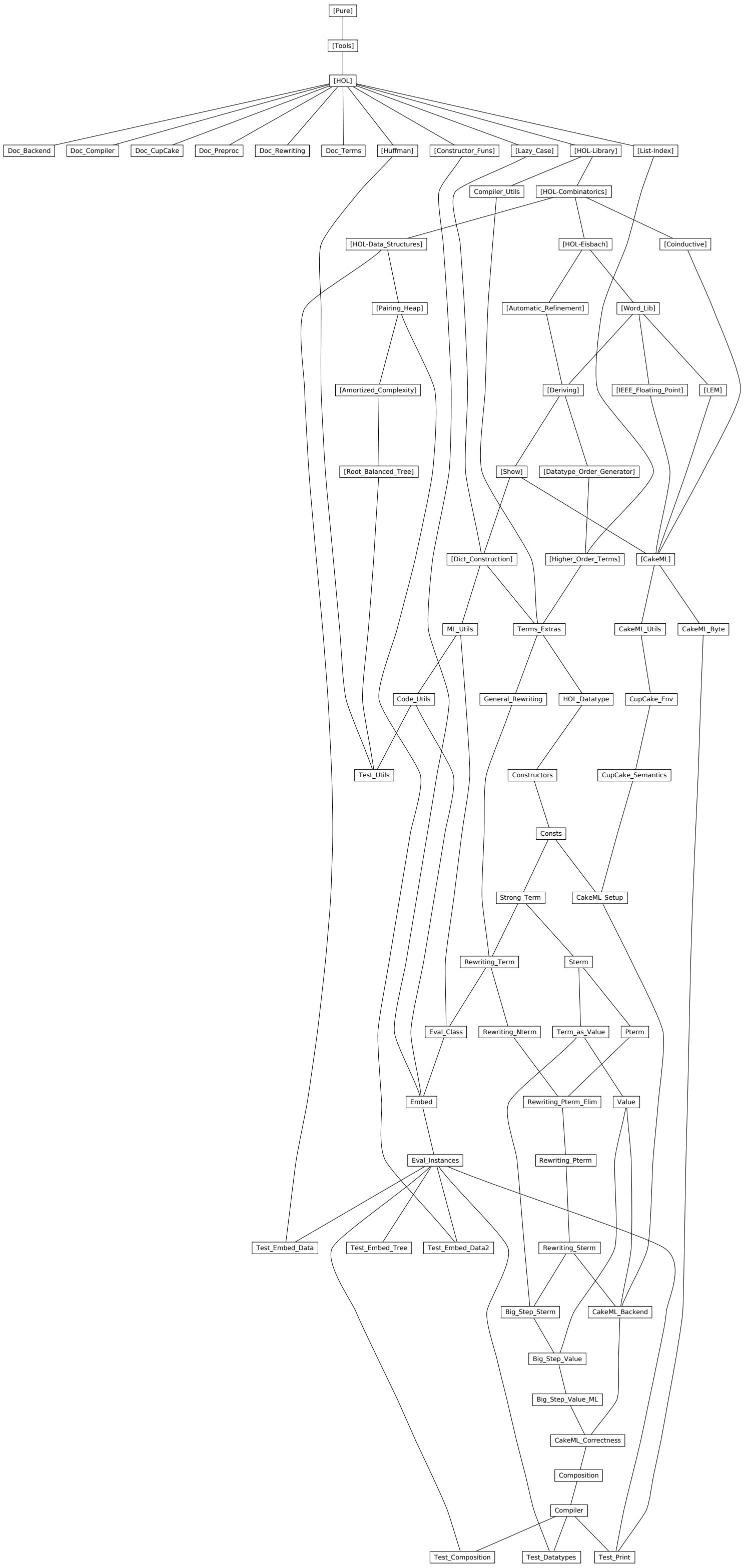
A Verified Code Generator from Isabelle/HOL to CakeML

Lars Hupel

May 4, 2022

Contents


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

Chapter 1

Terms

```
theory Doc-Terms
imports Main
begin

end
```

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

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

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

⟨ML⟩

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

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

⟨*proof*⟩

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

fun *unsplit-rule* :: (*name* × (*term list* × *'a*)) ⇒ (*term* × *'a*) **where**
unsplit-rule (*name*, (*args*, *rhs*)) = (*name* $$ *args*, *rhs*)

lemma *split-unsplit*: *split-rule* (*unsplit-rule t*) = *t*
⟨*proof*⟩

lemma *unsplit-split*:
  assumes *basic-rule r*
  shows *unsplit-rule* (*split-rule r*) = *r*
⟨*proof*⟩

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

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

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

lemma *mk-pat-simps*[*simp*]:
  mk-pat (*name* $$ *args*) = *Patconstr name* (*map mk-pat args*)
  mk-pat (*Free name*) = *Patvar name*
⟨*proof*⟩

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

lemma *mk-pat-frees*:
  assumes *linear p*
  shows *patvars* (*mk-pat p*) = *frees p*
⟨*proof*⟩

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

fun *pattern-compatible* :: *term* $\Rightarrow$ *term* $\Rightarrow$ *bool* **where**
pattern-compatible (t_1 \$ t_2) (u_1 \$ u_2) $\longleftrightarrow$ *pattern-compatible* t_1 u_1 $\wedge$ ($t_1 = u_1$ $\longrightarrow$ *pattern-compatible* t_2 u_2) |
pattern-compatible *t u* $\longleftrightarrow$ $t = u$ $\vee$ *non-overlapping t u*

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

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

lemma *pattern-compatible-refl*[*intro?*]: *pattern-compatible t t*
⟨*proof*⟩

corollary *pattern-compatile-reflP*[*intro*!]: *reflp pattern-compatible*
⟨*proof*⟩

lemma *pattern-compatible-cases*[*consumes 1*]:
  assumes *pattern-compatible t u*
  obtains (*eq*) $t = u$
    | (*non-overlapping*) *non-overlapping t u*
⟨*proof*⟩

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

lemma *rev-accum-rel-refl*[*intro*]: *reflp R* $\Longrightarrow$ *rev-accum-rel R xs xs*
⟨*proof*⟩

lemma *rev-accum-rel-length*:
  assumes *rev-accum-rel R xs ys*
  shows *length xs* = *length ys*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *rev-accum-rel-snoc-eqE*: *rev-accum-rel P* (*xs* @ [*a*]) (*xs* @ [*b*]) $\Longrightarrow$ *P a b*
⟨*proof*⟩

end

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

abbreviation *patterns-compatibles* :: (*term list* × *'a*) *fset* ⇒ *bool* **where**
patterns-compatibles ≡ *fpairwise* (λ($pats_1$, -) ($pats_2$, -). *patterns-compatible* $pats_1$ $pats_2$)

lemma *pattern-compatible-combD*:
assumes *length xs* = *length ys pattern-compatible* (*list-comb f xs*) (*list-comb f ys*)
shows *patterns-compatible xs ys*
⟨*proof*⟩

lemma *pattern-compatible-combI*[*intro*]:
assumes *patterns-compatible xs ys pattern-compatible f g*
shows *pattern-compatible* (*list-comb f xs*) (*list-comb g ys*)
⟨*proof*⟩

experiment begin

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

lemma *pattern-compatible t u* ⟷ *t* = *u* ∨ *non-overlapping t u*
⟨*proof*⟩

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

proposition *non-overlapping* (*list-comb c pats1*) (*list-comb c pats2*)
⟨*proof*⟩

proposition ¬ *patterns-compatible pats1 pats2*
⟨*proof*⟩

end

abbreviation *pattern-compatibles* :: (*term* × *'a*) *fset* ⇒ *bool* **where**
pattern-compatibles ≡ *fpairwise* (λ(lhs_1, -) (lhs_2, -). *pattern-compatible* lhs_1 lhs_2)

corollary *match-compatible-pat-eq*:
assumes *pattern-compatible* t_1 t_2 *linear* t_1 *linear* t_2
assumes *match* t_1 *u* = *Some* env_1 *match* t_2 *u* = *Some* env_2
shows $t_1 = t_2$
⟨*proof*⟩

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 $env_1 = env_2$
⟨*proof*⟩

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*⟩

lemma *compatible-find-match*:
    assumes *pattern-compatibles* (*fset-of-list cs*) *list-all* (*linear* ∘ *fst*) *cs is-fmap* (*fset-of-list cs*)
   assumes *match pat t* = *Some env* (*pat*, *rhs*) ∈ *set cs*
   shows *find-match cs t* = *Some* (*env*, *pat*, *rhs*)
⟨*proof*⟩

context *term* **begin**

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

abbreviation *arity-compatibles* :: (*'a* × *'b*) *fset* ⇒ *bool* **where**
arity-compatibles ≡ *fpairwise* (λ(lhs_1, -) (lhs_2, -). *arity-compatible* lhs_1 lhs_2)

definition *head* :: *'a* ⇒ *name* **where**
head t ≡ *const-name* (*fst* (*strip-comb t*))

abbreviation *heads-of* :: (*term* × *'a*) *fset* ⇒ *name fset* **where**
heads-of rs ≡ (*head* ∘ *fst*) |`| *rs*

end

definition *arity* :: (*'a list* × *'b*) *fset* ⇒ *nat* **where**
arity rs = *fthe-elem'* ((*length* ∘ *fst*) |`| *rs*)

lemma *arityI*:
  assumes *fBall rs* (λ(*pats*, -). *length pats* = *n*) *rs* ≠ {||}
  shows *arity rs* = *n*
⟨*proof*⟩

end

1.2 Reflecting HOL datatype definitions

theory *HOL-Datatype*
imports
  Terms-Extras
  HOL−Library.Datatype-Records
  HOL−Library.Finite-Map
  Higher-Order-Terms.Name
begin

datatype *typ* =
  TVar name |
  TApp name typ list

datatype-compat *typ*

context begin

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

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

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

end

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

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

⟨*ML*⟩

end

1.3 Constructor information

theory *Constructors*
imports
  Terms-Extras

HOL-Datatype
begin

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

locale *constructors* =
fixes *C-info* :: *C-info*
begin

definition *flat-C-info* :: (*string* × *nat* × *string*) *list* **where**
flat-C-info = *do* {
(*tname*, *Cs*) ← *sorted-list-of-fmap C-info*;
(*C*, *params*) ← *sorted-list-of-fmap* (*constructors Cs*);
[(*as-string C*, (*length params*, *as-string tname*))]
}

definition *all-tdefs* :: *name fset* **where**
all-tdefs = *fmdom C-info*

definition *C* :: *name fset* **where**
C = *ffUnion* (*fmdom* |‘| *constructors* |‘| *fmran C-info*)

definition *all-constructors* :: *name list* **where**
all-constructors =
concat (*map* (λ(-, *Cs*). *map fst* (*sorted-list-of-fmap* (*constructors Cs*))) (*sorted-list-of-fmap C-info*))

end

declare *constructors.C-def*[*code*]
declare *constructors.flat-C-info-def*[*code*]
declare *constructors.all-constructors-def*[*code*]

export-code
constructors.C constructors.flat-C-info constructors.all-constructors
checking *Scala*

end

1.4 Special constants

theory *Consts*
imports
Constructors
Higher-Order-Terms.Nterm
begin

locale *special-constants* = *constructors*

locale *pre-constants* = *special-constants* +
fixes *heads* :: *name fset*
begin

definition *all-consts* :: *name fset* **where**
all-consts = *heads* |∪| *C*

abbreviation *welldefined* :: *'a*::*term* ⇒ *bool* **where**
welldefined t ≡ *consts t* |⊆| *all-consts*

sublocale *welldefined*: *simple-syntactic-and welldefined*
⟨*proof*⟩

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*)
⟨*proof*⟩

end

end

1.5 Term algebra extended with wellformedness

theory *Strong-Term*
imports *Consts*
begin

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

assumes *all-frees-app*[*simp*]: *all-frees* (*app* u_1 u_2) = *all-frees* u_1 |∪| *all-frees* u_2
begin

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

end

context *pre-constants* **begin**

definition *shadows-consts* :: ′*a*::*pre-strong-term* ⇒ *bool* **where**
shadows-consts t ⟷ ¬ *fdisjnt all-consts* (*all-frees t*)

sublocale *shadows*: *simple-syntactic-or shadows-consts*
⟨*proof*⟩

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

end

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

class *strong-term* = *pre-strong-term* +
assumes *raw-frees-all-frees*: *abs-pred* (λ*t*. *frees t* |⊆| *all-frees t*) *t*
assumes *raw-subst-wellformed*: *abs-pred* (λ*t*. *wellformed t* ⟶ (∀ *env*. *wellformed-env env* ⟶ *wellformed* (*subst t env*))) *t*
begin

lemma *frees-all-frees*: *frees t* |⊆| *all-frees t*
⟨*proof*⟩

lemma *subst-wellformed*: *wellformed t* ⟹ *wellformed-env env* ⟹ *wellformed* (*subst t env*)
⟨*proof*⟩

end

global-interpretation *wellformed*: *subst-syntactic-and wellformed* :: ′*a*::*strong-term* ⇒ *bool*
⟨*proof*⟩

instantiation *term* :: *strong-term* **begin**

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

all-frees-term - = {||}

lemma *frees-all-frees-term*[*simp*]: *all-frees t = frees* (*t*::*term*)
⟨*proof*⟩

definition *wellformed-term* :: *term* ⇒ *bool* **where**
[*simp*]: *wellformed-term t* ⟷ *Term.wellformed t*

instance ⟨*proof*⟩

end

instantiation *nterm* :: *strong-term* **begin**

definition *wellformed-nterm* :: *nterm* ⇒ *bool* **where**
[*simp*]: *wellformed-nterm t* ⟷ *True*

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

instance ⟨*proof*⟩

end

lemma (**in** *pre-constants*) *shadows-consts-frees*:
  fixes *t* :: *'a*::*strong-term*
  shows ¬ *shadows-consts t* ⟹ *fdisjnt all-consts* (*frees t*)
⟨*proof*⟩

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

end

1.6 Terms with sequential pattern matching

theory *Sterm*
imports *Strong-Term*
begin

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

datatype-compat *sterm*

derive *linorder sterm*

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

type-synonym *sclauses* = (*term* × *sterm*) *list*

lemma *sterm-induct*[*case-names Sconst Svar Sabs Sapp*]:
  assumes ⋀*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* $\$_s$ *u*)
  shows *P t*
⟨*proof*⟩

