \in \textit{set cs} \rangle \langle \textit{consts} \ (\textit{Sabs \ cs}) \subseteq \_ \rangle
unfolding \textit{sconsts-sabs}
by (\textit{auto simp: list-all-iff})

next
have \textit{fdisjnt C} \((\textit{fdom} \ \Gamma')\)
unfolding \langle \textit{fdom} \ (\Gamma' = \textit{frees pat}) \rangle
using \langle \textit{shadows-consts t} \ (\textit{pat}, \textit{rhs}) \in \textit{set cs} \rangle
unfolding \langle t = \_ \rangle
by (\textit{auto simp: list-ex-iff fdisjnt-alt-def all-consts-def})

\textbf{thus \textit{fdisjnt C} \((\textit{fdom} \ (\Gamma ++ f \Gamma'))\)}
\textbf{using \textit{fdisjnt C} \((\textit{fdom} \ \Gamma')\)}
\textbf{unfolding \textit{fdisjnt-alt-def} \textbf{by \textit{auto}}}

next
show \textit{shadows-consts rhs}
using \langle (\textit{pat}, \textit{rhs}) \in \textit{set cs} \rangle \langle \textit{shadows-consts t} \rangle
unfolding \langle t = \_ \rangle
by (\textit{auto simp: list-ex-iff})

\textbf{qed \textit{blast}}

\textbf{show} \(\exists t'. \ \Gamma \vdash_v t \downarrow t' \land \textit{related-v} \ t' \ ml-rhs'

269
unfolding \( (t = -) \)
apply (intro exI conjI seval.intros)
apply (rule seval'.intros)
apply (rule seval'.intros)
apply fact+
done
qed
qed auto

theorem semantic-correctness:
fixes exp
assumes cupcake-evaluate-single env exp (Rval ml-v) is-cupcake-all-env env
assumes fmrel-on-fset (ids t) related-v \( \Gamma \) (fmap-of-ns (sem-env.v env))
assumes related-exp t exp
assumes wellformed t wellformed-venv \( \Gamma \)
assumes closed-venv \( \Gamma \) closed-except t (fmdom \( \Gamma \))
assumes fmpred (\( \lambda \cdot \) vwelldefined') \( \Gamma \) consts t \( \subseteq \) fmdom \( \Gamma \) \( \cup \) C
assumes fdisjnt C (fmdom \( \Gamma \))
assumes \( \neg \) shadows-consts t not-shadows-vconsts-env \( \Gamma \)
obtains v where \( \Gamma \vdash v \downarrow v \) related-v v ml-v
using semantic-correctness0[OF assms]
by auto
end end

5.4 Converting bytes to integers

theory CakeML-Byte
imports
  CakeML.Evaluate-Single
  Show.Show-Instances
begin

definition pat :: Ast.pat where
pat = Ast.Pcon (Some (Short "String-char-Char")) (map (\( \lambda \cdot \) Ast.Pvar ("b" @ show n)) [0..<8])

definition cake-plus :: exp \Rightarrow exp \Rightarrow 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 \Rightarrow exp \Rightarrow exp where
\textit{lemma} \textit{cake-times-correct}: 
\begin{align*}
\text{assumes} & \quad \text{evaluate env } s \ u = (s', \text{Real } (\text{IntLit } y)) \\
\text{assumes} & \quad \text{evaluate env } s' \ t = (s'', \text{Real } (\text{IntLit } x)) \\
\text{shows} & \quad \text{evaluate env } s \ (\text{cake-times } t \ u) = (s'', \text{Real } (\text{IntLit } (x \ast y)))
\end{align*}
\text{unfolding} \textit{cake-times-def} \text{ using} \textit{assms} \text{ by} \textit{simp}

definition \textit{cake-int-of-bool} :: exp \Rightarrow exp \text{ where} 
\text{cake-int-of-bool } e = \text{Ast.Mat } e \\
& \quad ([\text{Ast.Pcon } (\text{Some } (\text{Short } "\text{HOL-False}")) [], \text{Lit } (\text{IntLit } 0)), \\
& \quad (\text{Ast.Pcon } (\text{Some } (\text{Short } "\text{HOL-True}")) [], \text{Lit } (\text{IntLit } 1))]

definition \textit{summands} :: exp \text{ list} \text{ where} 
\text{summands} = \text{map } (\lambda n. \text{cake-times } (\text{Lit } (\text{IntLit } (2 \ast n)))) (\text{cake-int-of-bool } (\text{Ast.Var } (\text{Short } ("b" @ \text{show } n)))) [0..<8]

definition \textit{cake-int-of-byte} :: exp \text{ where} 
\text{cake-int-of-byte} = 
\text{Ast.Fun } "x" (\text{Ast.Mat } (\text{Ast.Var } (\text{Short } "x")) \text{ [[pat, foldl cake-plus } (\text{Lit } (\text{IntLit } 0)) \text{ summands]}))

\textit{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 → nterm → pterm → sterm

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

lemma (in prules) prewrite-closed:
  assumes rs ⊢ p t → t' closed t
  shows closed t'
using assms proof induction
  case (step name rhs)
    thus ?case
      using all-rules by force
next
  case (beta c)
    obtain pat rhs where c = (pat, rhs) by (cases c) auto
  with beta have closed-except rhs (frees pat)
    by (auto simp: closed-except-simps)
  show ?case

272
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 |- p t -->* t' closed t`
shows `closed t'`
using `assms`
by induction (auto intro: prewrite-closed)

corollary (in irules) compile-correct-rt:
assumes `Rewriting-Pterm.compile rs |- p t -->* t' finished rs`
shows `rs |- i t -->* t'`
using `assms proof (induction rule: rtranclp-induct)`
case step
thus `?case`
by (meson compile-correct rtranclp.simps)
qed auto

6.1.2 Reflexive-transitive closure of `prules.compile-correct`.

lemma (in prules) compile-correct-rt:
assumes `Rewriting-Sterm.compile rs |- p s --> s'`
shows `rs |- i s -->* s' wellformed s`
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) \in set rs`
using `assms by induct auto`

lemma (in srules) srewrite-wellformed:
assumes `rs |- s t --> t' wellformed t`
shows `wellformed t'`
using `assms proof induction`
case (step name rhs)
hence `(name, rhs) \in 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) \in 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 \((\text{pat}, \text{rhs}) \in \text{set cs}\) beta by (auto simp: list-all-iff)
next
  show wellformed-env env
    using \(\text{match pat t = Some env}\) beta
    by (auto intro: wellformed.match)
qed

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 \(\text{compile} \vdash_n u \longrightarrow u'\) closed \(u\)
  shows \(\text{rs} \vdash \text{nterm-to-term}' u \longrightarrow^* \text{nterm-to-term}' u'\)
using assms
proof induction
  case (step \(u' u''\))
  hence \(rs \vdash \text{nterm-to-term}' u \longrightarrow^{*} \text{nterm-to-term}' u'\)
    by auto
  also have \(rs \vdash \text{nterm-to-term}' u' \longrightarrow \text{nterm-to-term}' u''\)
    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 \(\text{transform-irule-set} rs \vdash_i u \longrightarrow^* u'' t \approx_p u\) 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' ≈ p u')
apply (rule irwre-rt-closed)
apply (rule rs ⊢ t −→∗ t')
apply (rule closed t)
apply blast
done

show ?case
apply (rule step.prems)
apply (rule rtranclp-trans)
apply fact+
done
qed blast
corollary (in irules) transform-correct-rt-no-abs:
  assumes transform-irule-set rs ⊢ t −→∗ u closed t no-abs u
  shows rs ⊢ t −→∗ u
proof –
  have t ≈ p t by (rule prelated-refl)
  obtain t' where rs ⊢ t −→∗ t' t' ≈ p u
    apply (rule transform-correct-rt)
    apply (rule assms)
    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 ⊢ t −→∗ u closed t no-abs u
  shows rs ⊢ t −→∗ u
using assms(1,2) proof (induction n arbitrary: rs)
  case (Suc n)
  interpret irules C rs by fact
  show ?case
    apply (rule transform-correct-rt-no-abs)
    apply (rule Suc.IH)
    apply (rule rules-transform)
    using Suc(3) apply (simp add: funpow-swap1)
    apply fact+
done
qed auto
corollary (in irules) transform-correct-rt-n-no-abs:
  assumes (transform-irule-set ^ n) rs ⊢ t −→∗ u closed t no-abs u
  shows rs ⊢ t −→∗ u
by (rule transform-correct-rt-n-no-abs0) (rule irules-axioms assms)
6.1.5 Iterated application of \textit{transform-irule-set}.

\textbf{definition} \textit{max-arity :: irule-set ⇒ nat where} \[\text{max-arity } rs = \text{fMax } ((\text{arity } \circ \text{snd}) \mid \cdot \mid rs)\]

\textbf{lemma} \textit{rules-transform-iter0}:
\begin{itemize}
  \item \textbf{assumes} \textit{rules C-info rs}
  \item \textbf{shows} \textit{rules C-info \((\text{transform-irule-set} ^\cdot n)\) rs}
\end{itemize}
\textbf{using} \textit{assms}
\textbf{by} (induction \(n\)) (auto intro: \textit{irules.rules-transform del: irulesI})

\textbf{lemma} \textit{(in irules) rules-transform-iter: irules C-info \((\text{transform-irule-set} ^\cdot n)\) rs}
\textbf{by} (rule rules-transform-iter0) (rule irules-axioms)

\textbf{lemma} \textit{transform-irule-set-n-heads: fst \mid \cdot \mid (\text{transform-irule-set} ^\cdot n) rs = fst \mid \cdot \mid rs}
\textbf{by} (induction \(n\)) (auto simp: transform-irule-set-heads)

\textbf{hide-fact} \textit{rules-transform-iter0}

\textbf{definition} \textit{transform-irule-set-iter :: irule-set ⇒ irule-set where} \[\text{transform-irule-set-iter } rs = (\text{transform-irule-set} ^\cdot \text{max-arity } rs) \cdot rs\]

\textbf{lemma} \textit{transform-irule-set-iter-heads: fst \mid \cdot \mid \text{transform-irule-set-iter } rs = fst \mid \cdot \mid rs}
\textbf{unfolding} \textit{transform-irule-set-iter-def by} (simp add: transform-irule-set-n-heads)