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

definition *app-sterm* **where**
app-sterm t u = *t* $\$_s$ *u*

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

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

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

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

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

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

fun *subst-sterm* **where**

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

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

instance
⟨*proof*⟩

end

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

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

lemma *abs-pred-stermI*[*intro*]:
  assumes ⋀*cs*. (⋀*pat t*. (*pat*, *t*) ∈ *set cs* ⟹ *P t*) ⟹ *P* (*Sabs cs*)
  shows *abs-pred P t*
⟨*proof*⟩

instance ⟨*proof*⟩
    including *fset.lifting fmap.lifting*
    ⟨*proof*⟩

end

lemma *no-abs-abs*[*simp*]: ¬ *no-abs* (*Sabs cs*)
⟨*proof*⟩

lemma *closed-except-simps*:
  closed-except (*Svar x*) *S* ⟷ *x* |∈| *S*
  closed-except (t_1 $\$_s$ t_2) *S* ⟷ *closed-except* t_1 *S* ∧ *closed-except* t_2 *S*
  closed-except (*Sabs cs*) *S* ⟷ *list-all* (λ(*pat*, *t*). *closed-except t* (*S* |∪| *frees pat*)) *cs*
  closed-except (*Sconst name*) *S* ⟷ *True*
⟨*proof*⟩

lemma *closed-except-sabs*:
  assumes *closed* (*Sabs cs*) (*pat*, *rhs*) ∈ *set cs*

shows *closed-except rhs* (*frees pat*)
⟨*proof*⟩

instantiation *sterm* :: *strong-term* **begin**

fun *wellformed-sterm* :: *sterm* ⇒ *bool* **where**
wellformed-sterm (t_1 $\$_s$ t_2) ⟷ *wellformed-sterm* t_1 ∧ *wellformed-sterm* t_2 |
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 (t_1 $\$_s$ t_2) = *all-frees-sterm* t_1 |∪| *all-frees-sterm* t_2 |
all-frees-sterm (*Sabs cs*) = *ffUnion* (*fset-of-list* (*map* (λ(*P*, *T*). *P* |∪| *T*) (*map* (*map-prod frees all-frees-sterm*) *cs*))) |
all-frees-sterm (*Sconst* -) = {||}

instance ⟨*proof*⟩

end

lemma *match-sabs*[*simp*]: ¬ *is-free t* ⟹ *match t* (*Sabs cs*) = *None*
⟨*proof*⟩

context *pre-constants* **begin**

lemma *welldefined-sabs*: *welldefined* (*Sabs cs*) ⟷ *list-all* (λ(-, *t*). *welldefined t*) *cs*
⟨*proof*⟩

lemma *shadows-consts-sterm-simps*[*simp*]:
shadows-consts (t_1 $\$_s$ t_2) ⟷ *shadows-consts* t_1 ∨ *shadows-consts* t_2
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*⟩

lemma *subst-shadows*:
assumes ¬ *shadows-consts* (*t*::*sterm*) *not-shadows-consts-env* Γ
shows ¬ *shadows-consts* (*subst t* Γ)
⟨*proof*⟩

end

end

1.7 Terms with explicit pattern matching

theory *Pterm*
imports
  ../Utils/Compiler-Utils
  Consts
  Sterm — Inclusion of this theory might seem a bit strange. Indeed, it is only for technical reasons: to allow for a **quickcheck** setup.
begin

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

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

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

lemma *sterm-to-pterm-total*:
  obtains t' **where** t = *sterm-to-pterm* t'
⟨*proof*⟩

lemma *pterm-induct*[*case-names Pconst Pvar Pabs Papp*]:
  assumes $\bigwedge x.\ P\ (Pconst\ x)$
  assumes $\bigwedge x.\ P\ (Pvar\ x)$
  assumes $\bigwedge cs.\ (\bigwedge pat\ t.\ (pat, t) \mathrel{|{\in}|} cs \Longrightarrow P\ t) \Longrightarrow P\ (Pabs\ cs)$
  assumes $\bigwedge t\ u.\ P\ t \Longrightarrow P\ u \Longrightarrow P\ (t\ \$_p\ u)$
  shows $P\ t$
⟨*proof*⟩

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

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

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

definition *const-pterm* **where**

const-pterm = *Pconst*

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

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

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

function (*sequential*) *subst-pterm* **where**
subst-pterm (*Pvar s*) *env* = (*case fmlookup env s of Some t* $\Rightarrow$ *t* | *None* $\Rightarrow$ *Pvar s*) |
subst-pterm (t_1 $\$_p$ t_2) *env* = *subst-pterm* t_1 *env* $\$_p$ *subst-pterm* t_2 *env* |
subst-pterm (*Pabs cs*) *env* = *Pabs* ((λ(*pat*, *rhs*). (*pat*, *subst-pterm rhs* (*fmdrop-fset* (*frees pat*) *env*))) |‘| *cs*) |
subst-pterm t - = *t*
⟨*proof*⟩

termination
⟨*proof*⟩
  including *fset.lifting* ⟨*proof*⟩

primrec *consts-pterm* :: *pterm* $\Rightarrow$ *name fset* **where**
consts-pterm (*Pconst x*) = {|*x*|} |
consts-pterm (t_1 $\$_p$ t_2) = *consts-pterm* t_1 |∪| *consts-pterm* t_2 |
consts-pterm (*Pabs cs*) = *ffUnion* (*snd* |‘| *map-prod id consts-pterm* |‘| *cs*) |
consts-pterm (*Pvar* -) = {||}

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

instance
⟨*proof*⟩

end

corollary *subst-pabs-id*:
  assumes $\bigwedge$*pat rhs*. (*pat*, *rhs*) |∈| *cs* $\Longrightarrow$ *subst rhs* (*fmdrop-fset* (*frees pat*) *env*) = *rhs*
  shows *subst* (*Pabs cs*) *env* = *Pabs cs*
⟨*proof*⟩

corollary *frees-pabs-alt-def*:
$\textit{frees}\ (\textit{Pabs cs}) = \textit{ffUnion}\ ((\lambda(\textit{pat}, \textit{rhs}).\ \textit{frees rhs} - \textit{frees pat})\ |`|\ \textit{cs})$
$\langle \textit{proof} \rangle$

lemma *sterm-to-pterm-frees*[*simp*]: $\textit{frees}\ (\textit{sterm-to-pterm}\ t) = \textit{frees}\ t$
$\langle \textit{proof} \rangle$

lemma *sterm-to-pterm-consts*[*simp*]: $\textit{consts}\ (\textit{sterm-to-pterm}\ t) = \textit{consts}\ t$
$\langle \textit{proof} \rangle$

lemma *subst-sterm-to-pterm*:
$\textit{subst}\ (\textit{sterm-to-pterm}\ t)\ (\textit{fmmap sterm-to-pterm env}) = \textit{sterm-to-pterm}\ (\textit{subst}\ t\ \textit{env})$
$\langle \textit{proof} \rangle$

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

definition *abs-pred-pterm* :: $(\textit{pterm} \Rightarrow \textit{bool}) \Rightarrow \textit{pterm} \Rightarrow \textit{bool}$ **where**
[*code del*]: $\textit{abs-pred}\ P\ t \longleftrightarrow (\forall\, \textit{cs}.\ t = \textit{Pabs cs} \longrightarrow (\forall\, \textit{pat}\ t.\ (\textit{pat}, t)\ |{\in}|\ \textit{cs} \longrightarrow P\ t) \longrightarrow P\ t)$

context begin

private lemma *abs-pred-trivI0*: $P\ t \Longrightarrow \textit{abs-pred}\ P\ (t{::}\textit{pterm})$
$\langle \textit{proof} \rangle$

instance $\langle \textit{proof} \rangle$

end

end

lemma *no-abs-abs*[*simp*]: $\neg\ \textit{no-abs}\ (\textit{Pabs cs})$
$\langle \textit{proof} \rangle$

lemma *sterm-to-pterm*:
assumes *no-abs t*
shows $\textit{sterm-to-pterm}\ t = \textit{convert-term}\ t$
$\langle \textit{proof} \rangle$

abbreviation *Pabs-single* (Λ_p -. - [*0*, *50*] *50*) **where**
$\textit{Pabs-single}\ x\ \textit{rhs} \equiv \textit{Pabs}\ \{|\ (\textit{Free}\ x, \textit{rhs})\ |\}$

lemma *closed-except-simps*:
$\textit{closed-except}\ (\textit{Pvar}\ x)\ S \longleftrightarrow x\ |{\in}|\ S$
$\textit{closed-except}\ (t_1\ \$_p\ t_2)\ S \longleftrightarrow \textit{closed-except}\ t_1\ S \wedge \textit{closed-except}\ t_2\ S$
$\textit{closed-except}\ (\textit{Pabs cs})\ S \longleftrightarrow \textit{fBall cs}\ (\lambda(\textit{pat}, t).\ \textit{closed-except}\ t\ (S\ |{\cup}|\ \textit{frees pat}))$
$\textit{closed-except}\ (\textit{Pconst name})\ S \longleftrightarrow \textit{True}$

⟨*proof*⟩

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

function (*sequential*) *wellformed-pterm* :: *pterm* ⇒ *bool* **where**
wellformed-pterm (t_1 $\$_p$ t_2) ⟷ *wellformed-pterm* t_1 ∧ *wellformed-pterm* t_2 |
wellformed-pterm (*Pabs cs*) ⟷ *fBall cs* (λ(*pat*, *t*). *linear pat* ∧ *wellformed-pterm t*) ∧ *is-fmap cs* ∧ *pattern-compatibles cs* ∧ *cs* ≠ {||} |
wellformed-pterm - ⟷ *True*
⟨*proof*⟩

termination
⟨*proof*⟩
  including *fset.lifting* ⟨*proof*⟩

primrec *all-frees-pterm* :: *pterm* ⇒ *name fset* **where**
all-frees-pterm (*Pvar x*) = {|*x*|} |
all-frees-pterm (t_1 $\$_p$ t_2) = *all-frees-pterm* t_1 |∪| *all-frees-pterm* t_2 |
all-frees-pterm (*Pabs cs*) = *ffUnion* ((λ(*P*, *T*). *P* |∪| *T*) |'| *map-prod frees all-frees-pterm* |'| *cs*) |
all-frees-pterm (*Pconst* -) = {||}

instance
⟨*proof*⟩

end

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

instance *pterm* :: *strong-term* ⟨*proof*⟩

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*)
⟨*proof*⟩

corollary *subst-closed-pabs*:
  assumes (*pat*, *rhs*) |∈| *cs closed* (*Pabs cs*)
  shows *subst rhs* (*fmdrop-fset* (*frees pat*) *env*) = *rhs*
⟨*proof*⟩

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

shadows-consts (*Pconst name*) ⟷ *False*
⟨*proof*⟩

end

1.8 Irreducible terms (values)

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

1.9 Viewing *sterm* as values

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

context *constructors* **begin**

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

lemma *value-distinct*:
Sabs cs ≠ *name* $$ *ts* (**is** *?P*)
name $$ *ts* ≠ *Sabs cs* (**is** *?Q*)
⟨*proof*⟩

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

lemma *svar-value*[*simp*]: ¬ *is-value* (*Svar name*)
⟨*proof*⟩

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*⟩

end

fun *smatch′* :: *pat* ⇒ *sterm* ⇒ (*name*, *sterm*) *fmap option* **where**
smatch′ (*Patvar name*) *t* = *Some* (*fmap-of-list* [(*name*, *t*)]) |
smatch′ (*Patconstr name ps*) *t* =
(*case strip-comb t of*
(*Sconst name′*, *vs*) ⇒
(*if name* = *name′* ∧ *length ps* = *length vs then*
map-option (*foldl* ($++_f$) *fmempty*) (*those* (*map2 smatch′ ps vs*))
else

None)
| - ⇒ *None*)

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

context *constructors* **begin**

context begin

private lemma *smatch-list-comb-is-value*:
assumes *is-value t*
shows *match* (*name* $$ *ps*) *t* = (*case strip-comb t of*
(*Sconst name′*, *vs*) ⇒
(*if name* = *name′* ∧ *length ps* = *length vs then*
map-option (*foldl* $(++_f)$ *fmempty*) (*those* (*map2 match ps vs*))
else
None)
| - ⇒ *None*)
⟨*proof*⟩

lemma *smatch-smatch′-eq*:
assumes *linear pat is-value t*
shows *match pat t* = *smatch′* (*mk-pat pat*) *t*
⟨*proof*⟩

end

end

end

1.10 A dedicated value type

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

datatype *value* =
is-Vconstr: *Vconstr name value list* |
Vabs sclauses (*name*, *value*) *fmap* |
Vrecabs (*name*, *sclauses*) *fmap name* (*name*, *value*) *fmap*

type-synonym *vrule* = *name* × *value*

⟨*ML*⟩

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))

⟨*ML*⟩

quickcheck-generator *value*
  constructors: *quickcheck.Value*

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

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

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

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

declare *pred.simps*[*simp del*]

lemma *pred-alt-def*[*simp*, *code*]:
  pred (*Vconstr name vs*) ⟷ *Q name* ∧ *list-all pred vs*
  pred (*Vabs cs* Γ) ⟷ *fmpred* (λ-. *pred*) Γ ∧ *P* Γ *cs*
  pred (*Vrecabs css name* Γ) ⟷ *fmpred* (λ-. *pred*) Γ ∧ *pred-fmap* (*P* Γ) *css* ∧ *name* |∈| *fmdom css* ∧ *R* (*fmdom css*)
⟨*proof*⟩

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

lemma *vmatch-env*:
  assumes *vmatch pat v* = *Some env pred v*
  shows *fmpred* (λ-. *pred*) *env*
⟨*proof*⟩

end

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

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

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

corollary *value-to-sterm-env*:
  assumes *fmpred* (λ-. *pred*) Γ
  shows *fmpred* (λ-. *S*) (*fmmap value-to-sterm* Γ)
⟨*proof*⟩

end

locale *value-sterm-pred* = *value-pred* + *S*: *simple-syntactic-and S* **for** *S* +
  assumes *const*: ⋀*name*. *Q name* ⟹ *S* (*const name*)
  assumes *abs*: ⋀Γ *cs*.
    (⋀*n v*. *fmlookup* Γ *n* = *Some v* ⟹ *pred v* ⟹ *S* (*value-to-sterm v*)) ⟹
    fmpred (λ-. *pred*) Γ ⟹
    P Γ *cs* ⟹
    S (*Sabs* (*map* (λ(*pat*, *t*). (*pat*, *subst t* (*fmmap value-to-sterm* (*fmdrop-fset* (*frees pat*) Γ)))) *cs*))
begin