\textbf{lemma} \textit{(in irules) finished-alt-def: finished } rs \longleftrightarrow \textit{max-arity } rs = 0
\textbf{proof}
\begin{itemize}
  \item \textbf{assume} \textit{max-arity } rs = 0
  \item \textbf{hence} ~ \textit{fBex } ((\text{arity } \circ \text{snd}) \mid \cdot \mid rs) (\lambda x. 0 < x)
  \item \textbf{using} \textit{nonempty}
  \item \textbf{unfolding} \textit{max-arity-def}
  \item \textbf{by} (metis \textit{fBex-fempty fmax-ex-gr not-less0})
  \item \textbf{thus} \textit{finished } rs
  \item \textbf{unfolding} \textit{finished-def}
  \item \textbf{by} \textit{force}
\end{itemize}
\textbf{next}
\begin{itemize}
  \item \textbf{assume} \textit{finished } rs
  \item \textbf{have} \textit{fMax } ((\text{arity } \circ \text{snd}) \mid \cdot \mid rs) \leq 0
  \item \textbf{proof} (rule \textit{fMax-le})
  \item \textbf{show} \textit{fBall } ((\text{arity } \circ \text{snd}) \mid \cdot \mid rs) (\lambda x. x \leq 0)
  \item \textbf{using} \textit{(finished } rs\text{)} \textit{unfolding} \textit{finished-def by} \textit{force}
\end{itemize}
\textbf{next}
\begin{itemize}
  \item \textbf{show} (\text{arity } \circ \text{snd}) \mid \cdot \mid rs \neq \{\}
  \item \textbf{using} \textit{nonempty by} \textit{force}
\end{itemize}

276
thus \( \text{max-arity } rs = 0 \)

unfolding \( \text{max-arity-def} \) by simp

lemma (in irules) transform-finished-id: finished \( rs \implies \text{transform-irule-set } rs = rs \)
unfolding transform-irule-set-def finished-def transform-irules-def map-prod-def id-apply
by (rule fset-map-snd-id) (auto simp: fmember_rep_eq elim!: fBallE)

lemma (in irules) max-arity-decr: \( \text{max-arity } (\text{transform-irule-set } rs) = \text{max-arity } rs - 1 \)
proof (cases finished \( rs \))
  case True
  thus \(?\)thesis
  by (auto simp: transform-finished-id finished-alt-def)
next
  case False
  have \((\text{arity } \circ \text{snd}) \mid \mid \text{transform-irule-set } rs = (\lambda x. x - 1) \mid \mid (\text{arity } \circ \text{snd}) \mid \mid \)
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 \((\text{name, } irs) \in fset rs\)
  hence \((\text{name, } irs) \in rs\)
    by (simp add: fmember_rep_eq)
  hence \(\text{arity-compatibles } irs \text{ } irs \neq \{\}\)
    using nonempty inner by (blast dest: fpairwiseD)+
  thus \(\text{arity } (\text{transform-irules } irs) = \text{arity } irs - 1\)
    by (simp add: arity-transform-irules)
qed

hence \(\text{max-arity } (\text{transform-irule-set } rs) = \text{fMax } ((\lambda x. x - 1) \mid \mid (\text{arity } \circ \text{snd}) \mid \mid rs)\)
unfolding max-arity-def by simp
also have \(= \text{fMax } ((\text{arity } \circ \text{snd}) \mid \mid rs) - 1\)
proof (rule fmax-decr)
  show \(\text{fBex } ((\text{arity } \circ \text{snd}) \mid \mid rs) (\leq 1)\)
    using False unfolding finished-def by force
qed

finally show \(?\)thesis
  unfolding max-arity-def
  by simp
  
qed

lemma max-arity-decr'0:
  assumes irules \( C \) \( rs \)
  shows \(\text{max-arity } ((\text{transform-irule-set } ^ n \) \( rs) = \text{max-arity } rs - n\)
proof (induction \( n \))
  case (Suc \( n \))

277
show ?case
apply simp
apply (subst irules.max-arity-decr)
using Suc assms by (auto intro: irules.rules-transform-iter del: irulesI)
qed auto

lemma (in irules) max-arity-decr': max-arity ((transform-irule-set \^ n) rs) = max-arity rs − n
by (rule max-arity-decr'0) (rule irules-axioms)

hide-fact max-arity-decr'0

lemma (in irules) transform-finished: finished (transform-irule-set-iter rs)
unfolding transform-irule-set-iter-def
by (auto simp: max-arity-decr' intro: rules-transform-iter del: Rewriting-Pterm-Elim.irulesI)

Trick as described in §7.1 in the locale manual.

locale irules' = irules
sublocale irules' \subseteq irules'-as-irules: irules C-info transform-irule-set-iter rs
unfolding transform-irule-set-iter-def by (rule rules-transform-iter)

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

sublocale irules' \subseteq irules'-as-prules: prules C-info Rewriting-Pterm.compile (transform-irule-set-iter rs)
by (rule irules'-as-irules.compile-rules) (rule transform-finished)

6.1.6 Big-step semantics

context srules begin

definition global-css :: (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

  Qed
apply (rule list.pred-mono-strong)
apply (rule all-rules)
apply (auto elim: clausesE simp: Stem.closed-except-simps)
done

moreover have list-all (\(\lambda(-, t). \sim \text{is-Vconstr} t\)) as-vrules
  unfolding as-vrules-def
  by (auto simp: list-all-iff)

ultimately show list-all vrule as-vrules
  unfolding list-all-iff by fastforce
next
  show distinct (map fst as-vrules)
    using distinct by auto
next
  show fdisjnt (fst | | fset-of-list rs) C
    using disjnt by simp
next
  show list-all (\(\lambda(-, rhs). \text{not-shadows-vconsts} rhs\)) as-vrules
    unfolding not-shadows-vconsts-def
    apply (rule list-all-as-vrulesI)
    apply auto
    apply (rule list.pred-mono-strong)
    apply (rule not-shadows)
    by (auto simp: list-all-iff list-ex-iff all-consts-def elim!: clausesE)
next
  show vconstructor-value-rs as-vrules
    unfolding vconstructor-value-rs-def
    apply (rule conjI)
    unfolding vconstructor-value-def
    apply (rule list-all-as-vrulesI)
    apply simp add: list-all-iff
    apply simp
    apply simp
    using disjnt by simp
next
  show list-all (\(\lambda(-, rhs). \text{vwelldefined} rhs\)) as-vrules
    unfolding vwelldefined-def
    apply (rule list-all-as-vrulesI)
    apply auto
    apply (rule list.pred-mono-strong)
    apply (rule vwelldefined-rs)
    apply auto
    apply (erule clausesE)
    apply hypsubst-thin
    apply (subst (asm) welldefined-sabs)
    by simp
next
  show distinct all-constructors