sublocale *pre-value-sterm-pred*
⟨*proof*⟩

end

```
global-interpretation vwellformed:
  value-sterm-pred
    λ-. wellformed-clauses
    λ-. True
    λ-. True
    wellformed
  defines vwellformed = vwellformed.pred
⟨proof⟩

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

global-interpretation vclosed:
  value-sterm-pred
    λΓ cs. list-all (λ(pat, t). closed-except t (fmdom Γ |∪| frees pat)) cs
    λ-. True
    λ-. True
    closed
  defines vclosed = vclosed.pred
⟨proof⟩

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

context pre-constants begin

sublocale vwelldefined:
  value-sterm-pred
    λ- cs. list-all (λ(-, t). welldefined t) cs
    λname. name |∈| C
    λdom. dom |⊆| heads
    welldefined
  defines vwelldefined = vwelldefined.pred
⟨proof⟩

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*⟩

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

abbreviation *vconstructor-value-env* ≡ *fmpred* (λ-. *vconstructor-value*)

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

end

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

context *pre-constants* **begin**

sublocale *not-shadows-vconsts*:
value-sterm-pred
λ- *cs*. *list-all* (λ(*pat*, *t*). *fdisjnt all-consts* (*frees pat*) ∧ ¬ *shadows-consts t*) *cs*
λ-. *True*
λ-. *True*
λ*t*. ¬ *shadows-consts t*
defines *not-shadows-vconsts* = *not-shadows-vconsts.pred*
⟨*proof*⟩

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

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

end

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

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

lemma (**in** *constructors*) *term-to-value-to-sterm*:
assumes *is-value t*
shows *value-to-sterm* (*term-to-value t*) = *t*
⟨*proof*⟩

lemma *vmatch-dom*:
assumes *vmatch pat v* = *Some env*
shows *fmdom env* = *patvars pat*

⟨*proof*⟩

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

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

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

lemma *veq-structure-simps*[*code*, *simp*]:
  veq-structure (*Vabs* cs_1 Γ_1) (*Vabs* cs_2 Γ_2)
  veq-structure (*Vrecabs* css_1 $name_1$ Γ_1) (*Vrecabs* css_2 $name_2$ Γ_2)
  veq-structure (*Vconstr* $name_1$ *ts*) (*Vconstr* $name_2$ *us*) ⟷ $name_1 = name_2$ ∧ *list-all2 veq-structure ts us*
⟨*proof*⟩

lemma *veq-structure-refl*[*simp*]: *veq-structure t t*
⟨*proof*⟩

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

lemma *veq-structure-eq-left*:
  assumes *veq-structure t u vno-abs t*
  shows *t* = *u*
⟨*proof*⟩

lemma *veq-structure-eq-right*:
  assumes *veq-structure t u vno-abs u*
  shows *t* = *u*
⟨*proof*⟩

fun *vmatch'* :: *pat* ⇒ *value* ⇒ (*name*, *value*) *fmap option* **where**
vmatch' (*Patvar name*) *v* = *Some* (*fmap-of-list* [(*name*, *v*)]) |
vmatch' (*Patconstr name ps*) *v* =
  (*case v of*
    Vconstr name' vs ⇒

(*if name* = *name′* ∧ *length ps* = *length vs then*
map-option (*foldl* ($++_f$) *fmempty*) (*those* (*map2 vmatch′ ps vs*))
else
None)
| - ⇒ *None*)

lemma *vmatch-vmatch′-eq*: *vmatch p v* = *vmatch′ p v*
⟨*proof*⟩

locale *value-struct-rel* =
fixes *Q* :: *value* ⇒ *value* ⇒ *bool*
assumes *Q-impl-struct*: *Q* t_1 t_2 ⟹ *veq-structure* t_1 t_2
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*
⟨*proof*⟩

lemma *eq-right*: *Q t u* ⟹ *vno-abs u* ⟹ *t* = *u*
⟨*proof*⟩

context begin

private lemma *vmatch′-rel*:
assumes *Q* t_1 t_2
shows *rel-option* (*fmrel Q*) (*vmatch′ p* t_1) (*vmatch′ p* t_2)
⟨*proof*⟩

lemma *vmatch-rel*: *Q* t_1 t_2 ⟹ *rel-option* (*fmrel Q*) (*vmatch p* t_1) (*vmatch p* t_2)
⟨*proof*⟩

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)
⟨*proof*⟩

lemmas *vfind-match-rel′* =
vfind-match-rel[
where *R* = (=) **and** cs_1 = *cs* **and** cs_2 = *cs* **for** *cs*,
unfolded prod.rel-eq,
OF list.rel-refl, *OF refl*]

end end

hide-fact *vmatch-vmatch′-eq*
hide-const *vmatch′*

global-interpretation *veq-structure*: *value-struct-rel veq-structure*
⟨*proof*⟩

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

lemma *env-eq-eq*:
  assumes *env-eq venv senv*
  shows *senv* = *fmmap value-to-sterm venv*
⟨*proof*⟩

context *constructors* **begin**

context begin

private lemma *vmatch-eq0*: *rel-option env-eq* (*vmatch p v*) (*smatch' p* (*value-to-sterm v*))
⟨*proof*⟩

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*))
⟨*proof*⟩

end

end

abbreviation *match-related* **where**
match-related ≡ (λ(Γ_1, *pat*$_1$, *rhs*$_1$) (Γ_2, *pat*$_2$, *rhs*$_2$). *rhs*$_1$ = *rhs*$_2$ ∧ *pat*$_1$ = *pat*$_2$ ∧ *env-eq* Γ_1 Γ_2)

lemma (**in** *constructors*) *find-match-eq*:
  assumes *list-all* (*linear* ∘ *fst*) *cs vconstructor-value v*
  shows *rel-option match-related* (*vfind-match cs v*) (*find-match cs* (*value-to-sterm v*))
⟨*proof*⟩

inductive *erelated* :: *value* ⇒ *value* ⇒ *bool* (- $\approx_e$ -) **where**
constr: *list-all2 erelated ts us* ⟹ *Vconstr name ts* $\approx_e$ *Vconstr name us* |
abs: *fmrel-on-fset* (*ids* (*Sabs cs*)) *erelated* Γ_1 Γ_2 ⟹ *Vabs cs* Γ_1 $\approx_e$ *Vabs cs* Γ_2 |
rec-abs:
  pred-fmap (λ*cs*. *fmrel-on-fset* (*ids* (*Sabs cs*)) *erelated* Γ_1 Γ_2) *css* ⟹
    Vrecabs css name Γ_1 $\approx_e$ *Vrecabs css name* Γ_2

code-pred *erelated* ⟨*proof*⟩

global-interpretation *erelated*: *value-struct-rel erelated*

⟨*proof*⟩

lemma *erelated-refl*[*intro*]: $t \approx_e t$
⟨*proof*⟩

export-code
value-to-sterm vmatch vwellformed vclosed erelated-i-i pre-constants.vwelldefined constructors.vconstructor-value-rs pre-constants.not-shadows-vconsts term-to-value vfind-match veq-structure vno-abs
checking *Scala*

end

Chapter 2

A smaller version of CakeML: *CupCakeML*

theory *Doc-CupCake*
imports *Main*
begin

end

2.1 CupCake environments

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

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

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

fun *is-cupcake-exp* :: *exp* ⇒ *bool* **where**
is-cupcake-exp (*Ast.Var* (*Short* -)) ⟷ *True* |
is-cupcake-exp (*Ast.App oper es*) ⟷ *oper* = *Ast.Opapp* ∧ *list-all is-cupcake-exp es* |
is-cupcake-exp (*Ast.Con* (*Some* (*Short* -)) *es*) ⟷ *list-all is-cupcake-exp es* |
is-cupcake-exp (*Ast.Fun* - *e*) ⟷ *is-cupcake-exp e* |
is-cupcake-exp (*Ast.Mat e cs*) ⟷ *is-cupcake-exp e* ∧ *list-all* (λ(*p*, *e*). *is-cupcake-pat p* ∧ *is-cupcake-exp e*) *cs* ∧ *cake-linear-clauses cs* |
is-cupcake-exp - ⟷ *False*

abbreviation *cupcake-clauses* :: (*Ast.pat* × *exp*) *list* ⇒ *bool* **where**
cupcake-clauses ≡ *list-all* (λ(*p*, *e*). *is-cupcake-pat p* ∧ *is-cupcake-exp e*)

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

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

definition *empty-sem-env* :: *v sem-env* **where**
empty-sem-env = ⦇ *sem-env.v* = *nsEmpty*, *sem-env.c* = *static-cenv* ⦈

lemma *v-of-empty-sem-env*[*simp*]: *sem-env.v empty-sem-env* = *nsEmpty*
⟨*proof*⟩

lemma *c-of-empty-sem-env*[*simp*]: *c empty-sem-env* = *static-cenv*
⟨*proof*⟩

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

lemma *is-cupcake-all-envE*:
  assumes *is-cupcake-all-env env*
  obtains *v c* **where** *env* = ⦇ *sem-env.v* = *Bind v* [], *sem-env.c* = *c* ⦈ *c* =
static-cenv list-all (*is-cupcake-value* ∘ *snd*) *v*
⟨*proof*⟩

fun *is-cupcake-ns* :: *v-ns* ⇒ *bool* **where**
is-cupcake-ns (*Bind v0* []) ⟷ *list-all* (*is-cupcake-value* ∘ *snd*) *v0* |
is-cupcake-ns - ⟷ *False*

lemma *is-cupcake-nsE*:
  assumes *is-cupcake-ns ns*
  obtains *v* **where** *ns* = *Bind v* [] *list-all* (*is-cupcake-value* ∘ *snd*) *v*
⟨*proof*⟩

lemma *is-cupcake-all-envD*:
assumes *is-cupcake-all-env env*
shows *is-cupcake-ns* (*sem-env.v env*) *cupcake-c-ns* (*c env*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

fun *cupcake-pmatch* :: ((*string*),(*string*),(*nat*∗*tid-or-exn*))*namespace* ⇒ *pat* ⇒ *v* ⇒(*string*∗*v*)*list* ⇒((*string*∗*v*)*list*)*match-result* **where**

```
cupcake-pmatch cenv (Pvar x) v0 env = Match ((x, v0)# env) |
cupcake-pmatch cenv (Pcon (Some (Short n)) ps) (Conv (Some (n′, t′)) vs) env =
  (case cupcake-nsLookup cenv n of
      Some (l, t)=>
        if same-tid t t′ ∧ (List.length ps = l) then
          if same-ctor (n, t) (n′,t′) then
            Matching.fold2 (λp v m. case m of
                Match env ⇒ cupcake-pmatch cenv p v env
            | m ⇒ m) Match-type-error ps vs (Match env)
          else
            No-match
        else
          Match-type-error
    | - => Match-type-error) |
cupcake-pmatch cenv - - - = Match-type-error
```

fun *cupcake-match-result* :: - ⇒ *v* ⇒(*pat*∗*exp*)*list* ⇒ *v* ⇒ (*exp* × *pat* × (*char list*

$\times$ *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' $\Rightarrow$ *Rval* (*e*, *p*, *env'*) |
No-match $\Rightarrow$ *cupcake-match-result cenv v0 pes err-v* |
Match-type-error $\Rightarrow$ *Rerr* (*Rabort Rtype-error*))
else
Rerr (*Rabort Rtype-error*))

lemma *cupcake-match-resultE*:
assumes *cupcake-match-result cenv v0 pes err-v* = *Rval* (*e*, *p*, *env'*)
obtains *init rest*
where *pes* = *init* @ (*p*, *e*) # *rest*
and *distinct* (*pat-bindings p* [])
and *list-all* (λ(*p*, *e*). *cupcake-pmatch cenv p v0* [] = *No-match* $\wedge$ *distinct* (*pat-bindings p* [])) *init*
and *cupcake-pmatch cenv p v0* [] = *Match env'*
$\langle$*proof*$\rangle$

lemma *cupcake-pmatch-eq*:
is-cupcake-pat pat $\Longrightarrow$ *pmatch-single envC s pat v0 env* = *cupcake-pmatch envC pat v0 env*
$\langle$*proof*$\rangle$

lemma *cupcake-match-result-eq*:
cupcake-clauses pes $\Longrightarrow$
match-result env s v pes err-v =
map-result (λ(*e*, -, *env'*). (*e*, *env'*)) *id* (*cupcake-match-result* (*c env*) *v pes err-v*)
$\langle$*proof*$\rangle$

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

lemma *cupcake-nsBind-preserve*:
is-cupcake-ns ns $\Longrightarrow$ *is-cupcake-value v0* $\Longrightarrow$ *is-cupcake-ns* (*nsBind k v0 ns*)
$\langle$*proof*$\rangle$

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*)
$\langle$*proof*$\rangle$

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*)
$\langle$*proof*$\rangle$

lemma *cupcake-nsAppend-preserve*: *is-cupcake-ns ns1* ⟹ *is-cupcake-ns ns2* ⟹ *is-cupcake-ns* (*nsAppend ns1 ns2*)
⟨*proof*⟩

lemma *cupcake-alist-to-ns-preserve*: *list-all* (*is-cupcake-value* ∘ *snd*) *env* ⟹ *is-cupcake-ns* (*alist-to-ns env*)
⟨*proof*⟩

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*⟩

context begin

lemma *cup-pmatch-list-length-neq*:
  length vs ≠ *length ps* ⟹ *Matching.fold2*(*λp v m. case m of*
    Match env ⇒ *cupcake-pmatch cenv p v env*
    | *m* ⇒ *m*) *Match-type-error ps vs m* = *Match-type-error*
  ⟨*proof*⟩

lemma *cup-pmatch-list-nomatch*:
  length vs = *length ps* ⟹ *Matching.fold2*(*λp v m. case m of*
    Match env ⇒ *cupcake-pmatch cenv p v env*
    | *m* ⇒ *m*) *Match-type-error ps vs No-match* = *No-match*
  ⟨*proof*⟩

lemma *cup-pmatch-list-typerr*:
  length vs = *length ps* ⟹ *Matching.fold2*(*λp v m. case m of*
    Match env ⇒ *cupcake-pmatch cenv p v env*
    | *m* ⇒ *m*) *Match-type-error ps vs Match-type-error* = *Match-type-error*
  ⟨*proof*⟩ **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*
        Match env ⇒ *cupcake-pmatch cenv p v env*
      | *m* ⇒ *m*)
      Match-type-error ps vs (*Match env*))
  ⟨*proof*⟩ **lemma** *cupcake-pmatch-preserve0*:
  is-cupcake-pat pat ⟹
   is-cupcake-value v0 ⟹
    list-all (*is-cupcake-value* ∘ *snd*) *env* ⟹
     cupcake-c-ns envC ⟹
      if-match (*list-all* (*is-cupcake-value* ∘ *snd*)) (*cupcake-pmatch envC pat v0 env*)
⟨*proof*⟩

lemma *cupcake-pmatch-preserve*:
is-cupcake-pat pat ⟹
is-cupcake-value v0 ⟹
list-all (*is-cupcake-value* ∘ *snd*) *env* ⟹
cupcake-c-ns envC ⟹
cupcake-pmatch envC pat v0 env = *Match env′* ⟹
list-all (*is-cupcake-value* ∘ *snd*) *env′*
⟨*proof*⟩

end

lemma *cupcake-match-result-preserve*:
cupcake-c-ns envC ⟹
cupcake-clauses pes ⟹
is-cupcake-value v ⟹
if-rval (λ(*e*, *p*, *env′*). *is-cupcake-pat p* ∧ *is-cupcake-exp e* ∧ *list-all* (*is-cupcake-value* ∘ *snd*) *env′*)
(*cupcake-match-result envC v pes err-v*)
⟨*proof*⟩

lemma *static-cenv-lookup*:
assumes *cupcake-nsLookup static-cenv i* = *Some* (*len*, *b*)
obtains *name* **where** *b* = *TypeId* (*Short name*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *cupcake-nsLookup-preserve*:
assumes *is-cupcake-ns ns nsLookup ns n* = *Some v0*
shows *is-cupcake-value v0*
⟨*proof*⟩

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*⟩

end

fun *list-all2-shortcircuit* **where**
list-all2-shortcircuit P (*x* # *xs*) (*y* # *ys*) ⟷ (*case y of Rval* - ⇒ *P x y* ∧ *list-all2-shortcircuit P xs ys* | *Rerr* - ⇒ *P x y*) |

list-all2-shortcircuit P [] [] ⟷ True |
list-all2-shortcircuit P - - ⟷ False

lemma *list-all2-shortcircuit-induct*[*consumes 1*, *case-names nil cons-val cons-err*]:
  assumes *list-all2-shortcircuit P xs ys*
  assumes *R* [] []
  assumes ⋀*x xs y ys. P x* (*Rval y*) ⟹ *list-all2-shortcircuit P xs ys* ⟹ *R xs ys* ⟹ *R* (*x* # *xs*) (*Rval y* # *ys*)
  assumes ⋀*x xs y ys. P x* (*Rerr y*) ⟹ *R* (*x* # *xs*) (*Rerr y* # *ys*)
  shows *R xs ys*
⟨*proof*⟩

lemma *list-all2-shortcircuit-mono*[*mono*]:
  assumes *R* ≤ *Q*
  shows *list-all2-shortcircuit R* ≤ *list-all2-shortcircuit Q*
⟨*proof*⟩

lemma *list-all2-shortcircuit-weaken*: *list-all2-shortcircuit P xs ys* ⟹ (⋀*xs ys. P xs ys* ⟹ *Q xs ys*) ⟹ *list-all2-shortcircuit Q xs ys*
⟨*proof*⟩

lemma *list-all2-shortcircuit-rval*[*simp*]:
  list-all2-shortcircuit P xs (*map Rval ys*) ⟷ *list-all2* (λ*x y. P x* (*Rval y*)) *xs ys*
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

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

app1:
list-all2-shortcircuit (*cupcake-evaluate-single env*) (*rev es*) *rs* ⟹
sequence-result rs = *Rval vs* ⟹
do-opapp (*rev vs*) = *Some* (*env*′, *e*) ⟹
cupcake-evaluate-single env′ *e bv* ⟹
cupcake-evaluate-single env (*App Opapp es*) *bv* |
app3:
list-all2-shortcircuit (*cupcake-evaluate-single env*) (*rev es*) *rs* ⟹
sequence-result rs = *Rval vs* ⟹
do-opapp (*rev vs*) = *None* ⟹
cupcake-evaluate-single env (*App Opapp es*) (*Rerr* (*Rabort Rtype-error*)) |
app6:
list-all2-shortcircuit (*cupcake-evaluate-single env*) (*rev es*) *rs* ⟹
sequence-result rs = *Rerr err* ⟹
cupcake-evaluate-single env (*App op0 es*) (*Rerr err*) |
mat1:
cupcake-evaluate-single env e (*Rval v0*) ⟹
cupcake-match-result (*c env*) *v0 pes Bindv* = *Rval* (*e*′, -, *env*′) ⟹
cupcake-evaluate-single (*env* (| *sem-env.v* := *nsAppend* (*alist-to-ns env*′) (*sem-env.v env*) |)) *e*′ *bv* ⟹
cupcake-evaluate-single env (*Mat e pes*) *bv* |
mat1error:
cupcake-evaluate-single env e (*Rval v0*) ⟹
cupcake-match-result (*c env*) *v0 pes Bindv* = *Rerr err* ⟹
cupcake-evaluate-single env (*Mat e pes*) (*Rerr err*) |
mat2:
cupcake-evaluate-single env e (*Rerr err*) ⟹
cupcake-evaluate-single env (*Mat e pes*) (*Rerr err*)

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

context begin

private lemma *cupcake-list-preserve0*:
list-all2-shortcircuit
(λ*e r*. *cupcake-evaluate-single env e r* ∧ (*is-cupcake-all-env env* ⟶ *is-cupcake-exp e* ⟶ *if-rval is-cupcake-value r*)) *es rs* ⟹
is-cupcake-all-env env ⟹ *list-all is-cupcake-exp es* ⟹ *sequence-result rs* = *Rval vs* ⟹ *list-all is-cupcake-value vs*
⟨*proof*⟩ **lemma** *cupcake-single-preserve0*:
cupcake-evaluate-single env e res ⟹ *is-cupcake-all-env env* ⟹ *is-cupcake-exp e* ⟹ *if-rval is-cupcake-value res*
⟨*proof*⟩

lemma *cupcake-single-preserve*:
cupcake-evaluate-single env e (*Rval res*) ⟹ *is-cupcake-all-env env* ⟹ *is-cupcake-exp e* ⟹ *is-cupcake-value res*
⟨*proof*⟩

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*
⟨*proof*⟩ **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*)
⟨*proof*⟩ **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*)
⟨*proof*⟩ **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*)
⟨*proof*⟩

lemma *cupcake-single-correct*:
assumes *cupcake-evaluate-single env e res is-cupcake-all-env env is-cupcake-exp e*
shows ∃ *s*′. *Big-Step-Unclocked-Single.evaluate env s e* (*s*′,*res*)
⟨*proof*⟩

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 ∃ *s*′. *evaluate-list* (*evaluate env*) (*s*::′*a state*) *es* (*s*′,*sequence-result rs*)
⟨*proof*⟩ **lemma** *cupcake-list-complete0*:
evaluate-list
(*λs e r*. *evaluate env s e r* ∧ (*is-cupcake-all-env env* ⟶ *is-cupcake-exp e* ⟶ *cupcake-evaluate-single env e* (*snd r*))) *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*)
⟨*proof*⟩ **lemma** *cupcake-single-complete0*:

evaluate env s e res $\Longrightarrow$ *is-cupcake-all-env env* $\Longrightarrow$ *is-cupcake-exp e* $\Longrightarrow$ *cupcake-evaluate-single env e* (*snd res*)
⟨*proof*⟩

lemma *cupcake-single-complete*:
evaluate env s e (*s'*, *res*) $\Longrightarrow$ *is-cupcake-all-env env* $\Longrightarrow$ *is-cupcake-exp e* $\Longrightarrow$ *cupcake-evaluate-single env e res*
⟨*proof*⟩

lemma *cupcake-list-complete*:
evaluate-list (*evaluate env*) *s1 es res* $\Longrightarrow$
is-cupcake-all-env env $\Longrightarrow$ *list-all is-cupcake-exp es* $\Longrightarrow$ $\exists$ *rs*. *list-all2-shortcircuit* (*cupcake-evaluate-single env*) *es rs* $\wedge$ *sequence-result rs* = (*snd res*)
⟨*proof*⟩ **lemma** *cupcake-list-state-preserve0*:
assumes *evaluate-list* (λ*s e res*. *Big-Step-Unclocked-Single.evaluate env s e res* $\wedge$ (*is-cupcake-all-env env* $\longrightarrow$ *is-cupcake-exp e* $\longrightarrow$ *s* = *fst res*)) *s es res*
list-all is-cupcake-exp es is-cupcake-all-env env
shows *s* = (*fst res*)
⟨*proof*⟩

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*)
⟨*proof*⟩

corollary *cupcake-single-correct-strong*:
assumes *cupcake-evaluate-single env e res is-cupcake-all-env env is-cupcake-exp e*
shows *Big-Step-Unclocked-Single.evaluate env s e* (*s*,*res*)
⟨*proof*⟩

corollary *cupcake-single-complete-weak*:
evaluate env s e (*s*, *res*) $\Longrightarrow$ *is-cupcake-all-env env* $\Longrightarrow$ *is-cupcake-exp e* $\Longrightarrow$ *cupcake-evaluate-single env e res*
⟨*proof*⟩

end end

hide-const (**open**) *c*

end

Chapter 3

Term rewriting

theory *Doc-Rewriting*
imports *Main*
begin

end
theory *General-Rewriting*
imports *Terms-Extras*
begin

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

lemma *rt-fun*:
  $R^{**}\ t\ t' \Longrightarrow R^{**}\ (app\ t\ u)\ (app\ t'\ u)$
⟨*proof*⟩

lemma *rt-arg*:
  $R^{**}\ u\ u' \Longrightarrow R^{**}\ (app\ t\ u)\ (app\ t\ u')$
⟨*proof*⟩

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

lemma *rt-list-comb*:
  assumes *list-all2* R^{**} *ts us* $R^{**}\ t\ u$
  shows R^{**} (*list-comb t ts*) (*list-comb u us*)
⟨*proof*⟩

end

end

3.1 Higher-order term rewriting using de-Bruijn indices

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

3.1.1 Matching and rewriting

type-synonym *rule* = *term* × *term*

inductive *rewrite* :: *rule fset* ⇒ *term* ⇒ *term* ⇒ *bool* (-/ ⊢/ - ⟶/ - [*50*,*0*,*50*] *50*) **for** *rs* **where**
step: *r* |∈| *rs* ⟹ *r* ⊢ *t* → *u* ⟹ *rs* ⊢ *t* ⟶ *u* |
beta: *rs* ⊢ (Λ *t* $ *t*′) ⟶ $t\ [t']_\beta$ |
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*
⟨*proof*⟩

abbreviation *rewrite-rt* :: *rule fset* ⇒ *term* ⇒ *term* ⇒ *bool* (-/ ⊢/ - ⟶∗/ - [*50*,*0*,*50*] *50*) **where**
rewrite-rt rs ≡ $(\textit{rewrite rs})^{**}$

lemma *rewrite-beta-alt*: $t\ [t']_\beta$ = *u* ⟹ *wellformed t*′ ⟹ *rs* ⊢ (Λ *t* $ *t*′) ⟶ *u*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *split-rule-fst*: *fst* (*split-rule r*) = *head* (*fst r*)
⟨*proof*⟩

locale *rules* = *constants C-info heads-of rs* **for** *C-info* **and** *rs* :: *rule fset* +
  assumes *all-rules*: *fBall rs rule*
  assumes *arity*: *arity-compatibles rs*

assumes *fmap*: *is-fmap rs*
assumes *patterns*: *pattern-compatibles rs*
assumes *nonempty*: $rs \neq \{||\}$
assumes *not-shadows*: *fBall rs* (λ(*lhs*, -). ¬ *shadows-consts lhs*)
assumes *welldefined-rs*: *fBall rs* (λ(-, *rhs*). *welldefined rhs*)
begin

lemma *rewrite-wellformed*:
assumes $rs \vdash t \longrightarrow t'$ *wellformed t*
shows *wellformed* t'
⟨*proof*⟩

lemma *rewrite-rt-wellformed*: $rs \vdash t \longrightarrow\!* \ t' \Longrightarrow$ *wellformed t* $\Longrightarrow$ *wellformed* t'
⟨*proof*⟩

lemma *rewrite-closed*: $rs \vdash t \longrightarrow t' \Longrightarrow$ *closed t* $\Longrightarrow$ *closed* t'
⟨*proof*⟩

lemma *rewrite-rt-closed*: $rs \vdash t \longrightarrow\!* \ t' \Longrightarrow$ *closed t* $\Longrightarrow$ *closed* t'