280
by \textbf{(fact distinct-ctr)}

\textbf{qed}

\textbf{sublocale} srules-as-vrules: vrules C-info as-vrules 
\textbf{by} (fact srules-as-vrules)

\textbf{lemma} rs’-rs-eq: srules-as-vrules.rs’ = rs
\textbf{unfolding} srules-as-vrules.rs’-def 
\textbf{unfolding} as-vrules-def
\textbf{apply} (subst map-prod-def)
\textbf{apply} simp
\textbf{unfolding} comp-def
\textbf{apply} (subst case-prod-twice)
\textbf{apply} (rule list-map-snd-id)
\textbf{unfolding} global-css-def
\textbf{using} all-rules map
\textbf{apply} (auto simp: list-all-iff map-of-is-map map-prod-def fmap-of-list.rep-eq)
\textbf{subgoal for} a b
\textbf{by} (erule ballE[\textbf{where} x = (a, b)], cases b, auto simp: is-abs-def term-cases-def)
\textbf{done}

\textbf{lemma} veval-correct:
\textbf{fixes} v
\textbf{assumes} as-vrules, fmempty \vdash v t \downarrow v wellformed t closed t
\textbf{shows} rs, fmempty \vdash_s t \downarrow value-to-sterm v
\textbf{using} assms
\textbf{by} (rule srules-as-vrules.veval-correct[unfolded rs’-rs-eq])

\textbf{end}

\textbf{6.1.7 ML-style semantics}

\textbf{context} srules \textbf{begin}

\textbf{lemma} as-vrules-mk-rec-env: fmap-of-list as-vrules = mk-rec-env global-css fmempty
\textbf{apply} (subst global-css-def)
\textbf{apply} (subst as-vrules-def)
\textbf{apply} (subst mk-rec-env-def)
\textbf{apply} (rule fmap-ext)
\textbf{apply} (subst fmlookup-fmmap-keys)
\textbf{apply} (subst fmap-of-list.rep-eq)
\textbf{apply} (subst fmap-of-list.rep-eq)
\textbf{apply} (subst map-of-map-keyed)
\textbf{apply} (subst (2) map-prod-def)
\textbf{apply} (subst id-apply)
\textbf{apply} (subst map-of-map)
\textbf{apply} simp
\textbf{apply} (subst option.map-comp)
\textbf{apply} (rule option.map-cong)

281
apply (rule refl)
apply simp
apply (subst global-css-def)
apply (rule refl)
done

abbreviation (input) vrelated ≡ srules-as-vrules.vrelated
notation srules-as-vrules.vrelated (v v/ - ≈ [0, 50])

lemma vrecabs-global-css-refl:
  assumes name |∈| fmdom global-css
  shows ⊢ v Vrecabs global-css name fmempty ≈ Vrecabs global-css name fmempty
using assms
proof (coinduction arbitrary: name)
  case vrelated
    have rel-option (λv₁ v₂. (∃ name. v₁ = Vrecabs global-css name fmempty ∧ v₂ = Vrecabs global-css name fmempty ∧ name |∈| fmdom global-css) ∨ ⊢ v v₁ ≈ v₂)
    (fmlookup (fmap-of-list as-vrules) y) (fmlookup (mk-rec-env global-css fmempty) y)
    for y
      apply (subst as-vrules-mk-rec-env)
      apply (rule option.rel-refl-strong)
      apply (rule disjI1)
      apply (simp add: mk-rec-env-def)
      apply (elim conjE exE)
      apply (intro exI conjI)
      by (auto intro: fmdomI)
    with vrelated show ?case
    by fastforce
qed

lemma as-vrules-refl-rs: fmrel-on-fset (fst |'| fset-of-list as-vrules) vrelated (fmap-of-list as-vrules) (fmap-of-list as-vrules)
apply rule
apply (subst (2) as-vrules-def)
apply (subst (2) as-vrules-def)
apply (simp add: fmap-of-list.rep-eq)
apply (rule rel-option-reflI)
apply simp
apply (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-fimage-eqI)

lemma as-vrules-refl-C: fmrel-on-fset C vrelated (fmap-of-list as-vrules) (fmap-of-list as-vrules)
proof
  fix c
  assume c |∈| C