⟨*proof*⟩

end

end

3.2 Higher-order term rewriting using explicit bound variable names

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

3.2.1 Definitions

type-synonym *nrule* = *term* × *nterm*

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

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

locale *nrules* = *constants C-info heads-of rs* **for** *C-info* **and** *rs* :: *nrule fset* +
assumes *all-rules*: *fBall rs nrule*
assumes *arity*: *arity-compatibles rs*

assumes *fmap*: *is-fmap rs*
assumes *patterns*: *pattern-compatibles rs*
assumes *nonempty*: $rs \neq \{||\}$
assumes *not-shadows*: *fBall rs not-shadowing*
assumes *welldefined-rs*: *fBall rs* (λ(-, *rhs*). *welldefined rhs*)

3.2.2 Matching and rewriting

inductive *nrewrite* :: *nrule fset* ⇒ *nterm* ⇒ *nterm* ⇒ *bool* (-/ $\vdash_n$/ - ⟶/ - [*50*,*0*,*50*] *50*) **for** *rs* **where**
step: $r\ |{\in}|\ rs \Longrightarrow r \vdash t \rightarrow u \Longrightarrow rs \vdash_n t \longrightarrow u$ |
beta: $rs \vdash_n ((\Lambda_n\ x.\ t)\ \$_n\ t') \longrightarrow$ *subst t* (*fmap-of-list* $[(x,\ t')]$) |
fun: $rs \vdash_n t \longrightarrow t' \Longrightarrow rs \vdash_n t\ \$_n\ u \longrightarrow t'\ \$_n\ u$ |
arg: $rs \vdash_n u \longrightarrow u' \Longrightarrow rs \vdash_n t\ \$_n\ u \longrightarrow t\ \$_n\ u'$

global-interpretation *nrewrite*: *rewriting nrewrite rs* **for** *rs*
⟨*proof*⟩

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

lemma (**in** *nrules*) *nrewrite-closed*:
assumes $rs \vdash_n t \longrightarrow t'$ *closed t*
shows *closed* t'
⟨*proof*⟩

corollary (**in** *nrules*) *nrewrite-rt-closed*:
assumes $rs \vdash_n t \longrightarrow{*}\ t'$ *closed t*
shows *closed* t'
⟨*proof*⟩

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 $|{\cup}|$ *frees t* $|{\cup}|$ *T* ∧ *fBall T* (λ*y*. $y > x$)
⟨*proof*⟩

lemma *term-to-nterm-all-vars*:
assumes *wellformed t fdisjnt* (*frees t*) *S*
shows *fdisjnt* (*all-frees* (*fresh-frun* (*term-to-nterm* [] *t*) (*T* $|{\cup}|$ *S*))) *S*
⟨*proof*⟩

end

fun *translate-rule* :: *name fset* ⇒ *rule* ⇒ *nrule* **where**

translate-rule S (*lhs*, *rhs*) = (*lhs*, *fresh-frun* (*term-to-nterm* [] *rhs*) (*frees lhs* |∪| *S*))

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

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

context *rules* **begin**

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

lemma *compile′-compile-eq*[*simp*]: *compile′ C-info rs* = *compile*
⟨*proof*⟩

lemma *compile-heads*: *heads-of compile* = *heads-of rs*
⟨*proof*⟩

lemma *compile-rules*: *nrules C-info compile*
⟨*proof*⟩

sublocale *rules-as-nrules*: *nrules C-info compile*
⟨*proof*⟩

end

3.2.4 Correctness of translation

theorem (**in** *rules*) *compile-correct*:
  assumes *compile* $\vdash_n$ *u* ⟶ *u′ closed u*
  shows *rs* ⊢ *nterm-to-term′ u* ⟶ *nterm-to-term′ u′*
⟨*proof*⟩

3.2.5 Completeness of translation

context *rules* **begin**

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

private lemma *compile-complete0*:
  assumes *rs* ⊢ *t* ⟶ *t′ closed t wellformed t*
  obtains *u′* **where** *compile* $\vdash_n$ *fst* (*run-state* (*term-to-nterm* [] *t*) *s*) ⟶ *u′ u′* $\approx_\alpha$ *fst* (*run-state* (*term-to-nterm* [] *t′*) *s′*)

⟨*proof*⟩

lemma *compile-complete*:
assumes *rs* $\vdash$ *t* $\longrightarrow$ *t′ closed t wellformed t*
obtains *u′* **where** *compile* $\vdash_n$ *term-to-nterm′ t* $\longrightarrow$ *u′ u′* $\approx_\alpha$ *term-to-nterm′ t′*
⟨*proof*⟩

end

end

3.2.6 Splitting into constants

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

abbreviation *arity-compatibles* :: (*term list* × *′a*) *fset* ⇒ *bool* **where**
arity-compatibles ≡ *fpairwise* (λ($pats_1$, -) ($pats_2$, -). *length* $pats_1$ = *length* $pats_2$)

lemma *arity-compatible-length*:
assumes *arity-compatibles rs* (*pats*, *rhs*) |∈| *rs*
shows *length pats* = *arity rs*
⟨*proof*⟩

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

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

lemma (**in** *pre-crules*) *crulesI*:
assumes ⋀*name crs*. (*name*, *crs*) |∈| *rs* ⟹ *arity-compatibles crs*
assumes ⋀*name crs*. (*name*, *crs*) |∈| *rs* ⟹ *is-fmap crs*
assumes ⋀*name crs*. (*name*, *crs*) |∈| *rs* ⟹ *patterns-compatibles crs*
assumes ⋀*name crs*. (*name*, *crs*) |∈| *rs* ⟹ *crs* ≠ {||}
assumes ⋀*name crs pats rhs*. (*name*, *crs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *crs* ⟹

linears pats
  assumes ⋀*name crs pats rhs.* (*name*, *crs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *crs* ⟹ *pats* ≠ []
  assumes ⋀*name crs pats rhs.* (*name*, *crs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *crs* ⟹ *fdisjnt* (*freess pats*) *all-consts*
  assumes ⋀*name crs pats rhs.* (*name*, *crs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *crs* ⟹ ¬ *shadows-consts rhs*
  assumes ⋀*name crs pats rhs.* (*name*, *crs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *crs* ⟹ *frees rhs* |⊆| *freess pats*
  assumes ⋀*name crs pats rhs.* (*name*, *crs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *crs* ⟹ *welldefined rhs*
  assumes *is-fmap rs rs* ≠ {||}
  shows *crules C-info rs*
⟨*proof*⟩

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

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

lemma *consts-of-heads*: *fst* |‘| *consts-of rs* = *heads-of rs*
⟨*proof*⟩

lemma (**in** *nrules*) *consts-rules*: *crules C-info* (*consts-of rs*)
⟨*proof*⟩

sublocale *nrules* ⊆ *nrules-as-crules?*: *crules C-info consts-of rs*
⟨*proof*⟩

3.2.7 Computability

export-code
  translate-rule consts-of arity nterm-to-term
  checking *Scala*

end

3.3 Higher-order term rewriting with explicit pattern matching

theory *Rewriting-Pterm-Elim*
imports
  Rewriting-Nterm
  ../Terms/Pterm
begin

3.3.1 Intermediate rule sets

type-synonym *irules* = (*term list* × *pterm*) *fset*

type-synonym *irule-set* = (*name* × *irules*) *fset*

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

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

lemma (**in** *pre-irules*) *irulesI*:
 assumes ⋀*name irs*. (*name*, *irs*) |∈| *rs* ⟹ *arity-compatibles irs*
 assumes ⋀*name irs*. (*name*, *irs*) |∈| *rs* ⟹ *is-fmap irs*
 assumes ⋀*name irs*. (*name*, *irs*) |∈| *rs* ⟹ *patterns-compatibles irs*
 assumes ⋀*name irs*. (*name*, *irs*) |∈| *rs* ⟹ *irs* ≠ {||}
 assumes ⋀*name irs pats rhs*. (*name*, *irs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *irs* ⟹ *linears pats*
 assumes ⋀*name irs pats rhs*. (*name*, *irs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *irs* ⟹ *abs-ish pats rhs*
 assumes ⋀*name irs pats rhs*. (*name*, *irs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *irs* ⟹ *fdisjnt* (*freess pats*) *all-consts*
 assumes ⋀*name irs pats rhs*. (*name*, *irs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *irs* ⟹ *closed-except rhs* (*freess pats*)
 assumes ⋀*name irs pats rhs*. (*name*, *irs*) |∈| *rs* ⟹ (*pats*, *rhs*) |∈| *irs* ⟹ *wellformed rhs*
 assumes ⋀*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*
⟨*proof*⟩

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

Translation from *nterm* to *pterm*

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

lemma *nterm-to-pterm-inj*: *nterm-to-pterm x* = *nterm-to-pterm y* ⟹ *x* = *y*
⟨*proof*⟩

lemma *nterm-to-pterm*:
  assumes *no-abs t*
  shows *nterm-to-pterm t* = *convert-term t*
⟨*proof*⟩

lemma *nterm-to-pterm-frees*[*simp*]: *frees* (*nterm-to-pterm t*) = *frees t*
⟨*proof*⟩

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

lemma (**in** *constants*) *shadows-nterm-to-pterm*[*simp*]: *shadows-consts* (*nterm-to-pterm t*) = *shadows-consts t*
⟨*proof*⟩

lemma *wellformed-nterm-to-pterm*[*intro*]: *wellformed* (*nterm-to-pterm t*)
⟨*proof*⟩

lemma *consts-nterm-to-pterm*[*simp*]: *consts* (*nterm-to-pterm t*) = *consts t*
⟨*proof*⟩

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

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

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

lemma *compile-heads*: *fst* |'| *compile rs* = *fst* |'| *rs*
⟨*proof*⟩

lemma (**in** *crules*) *compile-rules*: *irules C-info* (*compile rs*)
⟨*proof*⟩

sublocale *crules* ⊆ *crules-as-irules*: *irules C-info compile rs*
⟨*proof*⟩

Transformation of *irule-set*

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

lemma *arity-compatibles-transform-irules*:
  assumes *arity-compatibles rs*
  shows *arity-compatibles* (*transform-irules rs*)
⟨*proof*⟩

lemma *arity-transform-irules*:
  assumes *arity-compatibles rs rs* $\neq$ {||}
  shows *arity* (*transform-irules rs*) = (*if arity rs* = *0 then 0 else arity rs* − *1*)
⟨*proof*⟩

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

lemma *transform-irule-set-heads*: *fst* |'| *transform-irule-set rs* = *fst* |'| *rs*
⟨*proof*⟩

lemma (**in** *irules*) *rules-transform*: *irules C-info* (*transform-irule-set rs*)
⟨*proof*⟩

Matching and rewriting

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

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

lemma *irewrite-stepI*:
  assumes *match* (*name* \$\$ *pats*) *t* = *Some env subst rhs env* = *u*
  shows *name*, *pats*, *rhs* $\vdash_i$ *t* → *u*
⟨*proof*⟩

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

global-interpretation *irewrite*: *rewriting irewrite rs* **for** *rs*
⟨*proof*⟩

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

lemma (**in** *irules*) *irewrite-closed*:
  assumes *rs* $\vdash_i$ *t* $\longrightarrow$ *u closed t*
  shows *closed u*
⟨*proof*⟩

corollary (**in** *irules*) *irewrite-rt-closed*:
  assumes *rs* $\vdash_i$ *t* $\longrightarrow$* *u closed t*
  shows *closed u*
⟨*proof*⟩

Correctness of translation

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

global-interpretation *irelated*: *term-struct-rel-strong irelated*
⟨*proof*⟩

lemma *irelated-vars*: *t* $\approx_i$ *u* $\Longrightarrow$ *frees t* = *frees u*
⟨*proof*⟩

lemma *irelated-no-abs*:
  assumes *t* $\approx_i$ *u*
  shows *no-abs t* $\longleftrightarrow$ *no-abs u*
⟨*proof*⟩

lemma *irelated-subst*:
  assumes *t* $\approx_i$ *u irelated.P-env nenv penv*
  shows *subst t nenv* $\approx_i$ *subst u penv*
⟨*proof*⟩

lemma *related-irewrite-step*:
  assumes *name*, *pats*, *nterm-to-pterm rhs* $\vdash_i$ *u* $\rightarrow$ *u'* *t* $\approx_i$ *u*
  obtains *t'* **where** *unsplit-rule* (*name*, *pats*, *rhs*) $\vdash$ *t* $\rightarrow$ *t'* *t'* $\approx_i$ *u'*
⟨*proof*⟩

theorem (**in** *nrules*) *compile-correct*:
  assumes *compile* (*consts-of rs*) $\vdash_i$ *u* $\longrightarrow$ *u'* *t* $\approx_i$ *u closed t*
  obtains *t'* **where** *rs* $\vdash_n$ *t* $\longrightarrow$ *t'* *t'* $\approx_i$ *u'*
⟨*proof*⟩

corollary (**in** *nrules*) *compile-correct-rt*:

assumes *compile* (*consts-of rs*) $\vdash_i u \longrightarrow\!* \; u'$ $t \approx_i u$ *closed t*
obtains t' **where** *rs* $\vdash_n t \longrightarrow\!* \; t'$ $t' \approx_i u'$
⟨*proof*⟩

Completeness of translation

lemma (**in** *nrules*) *compile-complete*:
assumes *rs* $\vdash_n t \longrightarrow t'$ *closed t*
shows *compile* (*consts-of rs*) $\vdash_i$ *nterm-to-pterm t* $\longrightarrow$ *nterm-to-pterm* t'
⟨*proof*⟩

Correctness of transformation

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

context *irules* **begin**

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

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

lemma *prelated-refl*[*intro*!]: $t \approx_p t$
⟨*proof*⟩

sublocale *prelated*: *term-struct-rel prelated*
⟨*proof*⟩

lemma *prelated-pvars*:
assumes $t \approx_p u$
shows *frees t* = *frees u*
⟨*proof*⟩

corollary *prelated-closed*: $t \approx_p u \Longrightarrow$ *closed t* $\longleftrightarrow$ *closed u*
⟨*proof*⟩

lemma *prelated-no-abs-right*:
assumes $t \approx_p u$ *no-abs u*

shows $t = u$
⟨*proof*⟩

corollary *env-prelated-refl*[*intro*!]: *prelated.P-env env env*
⟨*proof*⟩

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

Instead, we use ⟦*match ?x ?u* = *Some ?env*; *?t* $\approx_p$ *?u*; $\bigwedge$*env*′. ⟦*match ?x ?t* = *Some env*′; *prelated.P-env env*′ *?env*⟧ $\Longrightarrow$ *?thesis*⟧ $\Longrightarrow$ *?thesis*.

lemma *prelated-subst*:
assumes $t_1 \approx_p t_2$ *prelated.P-env* env_1 env_2
shows *subst* t_1 $env_1 \approx_p$ *subst* t_2 env_2
⟨*proof*⟩

lemma *prelated-step*:
assumes *name*, *pats*, *rhs* $\vdash_i u \rightarrow u'$ $t \approx_p u$
obtains t' **where** *name*, *pats*, *rhs* $\vdash_i t \rightarrow t'$ $t' \approx_p u'$
⟨*proof*⟩

lemma *prelated-beta*: — same problem as *prelated.related-match*
assumes (*pat*, rhs_2) $\vdash t_2 \rightarrow u_2$ $rhs_1 \approx_p rhs_2$ $t_1 \approx_p t_2$
obtains u_1 **where** (*pat*, rhs_1) $\vdash t_1 \rightarrow u_1$ $u_1 \approx_p u_2$
⟨*proof*⟩

theorem *transform-correct*:
assumes *transform-irule-set rs* $\vdash_i u \longrightarrow u'$ $t \approx_p u$ *closed t*
obtains t' **where** *rs* $\vdash_i t \longrightarrow* t'$ — zero or one step **and** $t' \approx_p u'$
⟨*proof*⟩

end

Completeness of transformation

lemma (**in** *irules*) *transform-completeness*:
assumes *rs* $\vdash_i t \longrightarrow t'$ *closed t*
shows *transform-irule-set rs* $\vdash_i t \longrightarrow* t'$
⟨*proof*⟩

Computability

export-code
compile transform-irules
checking *Scala SML*

end

3.3.2 Pure pattern matching rule sets

theory *Rewriting-Pterm*
imports *Rewriting-Pterm-Elim*
begin

type-synonym *prule* = *name* × *pterm*

primrec *prule* :: *prule* ⇒ *bool* **where**
prule (-, *rhs*) ⟷ *wellformed rhs* ∧ *closed rhs* ∧ *is-abs rhs*

lemma *pruleI*[*intro*!]: *wellformed rhs* ⟹ *closed rhs* ⟹ *is-abs rhs* ⟹ *prule* (*name*, *rhs*)
⟨*proof*⟩

locale *prules* = *constants C-info fst* |‘| *rs* **for** *C-info* **and** *rs* :: *prule fset* +
  assumes *all-rules*: *fBall rs prule*
  assumes *fmap*: *is-fmap rs*
  assumes *not-shadows*: *fBall rs* (λ(-, *rhs*). ¬ *shadows-consts rhs*)
  assumes *welldefined-rs*: *fBall rs* (λ(-, *rhs*). *welldefined rhs*)

Rewriting

inductive *prewrite* :: *prule fset* ⇒ *pterm* ⇒ *pterm* ⇒ *bool* (-/ $\vdash_p$/ - ⟶/ - [*50*,*0*,*50*] *50*) **for** *rs* **where**
step: (*name*, *rhs*) |∈| *rs* ⟹ *rs* $\vdash_p$ *Pconst name* ⟶ *rhs* |
beta: *c* |∈| *cs* ⟹ *c* ⊢ *t* → t' ⟹ *rs* $\vdash_p$ *Pabs cs* $\$_p$ *t* ⟶ t' |
fun: *rs* $\vdash_p$ *t* ⟶ t' ⟹ *rs* $\vdash_p$ *t* $\$_p$ *u* ⟶ t' $\$_p$ *u* |
arg: *rs* $\vdash_p$ *u* ⟶ u' ⟹ *rs* $\vdash_p$ *t* $\$_p$ *u* ⟶ *t* $\$_p$ u'

global-interpretation *prewrite*: *rewriting prewrite rs* **for** *rs*
⟨*proof*⟩

abbreviation *prewrite-rt* :: *prule fset* ⇒ *pterm* ⇒ *pterm* ⇒ *bool* (-/ $\vdash_p$/ - ⟶∗/ - [*50*,*0*,*50*] *50*) **where**
prewrite-rt rs ≡ (*prewrite rs*)**

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

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

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

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

lemma *compile-heads*: *fst* |‘| *compile rs* = *fst* |‘| *rs*
⟨*proof*⟩

Correctness of translation

lemma *arity-zero-shape*:
  assumes *arity-compatibles rs arity rs* = *0 is-fmap rs rs* $\neq$ {||}
  obtains *t* **where** *rs* = {| ([], *t*) |}
⟨*proof*⟩

lemma (**in** *irules*) *compile-rules*:
  assumes *finished rs*
  shows *prules C-info* (*compile rs*)
⟨*proof*⟩

theorem (**in** *irules*) *compile-correct*:
  assumes *compile rs* $\vdash_p$ *t* $\longrightarrow$ *t'* *finished rs*
  shows *rs* $\vdash_i$ *t* $\longrightarrow$ *t'*
⟨*proof*⟩

theorem (**in** *irules*) *compile-complete*:
  assumes *rs* $\vdash_i$ *t* $\longrightarrow$ *t'* *finished rs*
  shows *compile rs* $\vdash_p$ *t* $\longrightarrow$ *t'*
⟨*proof*⟩

export-code
  compile finished
  checking *Scala*

end

3.4 Sequential pattern matching

theory *Rewriting-Sterm*
imports *Rewriting-Pterm*
begin

type-synonym *srule* = *name* × *sterm*

abbreviation *closed-srules* :: *srule list* ⇒ *bool* **where**
closed-srules ≡ *list-all* (*closed* ∘ *snd*)

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

lemma *sruleI*[*intro*!]: *wellformed rhs* ⟹ *closed rhs* ⟹ *is-abs rhs* ⟹ *srule* (*name*, *rhs*)
⟨*proof*⟩

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

lemma *map*: *is-map* (*set rs*)
⟨*proof*⟩

lemma *clausesE*:
  assumes (*name*, *rhs*) ∈ *set rs*
  obtains *cs* **where** *rhs* = *Sabs cs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma (**in** *srules*) *srewrite-stepI*: (*name*, *rhs*) ∈ *set rs* ⟹ *srewrite-step rs name rhs*
⟨*proof*⟩

hide-fact *srewrite-stepI0*

inductive *srewrite* :: *srule list* ⇒ *sterm* ⇒ *sterm* ⇒ *bool* (-/ $\vdash_s$/ - ⟶/ - [*50*,*0*,*50*] *50*) **for** *rs* **where**
step: *srewrite-step rs name rhs* ⟹ *rs* $\vdash_s$ *Sconst name* ⟶ *rhs* |
beta: *rewrite-first cs t t′* ⟹ *rs* $\vdash_s$ *Sabs cs* $\$_s$ *t* ⟶ *t′* |
fun: *rs* $\vdash_s$ *t* ⟶ *t′* ⟹ *rs* $\vdash_s$ *t* $\$_s$ *u* ⟶ *t′* $\$_s$ *u* |
arg: *rs* $\vdash_s$ *u* ⟶ *u′* ⟹ *rs* $\vdash_s$ *t* $\$_s$ *u* ⟶ *t* $\$_s$ *u′*

code-pred *srewrite* ⟨*proof*⟩

abbreviation *srewrite-rt* :: *srule list* ⇒ *sterm* ⇒ *sterm* ⇒ *bool* (-/ $\vdash_s$/ - ⟶*/ - [*50*,*0*,*50*] *50*) **where**
srewrite-rt rs ≡ (*srewrite rs*)**

global-interpretation *srewrite*: *rewriting srewrite rs* **for** *rs*
⟨*proof*⟩

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

Translation from *pterm* to *sterm*

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

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

lemma *pterm-to-sterm*:
  assumes *no-abs t*
  shows *pterm-to-sterm t* = *convert-term t*
⟨*proof*⟩

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

lemma *pterm-to-sterm-wellformed*:
  assumes *wellformed t*
  shows *wellformed* (*pterm-to-sterm t*)
⟨*proof*⟩
    including *fset.lifting* ⟨*proof*⟩

lemma *pterm-to-sterm-sterm-to-pterm*:
  assumes *wellformed t*
  shows *sterm-to-pterm* (*pterm-to-sterm t*) = *t*
⟨*proof*⟩

corollary *pterm-to-sterm-frees*: *wellformed t* ⟹ *frees* (*pterm-to-sterm t*) = *frees t*
⟨*proof*⟩

corollary *pterm-to-sterm-closed*:
  closed-except t S ⟹ *wellformed t* ⟹ *closed-except* (*pterm-to-sterm t*) *S*
⟨*proof*⟩

corollary *pterm-to-sterm-consts*: *wellformed t* ⟹ *consts* (*pterm-to-sterm t*) = *consts t*
⟨*proof*⟩

corollary (**in** *constants*) *pterm-to-sterm-shadows*:

wellformed t ⟹ *shadows-consts t* ⟷ *shadows-consts* (*pterm-to-sterm t*)
⟨*proof*⟩

definition *compile* :: *prule fset* ⇒ *srule list* **where**
compile rs = *ordered-fmap* (*map-prod id pterm-to-sterm* |\`| *rs*)

Correctness of translation

context *prules* **begin**

lemma *compile-heads*: *fst* |\`| *fset-of-list* (*compile rs*) = *fst* |\`| *rs*
⟨*proof*⟩

lemma *compile-rules*: *srules C-info* (*compile rs*)
⟨*proof*⟩
including *fset.lifting*
⟨*proof*⟩

sublocale *prules-as-srules*: *srules C-info compile rs*
⟨*proof*⟩

end

global-interpretation *srelated*: *term-struct-rel-strong* (*λp s. p* = *sterm-to-pterm s*)
⟨*proof*⟩

lemma *srelated-subst*:
assumes *srelated.P-env penv senv*
shows *subst* (*sterm-to-pterm t*) *penv* = *sterm-to-pterm* (*subst t senv*)
⟨*proof*⟩
including *fset.lifting*
⟨*proof*⟩

context begin

private lemma *srewrite-step-non-empty*: *srewrite-step rs′ name rhs* ⟹ *rs′* ≠ []
⟨*proof*⟩ **lemma** *compile-consE*:
assumes (*name*, *rhs′*) # *rest* = *compile rs is-fmap rs*
obtains *rhs* **where** *rhs′* = *pterm-to-sterm rhs* (*name*, *rhs*) |∈| *rs rest* = *compile* (*rs* − {| (*name*, *rhs*) |})
⟨*proof*⟩ **lemma** *compile-correct-step*:
assumes *srewrite-step* (*compile rs*) *name rhs is-fmap rs fBall rs prule*
shows (*name*, *sterm-to-pterm rhs*) |∈| *rs*
⟨*proof*⟩

lemma *compile-correct0*:
assumes *compile rs* $\vdash_s$ *u* ⟶ *u′* *prules C rs*
shows *rs* $\vdash_p$ *sterm-to-pterm u* ⟶ *sterm-to-pterm u′*

⟨*proof*⟩
  including *fset.lifting*
  ⟨*proof*⟩

end

lemma (**in** *prules*) *compile-correct*:
  assumes $compile\ rs \vdash_s u \longrightarrow u'$
  shows $rs \vdash_p sterm\text{-}to\text{-}pterm\ u \longrightarrow sterm\text{-}to\text{-}pterm\ u'$
⟨*proof*⟩

hide-fact *compile-correct0*

Completeness of translation

global-interpretation *srelated′*: *term-struct-rel-strong* ($\lambda p\ s.\ pterm\text{-}to\text{-}sterm\ p = s$)
⟨*proof*⟩

corollary *srelated-env-unique*:
  $srelated'.P\text{-}env\ penv\ senv \Longrightarrow srelated'.P\text{-}env\ penv\ senv' \Longrightarrow senv = senv'$
⟨*proof*⟩

lemma *srelated-subst′*:
  assumes *srelated′.P-env penv senv wellformed t*
  shows $pterm\text{-}to\text{-}sterm\ (subst\ t\ penv) = subst\ (pterm\text{-}to\text{-}sterm\ t)\ senv$
⟨*proof*⟩

lemma *srelated-find-match*:
  assumes $find\text{-}match\ cs\ t = Some\ (penv, pat, rhs)$ *srelated′.P-env penv senv*
  shows $find\text{-}match\ (map\ (map\text{-}prod\ id\ pterm\text{-}to\text{-}sterm)\ cs)\ (pterm\text{-}to\text{-}sterm\ t) = Some\ (senv, pat, pterm\text{-}to\text{-}sterm\ rhs)$
⟨*proof*⟩

lemma (**in** *prules*) *compile-complete*:
  assumes $rs \vdash_p t \longrightarrow t'$ *wellformed t*
  shows $compile\ rs \vdash_s pterm\text{-}to\text{-}sterm\ t \longrightarrow pterm\text{-}to\text{-}sterm\ t'$
⟨*proof*⟩

Computability

export-code *compile*
  checking *Scala*

end

3.5 Big-step semantics

theory *Big-Step-Sterm*

imports
Rewriting-Sterm
../Terms/Term-as-Value
begin

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

inductive (**in** *constructors*) *seval* :: *srule list* ⇒ (*name*, *sterm*) *fmap* ⇒ *sterm* ⇒ *sterm* ⇒ *bool* (-, -/ $\vdash_s$/ - ↓/ - [*50*,*0*,*50*] *50*) **for** *rs* **where**
const: (*name*, *rhs*) ∈ *set rs* ⟹ *rs*, Γ $\vdash_s$ *Sconst name* ↓ *rhs* |
var: *fmlookup* Γ *name* = *Some val* ⟹ *rs*, Γ $\vdash_s$ *Svar name* ↓ *val* |
abs: *rs*, Γ $\vdash_s$ *Sabs cs* ↓ *Sabs* (*map* (λ(*pat*, *t*). (*pat*, *subst t* (*fmdrop-fset* (*frees pat*) Γ))) *cs*) |
comb:
rs, Γ $\vdash_s$ *t* ↓ *Sabs cs* ⟹ *rs*, Γ $\vdash_s$ *u* ↓ *u′* ⟹
find-match cs u′ = *Some* (*env*, -, *rhs*) ⟹
rs, Γ $++_f$ *env* $\vdash_s$ *rhs* ↓ *val* ⟹
rs, Γ $\vdash_s$ *t* $\$_s$ *u* ↓ *val* |
constr:
name |∈| *C* ⟹
list-all2 (*seval rs* Γ) *ts us* ⟹
rs, Γ $\vdash_s$ *name* $$ *ts* ↓ *name* $$ *us*

lemma (**in** *constructors*) *seval-closed*:
assumes *rs*, Γ $\vdash_s$ *t* ↓ *u closed-srules rs closed-env* Γ *closed-except t* (*fmdom* Γ)
shows *closed u*
⟨*proof*⟩

lemma (**in** *srules*) *seval-wellformed*:
assumes *rs*, Γ $\vdash_s$ *t* ↓ *u wellformed t wellformed-env* Γ
shows *wellformed u*
⟨*proof*⟩

lemma (**in** *constants*) *seval-shadows*:
assumes *rs*, Γ $\vdash_s$ *t* ↓ *u* ¬ *shadows-consts t*
assumes *list-all* (λ(-, *rhs*). ¬ *shadows-consts rhs*) *rs*
assumes *not-shadows-consts-env* Γ
shows ¬ *shadows-consts u*
⟨*proof*⟩

lemma (**in** *constructors*) *seval-list-comb-abs*:
assumes *rs*, Γ $\vdash_s$ *name* $$ *args* ↓ *Sabs cs*
shows *name* ∈ *dom* (*map-of rs*)
⟨*proof*⟩

lemma (**in** *constructors*) *is-value-eval-id*:
assumes *is-value t closed t*
shows *rs*, Γ $\vdash_s$ *t* ↓ *t*
⟨*proof*⟩

lemma (**in** *constructors*) *ssubst-eval*:
  assumes *rs*, $\Gamma \vdash_s t \downarrow t'$ $\Gamma' \subseteq_f \Gamma$ *closed-env* Γ *value-env* Γ
  shows *rs*, $\Gamma \vdash_s$ *subst* $t\ \Gamma' \downarrow t'$
⟨*proof*⟩

lemma (**in** *constructors*) *seval-agree-eq*:
  assumes *rs*, $\Gamma \vdash_s t \downarrow u$ *fmrestrict-fset* $S\ \Gamma =$ *fmrestrict-fset* $S\ \Gamma'$ *closed-except* t S
  assumes S |⊆| *fmdom* Γ *closed-srules rs closed-env* Γ
  shows *rs*, $\Gamma' \vdash_s t \downarrow u$
⟨*proof*⟩
    including *fmap.lifting*
    ⟨*proof*⟩
        including *fmap.lifting fset.lifting*
        ⟨*proof*⟩

Correctness wrt *srewrite*

context *srules* **begin context begin**

private lemma *seval-correct0*:
  assumes *rs*, $\Gamma \vdash_s t \downarrow u$ *closed-except* t (*fmdom* Γ) *closed-env* Γ
  shows *rs* $\vdash_s$ *subst* $t\ \Gamma \longrightarrow\!* u$
⟨*proof*⟩

corollary *seval-correct*:
  assumes *rs*, *fmempty* $\vdash_s t \downarrow u$ *closed* t
  shows *rs* $\vdash_s t \longrightarrow\!* u$
⟨*proof*⟩

end end

end
theory *Big-Step-Value*
imports
  Big-Step-Sterm
  ../Terms/Value
begin

3.5.2 Big-step semantics evaluating to *value*

primrec *vrule* :: *vrule* ⇒ *bool* **where**
vrule (-, *rhs*) ⟷ *vwellformed rhs* ∧ *vclosed rhs* ∧ ¬ *is-Vconstr rhs*

locale *vrules* = *constants C-info fst* |'| *fset-of-list rs* **for** *C-info* **and** *rs* :: *vrule list* +
  assumes *all-rules*: *list-all vrule rs*
  assumes *distinct*: *distinct* (*map fst rs*)
  assumes *not-shadows*: *list-all* (λ(-, *rhs*). *not-shadows-vconsts rhs*) *rs*

  assumes *vconstructor-value-rs*: *vconstructor-value-rs rs*
  assumes *vwelldefined-rs*: *list-all* (λ(-, *rhs*). *vwelldefined rhs*) *rs*
begin

lemma *map*: *is-map* (*set rs*)
⟨*proof*⟩

end

abbreviation *value-to-sterm-rules* :: *vrule list* ⇒ *srule list* **where**
value-to-sterm-rules ≡ *map* (*map-prod id value-to-sterm*)