282
hence \( c \in fset-of-list (\map \text{fst} \text{as-vrules}) \)
using \text{srules-as-vrules.vconstructor-value-rs}
unfolding \text{vconstructor-value-rs-def fdisjnt-alt-def}
by auto
hence \( c \in fmdom (\map \text{fmap-of-list} \text{as-vrules}) \)
by simp
hence \( \text{fmlookup (\map \text{fmap-of-list} \text{as-vrules}) \ c = \text{None} } \)
by (metis \text{fmdom-notD})
thus rel-option vrelated \( (\text{fmlookup (\map \text{fmap-of-list} \text{as-vrules}) \ c)} \)
(\text{fmlookup (\map \text{fmap-of-list} \text{as-vrules}) \ c)}
by simp
qed

lemma \text{veval'}-correct'':
fixes \( t \, v \)
assumes \( \text{fmap-of-list as-vrules} \vdash v \downarrow v \)
assumes wellformed \( t \)
assumes \( \neg \text{shadows-consts} \ t \)
assumes welldefined \( t \)
assumes closed \( t \)
assumes \( \text{vno-abs} \ v \)
shows as-vrules, \( \text{fempty} \vdash v \downarrow v \)
proof –
obtain \( v_1 \) where \( \text{as-vrules, fempty} \vdash v \downarrow v_1 \vdash v_1 \approx v \)
using \( \text{fmap-of-list as-vrules} \vdash v_1 \downarrow v \)
proof (rule \text{srules-as-vrules.veval'}-correct', \text{unfold as-vrules-fst})
show wellformed \( t \neg \text{shadows-consts} \ t \) closed \( t \) const |\subseteq| all-consts
by \text{fact+}
next
show wellformed-venv (\text{fmap-of-list as-vrules})
apply rule
using \text{srules-as-vrules.all-rules}
apply (auto simp: list-all-iff)
done
next
show not-shadows-vconsts-env (\text{fmap-of-list as-vrules})
apply rule
using \text{srules-as-vrules.not-shadows}
apply (auto simp: list-all-iff)
done
next
have \text{fmrel-on-fset} (\text{fst} \setminus \text{fset-of-list as-vrules} \cup C) \ vrelated (\text{fmap-of-list as-vrules}) (\text{fmap-of-list as-vrules})
apply (rule \text{fmrel-on-fset-unionI})
apply (rule as-vrules-refl-rs)
apply (rule as-vrules-refl-C)
done
show \text{fmrel-on-fset} (\text{consts} \ t) \ vrelated (\text{fmap-of-list as-vrules}) (\text{fmap-of-list as-vrules})
apply (rule fnrel-on-fsubset)
apply fact+
using assms by (auto simp: allconsts-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 allconsts rs) env
nsEmpty, sem-env.c = nsEmpty |)

lemma compile-sem-env:
evaluate-dec ck mn env state (compile-group allconsts 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 allconsts 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

284
lemma compile-prog\[unfolded combine-dec-result.\text{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 | }))
using compile-sem-env' compile-group-def by simp

definition sem-env \::\ v \text{sem-env} where
sem-env \equiv\ extend-dec-env (as-sem-env empty-sem-env)
lemma cupcake-sem-env:
is-cupcake-all-env sem-env
unfolding as-sem-env-def sem-env-def
apply (rule is-cupcake-all-envI)
apply (simp add: extend-dec-env-def empty-sem-env-def nsEmpty-def)
apply (rule cupcake-nsAppend-preserve)
apply (simp add: empty-sem-env-def)
apply (simp add: nsEmpty-def)
apply (rule mk-letrec-cupcake)
apply simp
apply (simp add: empty-sem-env-def)
done

lemma sem-env-refl: fmrel related-v (fmmap-of-list as-vrules) (fmmap-of-ns (sem-env.v sem-env))
proof
fix name
show rel-option related-v (fmlookup (fmmap-of-list as-vrules) name) (fmlookup (fmmap-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
apply (simp add: fmmap-of-list[symmetric, unfolded apsnd-def map-prod-def id-def]
fmap.rel-map
global-css-def Let-def map-produce map-produce 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) \in\ 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 \( \langle \text{name}, \text{rhs} \rangle \in \text{set rs} \) all-rules

by (cases rhs) (auto simp: list-all-iff term-cases-def)

show \(?\text{thesis}\)

unfolding related-fun-alt-def \( \langle \text{clauses rhs} = \text{cs} \rangle \)

proof (intro conjI)

show list-all2 (rel-prod related-pat related-exp) cs (map (\( \lambda \) \( \text{pat} \), \( \text{t} \)). (mk-ml-pat (mk-pat pat), mk-con (frees pat |∪| all-consts) \( \text{t} \))) 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 \( \text{pat t} \)

proof (rule mk-exp-correctness)

have \( \neg \) shadows-consts rhs

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

by (auto simp: list-all-iff all-consts-def)

thus \( \neg \) shadows-consts \( \text{t} \)

unfolding \( \text{rhs} = \text{Sabs cs} \) using prems

by (auto simp: list-all-iff list-ex-iff)

next

have frees \( \text{t} \) |⊆| frees \( \text{pat} \)

using \( \langle \text{closed rhs} \rangle \) prems unfolding \( \langle \text{rhs} = \cdot \rangle \)

apply (auto simp: list-all-iff Sterm.closed-except-simps)

apply (erule ballE[where \( \text{x} = (\text{pat}, \text{t})\)])

apply (auto simp: closed-except-def)

done

moreover have \( \text{consts t} \) |⊆| all-consts

using (welldefined rhs) prems unfolding \( \langle \text{rhs} = \cdot \rangle \) welldefined-sabs

by (auto simp: list-all-iff all-consts-def)

ultimately show ids \( \text{t} \) |⊆| frees \( \text{pat} \) |∪| all-consts

unfolding ids-def by auto

qed (auto simp: all-consts-def)

done

next

have 1: frees (Sabs cs) = \( \{||\} \)

using \( \langle \text{closed rhs} \rangle \) unfolding \( \langle \text{rhs} = \text{Sabs cs} \rangle \)

by (auto simp: closed-except-def)

have 2: welldefined rhs

using swelldefined-rs \( \langle \text{name}, \text{rhs} \rangle \in \text{set rs} \)

by (auto simp: list-all-iff)

show fresh-fNext all-consts |\( \xi |\) 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 \notin all-consts
by (rule fNext-not-member)
qed
qed
done

lemma semantic-correctness':
assumes cupcake-evaluate-single sem-env (mk-con all-consts t) (Rval ml-v)
assumes welldefined t closed t \not\ shadows-consts t wellformed t
obtains v where fmap-of-list as-vrules \vdash t \downarrow 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 \not\ 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 (\\lambda \cdot list-all (\lambda (pat, t). const s t \mid \subseteq C \mid \cup \ fmdom global-css))
global-css
  apply (subst (2) global-css-def)
  apply (rule fmpred-of-list)
  apply (auto simp: map-prod-def)
subgoal premises prem for pat t
  proof –
    from prem obtain cs where t = Sabs cs
    by (elim clausesE)
    have welldefined t