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

lemma (**in** *vrules*) *veval-wellformed*:
  assumes *rs*, Γ $\vdash_v$ *t* ↓ *v wellformed t wellformed-venv* Γ
  shows *vwellformed v*
⟨*proof*⟩

lemma (**in** *vrules*) *veval-closed*:
  assumes *rs*, Γ $\vdash_v$ *t* ↓ *v closed-except t* (*fmdom* Γ) *closed-venv* Γ
  assumes *wellformed t wellformed-venv* Γ
  shows *vclosed v*
⟨*proof*⟩

lemma (**in** *vrules*) *veval-constructor-value*:
  assumes *rs*, Γ $\vdash_v$ *t* ↓ *v vconstructor-value-env* Γ
  shows *vconstructor-value v*

⟨*proof*⟩

lemma (**in** *vrules*) *veval-welldefined*:
  assumes *rs*, $\Gamma \vdash_v t \downarrow v$ *fmpred* (λ-. *vwelldefined*) Γ *welldefined t*
  shows *vwelldefined v*
⟨*proof*⟩

Correctness wrt *constructors.seval*

context *vrules* **begin**

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

lemma *value-to-sterm-srules*: *srules C-info rs*′
⟨*proof*⟩

end

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

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

context *vrules* **begin**

context begin

private lemma *veval-correct0*:
  assumes *rs*, $\Gamma \vdash_v t \downarrow v$ *wellformed t wellformed-venv* Γ
  assumes *closed-except t* (*fmdom* Γ) *closed-venv* Γ
  assumes *vconstructor-value-env* Γ
  shows *rs*′, *fmmap value-to-sterm* $\Gamma \vdash_s t \downarrow$ *value-to-sterm v*
⟨*proof*⟩
    including *fmap.lifting fset.lifting*
    ⟨*proof*⟩
    including *fmap.lifting fset.lifting*
    ⟨*proof*⟩

lemma *veval-correct*:
  assumes *rs*, *fmempty* $\vdash_v t \downarrow v$ *wellformed t closed t*
  shows *rs*′, *fmempty* $\vdash_s t \downarrow$ *value-to-sterm v*

$\langle$*proof*$\rangle$

end end

end

3.5.3 Big-step semantics with conflation of constants and variables

theory *Big-Step-Value-ML*
imports *Big-Step-Value*
begin

definition *mk-rec-env* :: (*name*, *sclauses*) *fmap* $\Rightarrow$ (*name*, *value*) *fmap* $\Rightarrow$ (*name*, *value*) *fmap* **where**
mk-rec-env css Γ' = *fmmap-keys* (λ*name cs. Vrecabs css name* Γ') *css*

context *special-constants* **begin**

inductive *veval'* :: (*name*, *value*) *fmap* $\Rightarrow$ *sterm* $\Rightarrow$ *value* $\Rightarrow$ *bool* (-/ $\vdash_v$/ - $\downarrow$/ - [*50*,*0*,*50*] *50*) **where**
const: *name* $|\notin|$ *C* $\Longrightarrow$ *fmlookup* Γ *name* = *Some val* $\Longrightarrow$ $\Gamma \vdash_v$ *Sconst name* $\downarrow$ *val* |
var: *fmlookup* Γ *name* = *Some val* $\Longrightarrow$ $\Gamma \vdash_v$ *Svar name* $\downarrow$ *val* |
abs: $\Gamma \vdash_v$ *Sabs cs* $\downarrow$ *Vabs cs* Γ |
comb:
  $\Gamma \vdash_v$ *t* $\downarrow$ *Vabs cs* Γ' $\Longrightarrow$ $\Gamma \vdash_v$ *u* $\downarrow$ *u'* $\Longrightarrow$
  vfind-match cs u' = *Some* (*env*, -, *rhs*) $\Longrightarrow$
  Γ' $++_f$ *env* $\vdash_v$ *rhs* $\downarrow$ *val* $\Longrightarrow$
  $\Gamma \vdash_v$ *t* $\$_s$ *u* $\downarrow$ *val* |
rec-comb:
  $\Gamma \vdash_v$ *t* $\downarrow$ *Vrecabs css name* Γ' $\Longrightarrow$
  fmlookup css name = *Some cs* $\Longrightarrow$
  $\Gamma \vdash_v$ *u* $\downarrow$ *u'* $\Longrightarrow$
  vfind-match cs u' = *Some* (*env*, -, *rhs*) $\Longrightarrow$
  Γ' $++_f$ *mk-rec-env css* Γ' $++_f$ *env* $\vdash_v$ *rhs* $\downarrow$ *val* $\Longrightarrow$
  $\Gamma \vdash_v$ *t* $\$_s$ *u* $\downarrow$ *val* |
constr: *name* $|\in|$ *C* $\Longrightarrow$ *list-all2* (*veval'* Γ) *ts us* $\Longrightarrow$ $\Gamma \vdash_v$ *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* Γ *n* = *Some u'*
      vfind-match cs u' = *Some* (*env*, *pat*, *rhs*)
      Γ $++_f$ *env* $\vdash_v$ *rhs* $\downarrow$ *v*
$\langle$*proof*$\rangle$

lemma *veval'-wellformed*:

assumes $\Gamma \vdash_v t \downarrow v$ *wellformed t wellformed-venv* Γ
shows *vwellformed v*
⟨*proof*⟩

lemma (**in** *constants*) *veval′-shadows*:
assumes $\Gamma \vdash_v t \downarrow v$ *not-shadows-vconsts-env* Γ ¬ *shadows-consts t*
shows *not-shadows-vconsts v*
⟨*proof*⟩

lemma *veval′-closed*:
assumes $\Gamma \vdash_v t \downarrow v$ *closed-except t* (*fmdom* Γ) *closed-venv* Γ
assumes *wellformed t wellformed-venv* Γ
shows *vclosed v*
⟨*proof*⟩

primrec *vwelldefined′* :: *value* ⇒ *bool* **where**
vwelldefined′ (*Vconstr name vs*) ⟷ *list-all vwelldefined′ vs* |
vwelldefined′ (*Vabs cs* Γ) ⟷
pred-fmap id (*fmmap vwelldefined′* Γ) ∧
list-all (λ(*pat*, *t*). *consts t* |⊆| (*fmdom* Γ |∪| *C*)) *cs* ∧
fdisjnt C (*fmdom* Γ) |
vwelldefined′ (*Vrecabs css name* Γ) ⟷
pred-fmap id (*fmmap vwelldefined′* Γ) ∧
pred-fmap (λ*cs*.
list-all (λ(*pat*, *t*). *consts t* |⊆| *fmdom* Γ |∪| (*C* |∪| *fmdom css*)) *cs* ∧
fdisjnt C (*fmdom* Γ)) *css* ∧
name |∈| *fmdom css* ∧
fdisjnt C (*fmdom css*)

lemma *vmatch-welldefined′*:
assumes *vmatch pat v* = *Some env vwelldefined′ v*
shows *fmpred* (λ-. *vwelldefined′*) *env*
⟨*proof*⟩

lemma *sconsts-list-comb*:
consts (*list-comb f xs*) |⊆| *S* ⟷ *consts f* |⊆| *S* ∧ *list-all* (λ*x*. *consts x* |⊆| *S*) *xs*
⟨*proof*⟩

lemma *sconsts-sabs*:
consts (*Sabs cs*) |⊆| *S* ⟷ *list-all* (λ(-, *t*). *consts t* |⊆| *S*) *cs*
⟨*proof*⟩

lemma (**in** *constants*) *veval′-welldefined′*:
assumes $\Gamma \vdash_v t \downarrow v$ *fdisjnt C* (*fmdom* Γ)
assumes *consts t* |⊆| *fmdom* Γ |∪| *C fmpred* (λ-. *vwelldefined′*) Γ
assumes *wellformed t wellformed-venv* Γ
assumes ¬ *shadows-consts t not-shadows-vconsts-env* Γ
shows *vwelldefined′ v*

⟨*proof*⟩

end

Correctness wrt *veval*

context *vrules* **begin**

The following relation can be characterized as follows:

- Values have to have the same structure. (We prove an interpretation of *value-struct-rel*.)
- For closures, the captured environments must agree on constants and variables occurring in the body. The first *value* argument is from *veval* (i.e. from *CakeML-Codegen.Big-Step-Value*), the second from *veval'*.

coinductive *vrelated* :: *value* ⇒ *value* ⇒ *bool* ($\vdash_v$/ - ≈ - [*0*, *50*] *50*) **where**
constr: *list-all2 vrelated ts us* ⟹ $\vdash_v$ *Vconstr name ts* ≈ *Vconstr name us* |
abs:
  fmrel-on-fset (*frees* (*Sabs cs*)) *vrelated* Γ_1 Γ_2 ⟹
   fmrel-on-fset (*consts* (*Sabs cs*)) *vrelated* (*fmap-of-list rs*) Γ_2 ⟹
    $\vdash_v$ *Vabs cs* Γ_1 ≈ *Vabs cs* Γ_2 |
rec-abs:
  pred-fmap (λ*cs*.
   fmrel-on-fset (*frees* (*Sabs cs*)) *vrelated* Γ_1 Γ_2 ∧
   fmrel-on-fset (*consts* (*Sabs cs*)) *vrelated* (*fmap-of-list rs*) (Γ_2 $++_f$ *mk-rec-env css* Γ_2)) *css* ⟹
    name |∈| *fmdom css* ⟹
    $\vdash_v$ *Vrecabs css name* Γ_1 ≈ *Vrecabs css name* Γ_2

Perhaps unexpectedly, *vrelated* is not reflexive. The reason is that it does not just check syntactic equality including captured environments, but also adherence to the external rules.

sublocale *vrelated*: *value-struct-rel vrelated*
⟨*proof*⟩

The technically involved relation *vrelated* implies a weaker, but more intuitive property: If $\vdash_v$ $t \approx u$ then t and u are equal after termification (i.e. conversion with *value-to-sterm*). In fact, if both terms are ground terms, it collapses to equality. This follows directly from the interpretation of *value-struct-rel*.

lemma *veval'-correct*:
  assumes $\Gamma_2 \vdash_v t \downarrow v_2$ *wellformed t wellformed-venv* Γ_2
  assumes ¬ *shadows-consts t not-shadows-vconsts-env* Γ_2
  assumes *welldefined t*
  assumes *fmpred* (λ-. *vwelldefined*) Γ_1
  assumes *fmrel-on-fset* (*frees t*) *vrelated* Γ_1 Γ_2

assumes *fmrel-on-fset* (*consts* t) *vrelated* (*fmap-of-list* *rs*) Γ_2
obtains v_1 **where** *rs*, $\Gamma_1 \vdash_v t \downarrow v_1 \vdash_v v_1 \approx v_2$
⟨*proof*⟩

lemma *veval′-correct′*:
assumes $\Gamma_2 \vdash_v t \downarrow v_2$ *wellformed* t *wellformed-venv* Γ_2
assumes ¬ *shadows-consts* t *not-shadows-vconsts-env* Γ_2
assumes *welldefined* t
assumes *closed* t
assumes *fmrel-on-fset* (*consts* t) *vrelated* (*fmap-of-list* *rs*) Γ_2
obtains v_1 **where** *rs*, *fmempty* $\vdash_v t \downarrow v_1 \vdash_v v_1 \approx v_2$
⟨*proof*⟩

end

Preservation of extensional equality

lemma (**in** *constants*) *veval′-agree-eq*:
assumes $\Gamma \vdash_v t \downarrow v$ *fmrel-on-fset* (*ids* t) *erelated* Γ' Γ
assumes *closed-venv* Γ *closed-except* t (*fmdom* Γ)
assumes *wellformed* t *wellformed-venv* Γ *fdisjnt* C (*fmdom* Γ)
assumes *consts* t $|\subseteq|$ *fmdom* Γ $|\cup|$ C *fmpred* (λ-. *vwelldefined′*) Γ
assumes ¬ *shadows-consts* t *not-shadows-vconsts-env* Γ
obtains v' **where** $\Gamma' \vdash_v t \downarrow v'$ $v' \approx_e v$
⟨*proof*⟩

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 $\mathit{wellformed}\ t\ \mathit{rs} \vdash t \longrightarrow* t'\ \mathit{rs} \vdash t' \approx a$
shows $\mathit{rs} \vdash t \downarrow a$
$\langle \mathit{proof} \rangle$

lemma $\mathit{eval'E}[\mathit{elim}]$:
assumes $\mathit{rs} \vdash t \downarrow a$
obtains t' **where** $\mathit{wellformed}\ t\ \mathit{rs} \vdash t \longrightarrow* t'\ \mathit{rs} \vdash t' \approx a$
$\langle \mathit{proof} \rangle$

lemma *eval-trivI*: $\mathit{rs} \vdash t \approx a \Longrightarrow \mathit{rs} \vdash t \downarrow a$
$\langle \mathit{proof} \rangle$

lemma *eval-compose*:
assumes $\mathit{wellformed}\ t\ \mathit{rs} \vdash t \longrightarrow* t'\ \mathit{rs} \vdash t' \downarrow a$
shows $\mathit{rs} \vdash t \downarrow a$
$\langle \mathit{proof} \rangle$

end

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

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

instance
$\langle \mathit{proof} \rangle$

end

corollary *eval-funD*:
assumes $\mathit{rs} \vdash t \approx f\ \mathit{rs} \vdash t_x \downarrow x$
shows $\mathit{rs} \vdash t\ \$\ t_x \downarrow f\ x$
$\langle \mathit{proof} \rangle$

corollary $\mathit{eval'\text{-}funD}$:
assumes $\mathit{rs} \vdash t \downarrow f\ \mathit{rs} \vdash t_x \downarrow x$
shows $\mathit{rs} \vdash t\ \$\ t_x \downarrow f\ x$
$\langle \mathit{proof} \rangle$

lemma *eval-ext*:
assumes $\mathit{wellformed}\ f\ \bigwedge x\ t.\ \mathit{rs} \vdash t \downarrow x \Longrightarrow \mathit{rs} \vdash f\ \$\ t \downarrow a\ x$
shows $\mathit{rs} \vdash f \approx a$
$\langle \mathit{proof} \rangle$

lemma $\mathit{eval'\text{-}ext}$:
assumes $\mathit{wellformed}\ f\ \bigwedge x\ t.\ \mathit{rs} \vdash t \downarrow x \Longrightarrow \mathit{rs} \vdash f\ \$\ t \downarrow a\ x$
shows $\mathit{rs} \vdash f \downarrow a$
$\langle \mathit{proof} \rangle$

lemma *eval′-ext-alt*:
fixes $f :: {'a}{::}evaluate \Rightarrow {'b}{::}evaluate$
assumes $wellformed'\ 1\ t\ \bigwedge u\ x.\ rs \vdash u \downarrow x \Longrightarrow rs \vdash t\ [u]_\beta \downarrow f\ x$
shows $rs \vdash \Lambda\ t \downarrow f$
⟨*proof*⟩

lemma *eval-impl-wellformed*[*dest*]: $rs \vdash t \approx a \Longrightarrow wellformed'\ n\ t$
⟨*proof*⟩

lemma *eval′-impl-wellformed*[*dest*]: $rs \vdash t \downarrow a \Longrightarrow wellformed'\ n\ t$
⟨*proof*⟩

lemma *wellformed-unpack*:
$wellformed'\ n\ (t\ \$\ u) \Longrightarrow wellformed'\ n\ t$
$wellformed'\ n\ (t\ \$\ u) \Longrightarrow wellformed'\ n\ u$
$wellformed'\ n\ (\Lambda\ t) \Longrightarrow wellformed'\ (Suc\ n)\ t$
⟨*proof*⟩

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

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

context begin

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

private lemmas *rewrite-rt-fun* = *rewrite.rt-fun*[*unfolded app-term-def*]
private lemmas *rewrite-rt-arg* = *rewrite.rt-arg*[*unfolded app-term-def*]

⟨*ML*⟩

end

⟨*ML*⟩

end

4.2 Deep embedding of Pure terms into term-rewriting logic

theory *Embed*
imports
  Constructor-Funs.Constructor-Funs
  ../Utils/Code-Utils
  Eval-Class
keywords *embed* :: *thy-decl*
begin

fun *non-overlapping′* :: *term* $\Rightarrow$ *term* $\Rightarrow$ *bool* **where**
non-overlapping′ (*Const x*) (*Const y*) $\longleftrightarrow$ $x \neq y$ |
non-overlapping′ (*Const* -) (- \$ -) $\longleftrightarrow$ *True* |
non-overlapping′ (- \$ -) (*Const* -) $\longleftrightarrow$ *True* |
non-overlapping′ (t_1 \$ t_2) (u_1 \$ u_2) $\longleftrightarrow$ *non-overlapping′* t_1 u_1 $\vee$ *non-overlapping′* t_2 u_2 |
non-overlapping′ - - $\longleftrightarrow$ *False*

lemma *non-overlapping-approx*:
  assumes *non-overlapping′ t u*
  shows *non-overlapping t u*
⟨*proof*⟩

fun *pattern-compatible′* :: *term* $\Rightarrow$ *term* $\Rightarrow$ *bool* **where**
pattern-compatible′ (t_1 \$ t_2) (u_1 \$ u_2) $\longleftrightarrow$ *pattern-compatible′* t_1 u_1 $\wedge$ ($t_1 = u_1$ $\longrightarrow$ *pattern-compatible′* t_2 u_2) |