287
using welldefined-rs prems
by (auto simp: list-all-iff fmdom-global-css)

show thesis
using (welldefined t);
unfolding (t = \cdot) welldefined-sabs
by (auto simp: all-consts-def list-all-iff fmdom-global-css)
qed
done

show fnpred (∑ v. welldefined') (fmap-of-list as-vrules)
apply (rule fnpred-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 fnimage-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 fnpred-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 fnrel-on-fset (ids t) related-v (fmap-of-list as-vrules) (fmap-of-ns (sem-env.v sem-env))
  unfolding fnrel-on-fset-fnrel-restrict
  apply (rule fnrel-restrict-fset)
  apply (rule sem-env-refl)
done
next
show consts t \subseteq fmdom (fmap-of-list as-vrules) \cup C
apply (subst fmdom-fmap-of-list)
apply (subst as-vrules-fst')
apply simp
using assms by (auto simp: all-consts-def)
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)
  using assms proof (induction v)
  case (Conv c vs)
    then obtain name tid where c = Some ((as-string name), TId (Short tid))
      apply (elim is-cupcake-value.elims)
    subgoal
      by (metis name.sel v.simps(2))
    by auto
  show ?case
    unfolding (c = -)
    apply simp
    apply (rule related-v.conv)
    apply (simp add: list.rel-map)
    apply (rule list.rel-refl-strong)
    apply (rule Conv)
    using Conv unfolding (c = -)
    by (auto simp: list-all-iff)

qed auto

lemma related-v-abs-free-uniq:
  assumes related-v v1 ml-v related-v v2 ml-v cake-no-abs ml-v
  shows v1 = v2
  using assms proof (induction arbitrary: v)
  case (Conv vs1 ml-vs name)
    then obtain vs2 where v2 = Vconstr name vs2 list-all2 related-v vs2 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 (=) vs1 vs2
    using (list-all2 - vs1 -) (list-all2 - vs2 -) (list-all cake-no-abs ml-vs)
    by (induction arbitrary: vs2 rule: list.rel-induct) (auto simp: list-all2-Cons2)
  thus ?case

qed auto
\[
\v_2 = -
\]

by \((\text{simp add: list.rel-eq})\)

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 \(v = \text{cake-to-value} ml-v\)
using assms by \((\text{metis cake-to-value-related related-v-abs-free-uniq})\)

end

context srules begin

lemma cupcake-sem-env-preserve:
assumes cupcake-evaluate-single sem-env \((\text{mk-con} S t) (\text{Rval} ml-v) \text{ wellformed} t\)
shows is-cupcake-value ml-v
apply \((\text{rule} \ \text{cupcake-single-preserve}[OF assms(1)])\)
apply \((\text{rule} \ \text{cupcake-sem-env})\)
apply \((\text{rule} \ \text{mk-exp-cupcake})\)
apply fact
done

lemma semantic-correctness'' :
assumes cupcake-evaluate-single sem-env \((\text{mk-con} \ \text{all-consts} t) (\text{Rval} ml-v)\)
assumes welldefined t closed t \(\neg \text{shadows-consts} t \ \text{wellformed} t\)
assumes cake-no-abs ml-v
shows \(\text{fmap-of-list} \ \text{as-vrules} \vdash v t \downarrow \text{cake-to-value} ml-v\)
using assms
by \((\text{metis cupcake-sem-env-preserve semantic-correctness' related-v-abs-free-cake-to-value})\)

end

6.1.9 Composition

context rules begin

abbreviation term-to-nterm where
\(\text{term-to-nterm} \ t \equiv \text{fresh-frun} (\text{Term-to-Nterm.\ term-to-nterm} \ [] \ t) \ \text{all-consts}\)

abbreviation stem-to-cake where
\(\text{stem-to-cake} \equiv \text{rules-as-nrules.crules-as-irules'.irules'-as-prules.prules-as-srules.mk-con all-consts}\)

abbreviation term-to-cake t \equiv stem-to-cake \((\text{pterm-to-sterm} \ (\text{nterm-to-nterm} \ t)))\)
abbreviation cake-to-term t \equiv (\text{convert-term} \ (\text{value-to-sterm} \ (\text{cake-to-value} \ t))) :: \text{term}\

abbreviation cake-sem-env \equiv \text{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 (term-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 ⊢ₐ 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
have ?heads = heads-of rs
    using fmdom-compiled unfolding compiled-def by simp

have wellformed (nterm-to-pterms (term-to-nterm t))
    by auto
hence wellformed (pterms-to-sterm (nterm-to-pterms (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 m-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 m-v)

using unclocked-single-fun-eq by fastforce

have cupcake-evaluate-single cake-sem-env (stern-to-cake (pterm-to-stern (nterm-to-ptermt
(term-to-nterm t))) (Rval m-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-stern-wellformed)

hence no-abs (cake-to-value ml-v)

using (cake-no-abs -)


hence no-abs (value-to-stern (cake-to-value ml-v))

by (metis vno-abs-value-to-stern)

hence no-abs (stern-to-ptermt (value-to-stern (cake-to-value ml-v)))

by (metis stern-to-ptermt 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-stern (nterm-to-ptermt (term-to-nterm t)))

apply (subst pterm-to-ptermt-consts)

apply fact

apply (subst consts-nterm-to-ptermt)

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 \( \neg \text{shadows-consts (pterm-to-sterm (nterm-to-pterms (term-to-nterm t)))} \)
apply (subst pterm-to-sterm-shadows [symmetric])
apply fact
apply (subst shadows-nterm-to-pterms)
unfolding shadows-consts-def
apply simp
apply (rule term-to-nterm-all-vars [where \( T = \text{fempty, simplified, THEN} \) fdisjnt-swap])
apply (fold wellformed-term-def)
apply fact
using (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-pterms (term-to-nterm t))
using closed-nterm-to-pterms unfolding closed-except-def
by auto
have closed (pterm-to-sterms (nterm-to-pterms (term-to-nterm t)))
unfolding closed-except-def
apply (subst pterm-to-sterm-frees)
apply fact
using closed-nterm-to-pterms unfolding closed-except-def
by auto
have fmap-of-list compiled \( \vdash v \) pterm-to-sterm (nterm-to-pterms (term-to-nterm t)) \( \downarrow \) cake-to-value ml-v
by (rule cake-semantic-correctness)
fact+
hence have closed (term-to-nterm t)
using assms unfolding closed-except-def
by auto
have fmap-of-list compiled \( \vdash v \) pterm-to-sterm (nterm-to-pterms (term-to-nterm t)) \( \downarrow \) cake-to-value ml-v
by (rule cake-semantic-correctness)
fact+
proof
have \( \neg \text{rules-as-nrules, crules-as-irules', irules'-as-prules, prules-as-srules, as-irules} \)
\( \vdash v \) pterm-to-sterms (nterm-to-pterms (term-to-nterm t)) \( \downarrow \) cake-to-value ml-v
using assms unfolding compiled-def by simp
next
show \( \neg \) rules-as-nrules, crules-as-irules', irules'-as-prules, prules-as-srules, as-irules, shadows-consts
(pterms-to-sterm (nterm-to-pterms (term-to-nterm t)))
using (\( \neg \) shadows-consts (\( :-\) stern) (?heads = heads-of rs) by auto
qed fact+
hence Rewriting-Sterm.compile (Rewriting-Pterm.compile (transform-irule-set-iter
(Rewriting-Pterm-Elim.compile (consts-of compile)))
\( \vdash v \) pterm-to-sterms (nterm-to-pterms (term-to-nterm t)) \( \downarrow \) 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

293
\((\text{Rewriting-Pterm-Elim.compile (consts-of compile)})\) \(\vdash_p \text{pterm-to-sterm (nterm-to-pterms) (term-to-nterm t)}\) \(\rightarrow^{*} \text{value-to-sterm (cake-to-value ml-v)}\)

by (rule rules-as-nrules.crules-as-irules'.irules'-as-prules.prules-as-srules.seval-correct)

\text{hence Rewriting-Pterm.compile (transform-irule-set-iter (Rewriting-Pterm-Elim.compile (consts-of compile)))} \(\vdash_i \text{stern-to-pterms (pterm-to-sterm (nterm-to-pterms (term-to-nterm t)))} \rightarrow^{*} \text{stern-to-sterm (value-to-sterm (cake-to-value ml-v))}\)

by (rule rules-as-nrules.crules-as-irules'.irules'-as-prules.compile-correct-rnt)

\text{hence Rewriting-Pterm.compile (transform-irule-set-iter (Rewriting-Pterm-Elim.compile (consts-of compile)))} \(\vdash_p \text{nterm-to-pterms (term-to-nterm t)} \rightarrow^{*} \text{stern-to-pterms (value-to-sterm (cake-to-value ml-v))}\)

by (subst (asm) pterm-to-sterm-stern-to-pterms) fact

\text{hence transform-irule-set-iter (Rewriting-Pterm-Elim.compile (consts-of compile))} \(\vdash_i \text{nterm-to-pterms (term-to-nterm t)} \rightarrow^{*} \text{stern-to-sterm (value-to-sterm (cake-to-value ml-v))}\)

by (rule rules-as-nrules.crules-as-irules'.irules'-as-prules.compile-correct-rnt)

\text{hence rewrite (rules-as-nrules.crules-as-irules.transform-finished)}

\text{have Rewriting-Pterm-Elim.compile (consts-of compile)} \(\vdash_i \text{nterm-to-pterms (term-to-nterm t)} \rightarrow^{*} \text{stern-to-sterm (value-to-sterm (cake-to-value ml-v))}\)

apply (rule rules-as-nrules.crules-as-irules.transform-correct-nt-no-abs)

using (transform-irule-set-iter -i - \(\rightarrow^{*} \) unfolding transform-irule-set-iter-def

apply simp

apply fact+

done

\text{then obtain} t' where compile \(\vdash_n \text{term-to-nterm t} \rightarrow^{*} t' t' \approx_i \text{stern-to-pterms}\)

(value-to-sterm (cake-to-value ml-v))

using (closed (term-to-nterm t))

by (metis rules-as-nrules.compile-correct-rnt)

\text{hence no-abs t'}

using (no-abs (stern-to-pterms -))

by (metis inrelated-no-abs)

\text{have rs} \(\vdash \text{nterm-to-termt' (term-to-nterm t)} \rightarrow^{*} \text{nterm-to-termt'} t'\)

by (rule compile-correct-rnt) fact+

\text{hence rs} \(\vdash t \rightarrow^{*} \text{nterm-to-termt'} t'\)

apply (subst (asm) fresh-frun-def)

apply (subst (asm) term-to-nterm-nterm-to-term[where \(S = \text{empty} \) and \(t = t\), simplified])

apply (fold wellformed-term-def)

apply fact

using asms unfolding closed-exception-def by auto

\text{have nterm-to-pterms t' = stern-to-pterms (value-to-sterm (cake-to-value ml-v))}

using (t' \approx_i -)

by auto

\text{hence (convert-term t' :: pterm) = convert-term (value-to-sterm (cake-to-value ml-v))}

apply (subst (asm) nterm-to-pterms)
apply fact
apply (subst (asm) stem-to-pterms)
apply fact
apply assumption
done

hence nterm-to-pterms t' = convert-term (value-to-sterm (cake-to-value ml-v))
apply (subst nterm-to-pterms')
apply (rule no-abs t')
apply (rule convert-term-inj)

subgoal premises
apply (rule convert-term-no-abs)
apply fact
done

subgoal premises
apply (rule convert-term-no-abs)
apply fact
done
apply (subst convert-term-idem)
apply (rule no-abs t')
apply (rule convert-term-idem)
apply (rule no-abs (value-to-sterm (cake-to-value ml-v)))
apply assumption
done

thus ?thesis
using rs t —>∗ nterm-to-pterms t'
by simp
qed

end


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)
(pterms-to-sterm (nterm-to-pterms (fresh-frun (term-to-nterm [] t) (heads-of rs |\]| constructors.C C-info))))

lemma (in rules) Compiler.term-to-exp C-info rs = term-to-cake
unfolding term-to-exp-def by (simp add: all-consts-def)

primrec compress-pterms :: pterm ⇒ pterm where
compress-pterms (Pabs cs) = Pabs (fcompress (map-prod id compress-pterms |\]| cs))

6.2 Executable compilation chain
compress-ptermin (Pconst name) = Pconst name |
compress-ptermin (Pvar name) = Pvar name |
compress-ptermin (t $p u) = compress-ptermin t $p compress-ptermin u

lemma compress-ptermin-eq[simp]: compress-ptermin t = t
by (induction t) (auto simp: subst-pabs-id fset-map-snd-id map-prod-def fnember rep-eq)
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 (fcompress ◦ fimage (map-prod id compress-ptermin)))
definition compress-prule-set :: prule fset ⇒ prule fset where
compress-prule-set = fcompress ◦ fimage (map-prod id compress-ptermin)
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.mkletrec-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
\textbf{compile} $C$-info =
\begin{itemize}
    \item CakeML-Backend.compile' $C$-info $\circ$
    \item Rewriting-Sterm.compile $\circ$
    \item compress-prule-set $\circ$
    \item Rewriting-Pterm.compile $\circ$
    \item transform-irule-set-iter $\circ$
    \item compress-irule-set $\circ$
    \item Rewriting-Pterm-Elim.compile $\circ$
    \item compress-crule-set $\circ$
    \item Rewriting-Ntermconsts-of $\circ$
    \item fcompress $\circ$
    \item Rewriting-Nterm.compile' $C$-info $\circ$
    \item fcompress
\end{itemize}

\textbf{definition} compile-to-env :: $C$-info $\Rightarrow$ rule fset $\Rightarrow$ v sem-env where
\begin{itemize}
    \item compile-to-env $C$-info =
        \begin{itemize}
            \item sem-env $C$-info $\circ$
            \item Rewriting-Sterm.compile $\circ$
            \item compress-prule-set $\circ$
            \item Rewriting-Pterm.compile $\circ$
            \item transform-irule-set-iter $\circ$
            \item compress-irule-set $\circ$
            \item Rewriting-Pterm-Elim.compile $\circ$
            \item compress-crule-set $\circ$
            \item Rewriting-Nterm.consts-of $\circ$
            \item fcompress $\circ$
            \item Rewriting-Nterm.compile' $C$-info $\circ$
            \item fcompress
        \end{itemize}
\end{itemize}

\textbf{lemma} (in rules) Compiler.compile-to-env $C$-info $rs$ = rules.cake-sem-env $C$-info $rs$
\begin{itemize}
    \item unfolding Compiler.compile-to-env-def Compiler.sem-env-def Compiler.as-sem-env-def
    \item Compiler.empty-sem-env-def
\end{itemize}
\begin{itemize}
    \item unfolding rules-as-nrules.crules-as-irules'.irules'-as-prules.prules-as-srules.sem-env-def
    \item unfolding rules-as-nrules.crules-as-irules'.irules'-as-prules.prules-as-srules.as-sem-env-def
\end{itemize}
\begin{itemize}
    \item unfolding empty-sem-env-def
\end{itemize}
\begin{itemize}
    \item by (auto simp:
        \begin{itemize}
            \item Compiler.compress-irule-set-eq[abs-def]
            \item Composition.transform-irule-set-iter-def[abs-def]
            \item Compiler.transform-irule-set-iter-def[abs-def] comp-def pre-constants.all-consts-def)
        \end{itemize}
\end{itemize}

\textbf{export-code}
\begin{itemize}
    \item term-to-exp compile compile-to-env
    \item checking Scala
\end{itemize}

\textbf{end}