pattern-compatible′ t u $\longleftrightarrow$ $t = u$ $\vee$ *non-overlapping′ t u*

lemma *pattern-compatible-approx*:
  assumes *pattern-compatible′ t u*
  shows *pattern-compatible t u*
⟨*proof*⟩

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

definition *rules′* :: *C-info* $\Rightarrow$ *rule fset* $\Rightarrow$ *bool* **where**
rules′ C-info rs $\longleftrightarrow$
  fBall rs rule $\wedge$
  arity-compatibles rs $\wedge$
  is-fmap rs $\wedge$
  pattern-compatibles′ rs $\wedge$
  rs $\neq$ {||} $\wedge$
  fBall rs (λ(*lhs*, -). $\neg$ *pre-constants.shadows-consts C-info* (*heads-of rs*) *lhs*) $\wedge$

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

lemma *rules-approx*:
assumes *rules′ C-info rs*
shows *rules C-info rs*
⟨*proof*⟩

lemma *embed-ext*: $f \equiv g \Longrightarrow f\,x \equiv g\,x$
⟨*proof*⟩

⟨*ML*⟩

consts *lift-term* :: *′a* ⇒ *term* (⟨-⟩)

⟨*ML*⟩

end

4.3 Default instances

theory *Eval-Instances*
imports *Embed*
begin

⟨*ML*⟩

derive *evaluate nat bool list unit prod sum option char num name term*

end

Chapter 5

Final stage: Translation to CakeML

theory *Doc-Backend*
imports *Main*
begin

end

5.1 Basic CakeML setup

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

global-interpretation *name*: *rekey Name*
  rewrites *inv Name* = *as-string*
⟨*proof*⟩

global-interpretation *name-as-string*: *rekey as-string*
  ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fmap-of-nsBind*[*simp*]: *fmap-of-ns* (*nsBind* (*as-string k*) *v0 ns*) = *fmupd k v0* (*fmap-of-ns ns*)
⟨*proof*⟩

lemma *fmap-of-nsAppend*[*simp*]: *fmap-of-ns* (*nsAppend ns1 ns2*) = *fmap-of-ns ns2* $++_f$ *fmap-of-ns ns1*
⟨*proof*⟩

lemma *fmap-of-alist-to-ns*[*simp*]: *fmap-of-ns* (*alist-to-ns xs*) = *fmap-of-list* (*map* (*map-prod Name id*) *xs*)
⟨*proof*⟩

lemma *fmap-of-nsEmpty*[*simp*]: *fmap-of-ns nsEmpty* = *fmempty*
⟨*proof*⟩

context begin

private lemma *build-rec-env-fmap0*:
fmap-of-ns (*foldr* (λ(*f*, *x*, *e*). *nsBind f* (*Recclosure* env_Λ *funs' f*)) *funs env*) =
fmap-of-ns env $++_f$ *fmap-of-list* (*map* (λ(*f*, -). (*Name f*, *Recclosure* env_Λ *funs' f*)) *funs*)
⟨*proof*⟩

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

lemma *build-rec-env-fmap*:
fmap-of-ns (*build-rec-env funs* env_Λ *env*) = *fmap-of-ns env* $++_f$ *cake-mk-rec-env funs* env_Λ
⟨*proof*⟩

end

5.2 Constructors according to CakeML

definition *cake-tctor* :: *string* ⇒ *tctor* **where**

cake-tctor name = (*if name* = *''fun'' then Ast.TC-fn else Ast.TC-name* (*Short name*))

primrec *typ-to-t* :: *typ* ⇒ *Ast.t* **where**
typ-to-t (*TVar name*) = *Ast.Tvar* (*as-string name*) |
typ-to-t (*TApp name args*) = *Ast.Tapp* (*map typ-to-t args*) (*cake-tctor* (*as-string name*))

context *constructors* **begin**

definition *as-static-cenv* :: *c-ns* **where**
as-static-cenv = *Bind* (*rev* (*map* (*map-prod id* (*map-prod id* (*TypeId* ∘ *Short*))) *flat-C-info*)) []

lemma *as-static-cenv-cakeml-static-env*: *cakeml-static-env as-static-cenv*
⟨*proof*⟩

sublocale *cake-static-env?*: *cakeml-static-env as-static-cenv*
⟨*proof*⟩

definition *as-cake-type-def* :: *Ast.type-def* **where**
as-cake-type-def =
  map (λ(*name*, *dt-def*). (*map as-string* (*tparams dt-def*), *as-string name*,
    map (λ(*C*, *params*). (*as-string C*, *map typ-to-t params*))
      (*sorted-list-of-fmap* (*constructors dt-def*))))
  (*sorted-list-of-fmap C-info*)

definition *cake-dt-prelude* :: *Ast.dec* **where**
cake-dt-prelude = *Ast.Dtype empty-locs as-cake-type-def*

definition *cake-all-types* :: *tid-or-exn set* **where**
cake-all-types = (*TypeId* ∘ *Short* ∘ *as-string*) ' *fset all-tdefs*

definition *empty-state-with-types* :: *unit SemanticPrimitives.state* **where**
empty-state-with-types =
  ⦇ *clock* = *0*, *refs* = [], *ffi* = *empty-ffi-state*, *defined-types* = *cake-all-types*, *defined-mods* = {} ⦈

lemma *empty-state-with-types-alt-def*:
  empty-state-with-types = *empty-state* ⦇ *defined-types* := *cake-all-types* ⦈
⟨*proof*⟩

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
⟨*proof*⟩ **lemma** *env-types-update*: *build-tdefs* [] *as-cake-type-def* = *as-static-cenv*
⟨*proof*⟩ **lemmas** *evaluate-type* =
evaluate-dec.dtype1[
where *new-tdecs* = *cake-all-types* **and** *s* = *empty-state* **and** *mn* = [] **and** *tds* = *as-cake-type-def*,
unfolded state-types-update env-types-update,
folded empty-sem-env-def]

private lemma *type-defs-to-new-tdecs*:
type-defs-to-new-tdecs [] *as-cake-type-def* =
set (*map* (λ*name*. *TypeId* (*Short* (*as-string name*))) (*sorted-list-of-fset* (*fmdom C-info*)))
⟨*proof*⟩ **lemma** *cakeml-convoluted1*: *foldr* (λ(*n*, *ts*). (#) *n*) *ys xs* = *map fst ys* @ *xs*
⟨*proof*⟩ **lemma** *cakeml-convoluted2*: *foldr* (λ*x y*. *f x* @ *y*) *xs ys* = *concat* (*map f xs*) @ *ys*
⟨*proof*⟩ **lemma** *check-dup-ctors-alt-def*: *check-dup-ctors tds* ⟷ *distinct* (*tds* ≫= (λ(-, -, *cons*). *map fst cons*))
⟨*proof*⟩

lemma *evaluate-dec-prelude*:
evaluate-dec t [] *env empty-state cake-dt-prelude* (*empty-state-with-types*, *Rval empty-sem-env*)
⟨*proof*⟩

end

end

Computability

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

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

end

5.3 CakeML backend

theory *CakeML-Backend*

imports
  CakeML-Setup
  ../Terms/Value
  ../Rewriting/Rewriting-Sterm
begin

5.3.1 Compilation

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

lemma *mk-pat-cupcake*[*intro*]: *is-cupcake-pat* (*mk-ml-pat pat*)
⟨*proof*⟩

context begin

private fun *frees′* :: *term* ⇒ *name list* **where**
frees′ (*Free x*) = [*x*] |
frees′ (t_1 \$ t_2) = *frees′* t_2 @ *frees′* t_1 |
frees′ (Λ *t*) = *frees′ t* |
frees′ - = []

private lemma *frees′-eq*[*simp*]: *fset-of-list* (*frees′ t*) = *frees t*
⟨*proof*⟩ **lemma** *frees′-list-comb*: *frees′* (*list-comb f xs*) = *concat* (*rev* (*map frees′ xs*)) @ *frees′ f*
⟨*proof*⟩ **lemma** *frees′-distinct*: *linear pat* ⟹ *distinct* (*frees′ pat*)
⟨*proof*⟩ **fun** *pat-bindings′* :: *Ast.pat* ⇒ *name list* **where**
pat-bindings′ (*Ast.Pvar n*) = [*Name n*] |
pat-bindings′ (*Ast.Pcon - ps*) = *concat* (*rev* (*map pat-bindings′ ps*)) |
pat-bindings′ (*Ast.Pref p*) = *pat-bindings′ p* |
pat-bindings′ (*Ast.Ptannot p -*) = *pat-bindings′ p* |
pat-bindings′ - = []

private lemma *pat-bindings′-eq*:
  map Name (*pats-bindings ps xs*) = *concat* (*rev* (*map pat-bindings′ ps*)) @ *map Name xs*
  map Name (*pat-bindings p xs*) = *pat-bindings′ p* @ *map Name xs*
⟨*proof*⟩ **lemma** *pat-bindings′-empty-eq*: *map Name* (*pat-bindings p* []) = *pat-bindings′ p*
⟨*proof*⟩ **lemma** *pat-bindings′-eq-frees*: *linear p* ⟹ *pat-bindings′* (*mk-ml-pat* (*mk-pat p*)) = *frees′ p*
⟨*proof*⟩

lemma *mk-pat-distinct*: *linear pat* ⟹ *distinct* (*pat-bindings* (*mk-ml-pat* (*mk-pat pat*)) [])
⟨*proof*⟩

end

locale *cakeml* = *pre-constants*
begin

fun
mk-exp :: *name fset* ⇒ *sterm* ⇒ *exp* **and**
mk-clauses :: *name fset* ⇒ (*term* × *sterm*) *list* ⇒ (*Ast.pat* × *exp*) *list* **and**
mk-con :: *name fset* ⇒ *sterm* ⇒ *exp* **where**
mk-exp - (*Svar s*) = *Ast.Var* (*Short* (*as-string s*)) |
mk-exp - (*Sconst s*) = (*if s* |∈| *C then Ast.Con* (*Some* (*Short* (*as-string s*))) [] *else Ast.Var* (*Short* (*as-string s*))) |
mk-exp S (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*) ⇒
if c |∈| *C then Ast.Con* (*Some* (*Short* (*as-string c*))) (*map* (*mk-con S*) *args*) *else mk-exp S t*
| - ⇒ *mk-exp S t*) |
mk-clauses S cs = *map* (λ(*pat*, *t*). (*mk-ml-pat* (*mk-pat pat*), *mk-con* (*frees pat* |∪| *S*) *t*)) *cs*

context begin

private lemma *mk-exp-cupcake0*:
wellformed t ⟹ *is-cupcake-exp* (*mk-exp S t*)
wellformed-clauses cs ⟹ *cupcake-clauses* (*mk-clauses S cs*) ∧ *cake-linear-clauses* (*mk-clauses S cs*)
wellformed t ⟹ *is-cupcake-exp* (*mk-con S t*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *mk-letrec-cupcake*:
list-all (λ(-, -, *exp*). *is-cupcake-exp exp*) (*mk-letrec-body S rs*)
⟨*proof*⟩

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*
⟨*proof*⟩

5.3.2 Computability

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

5.3.3 Correctness of semantic functions

abbreviation *related-pat* :: *term* ⇒ *Ast.pat* ⇒ *bool* **where**
related-pat t p ≡ (*p* = *mk-ml-pat* (*mk-pat t*))

context *cakeml′* **begin**

inductive *related-exp* :: *sterm* ⇒ *exp* ⇒ *bool* **where**

var: *related-exp* (*Svar name*) (*Ast.Var* (*Short* (*as-string name*))) |
const: *name* |∉| *C* ⟹ *related-exp* (*Sconst name*) (*Ast.Var* (*Short* (*as-string name*)))
|
constr: *name* |∈| *C* ⟹ *list-all2 related-exp ts es* ⟹
    related-exp (*name* $\$\$$ *ts*) (*Ast.Con* (*Some* (*Short* (*as-string name*))) *es*) |
app: *related-exp* t_1 u_1 ⟹ *related-exp* t_2 u_2 ⟹ *related-exp* (t_1 $\$_s$ t_2) (*Ast.App Ast.Opapp* [u_1, u_2]) |
fun: *list-all2* (*rel-prod related-pat related-exp*) *cs ml-cs* ⟹
    n |∉| *ids* (*Sabs cs*) ⟹ *n* |∉| *all-consts* ⟹
      related-exp (*Sabs cs*) (*Ast.Fun* (*as-string n*) (*Ast.Mat* (*Ast.Var* (*Short* (*as-string n*))) *ml-cs*)) |
mat: *list-all2* (*rel-prod related-pat related-exp*) *cs ml-cs* ⟹
  related-exp scr ml-scr ⟹
    related-exp (*Sabs cs* $\$_s$ *scr*) (*Ast.Mat ml-scr ml-cs*)

lemma *related-exp-is-cupcake*:
  assumes *related-exp t e wellformed t*
  shows *is-cupcake-exp e*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *related-funE*:
  assumes *related-fun cs n e*
  obtains *ml-cs*
   where *e* = *Ast.Mat* (*Ast.Var* (*Short* (*as-string n*))) *ml-cs n* |∉| *ids* (*Sabs cs*) *n* |∉| *all-consts*
   and *list-all2* (*rel-prod related-pat related-exp*) *cs ml-cs*
⟨*proof*⟩

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*⟩

inductive *related-v* :: *value* ⇒ *v* ⇒ *bool* **where**
conv:

list-all2 related-v us vs ⟹
related-v (*Vconstr name us*) (*Conv* (*Some* (*as-string name*, -)) *vs*) |
closure:
related-fun cs n e ⟹
fmrel-on-fset (*ids* (*Sabs cs*)) *related-v* Γ (*fmap-of-ns* (*sem-env.v env*)) ⟹
related-v (*Vabs cs* Γ) (*Closure env* (*as-string n*) *e*) |
rec-closure:
fmrel-on-fset (*fbind* (*fmran css*) (*ids* ∘ *Sabs*)) *related-v* Γ (*fmap-of-ns* (*sem-env.v env*)) ⟹
fmrel (λ*cs*. λ(*n*, *e*). *related-fun cs n e*) *css* (*fmap-of-list* (*map* (*map-prod Name* (*map-prod Name id*)) *exps*)) ⟹
related-v (*Vrecabs css name* Γ) (*Recclosure env exps* (*as-string name*))

abbreviation *var-env* :: (*name*, *value*) *fmap* ⇒ (*string* × *v*) *list* ⇒ *bool* **where**
var-env Γ *ns* ≡ *fmrel related-v* Γ (*fmap-of-list* (*map* (*map-prod Name id*) *ns*))

lemma *related-v-ext*:
assumes *related-v v ml-v*
assumes $v' \approx_e v$
shows *related-v* v' *ml-v*
⟨*proof*⟩

context begin

private inductive *match-result-related* :: (*string* × *v*) *list* ⇒ (*string* × *v*) *list match-result* ⇒ (*name*, *value*) *fmap option* ⇒ *bool* **for** *eenv* **where**
no-match: *match-result-related eenv No-match None* |
error: *match-result-related eenv Match-type-error* - |
match: *var-env* Γ *eenv-m* ⟹ *match-result-related eenv* (*Match* (*eenv-m* @ *eenv*)) (*Some* Γ)

private corollary *match-result-related-empty*: *match-result-related eenv* (*Match eenv*) (*Some fmempty*)
⟨*proof*⟩ **fun** *is-Match* :: *'a match-result* ⇒ *bool* **where**
is-Match (*Match* -) ⟷ *True* |
is-Match - ⟷ *False*

lemma *cupcake-pmatch-related*:
assumes *related-v v ml-v*
shows *match-result-related eenv* (*cupcake-pmatch as-static-cenv* (*mk-ml-pat pat*) *ml-v eenv*) (*vmatch pat v*)
⟨*proof*⟩

lemma *match-all-related*:
assumes *list-all2* (*rel-prod related-pat related-exp*) *cs ml-cs*
assumes *list-all* (λ(*pat*, -). *linear pat*) *cs*
assumes *related-v v ml-v*
assumes *cupcake-match-result as-static-cenv ml-v ml-cs Bindv* = *Rval* (*ml-rhs*, *ml-pat*, *eenv*)

obtains *rhs pat* Γ **where**
ml-pat = *mk-ml-pat* (*mk-pat pat*)
related-exp rhs ml-rhs
vfind-match cs v = *Some* (Γ, *pat*, *rhs*)
var-env Γ *eenv*
⟨*proof*⟩

end end

end

5.3.4 Correctness of compilation

theory *CakeML-Correctness*
imports
CakeML-Backend
../Rewriting/Big-Step-Value-ML
begin

context *cakeml′* **begin**

lemma *mk-rec-env-related*:
assumes *fmrel* (λ*cs* (*n*, *e*). *related-fun cs n e*) *css* (*fmap-of-list* (*map* (*map-prod Name* (*map-prod Name id*)) *funs*))
assumes *fmrel-on-fset* (*fbind* (*fmran css*) (*ids* ∘ *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*⟩

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*)
ids t |⊆| *S* ⟹ *all-consts* |⊆| *S* ⟹ ¬ *shadows-consts t* ⟹ *related-exp t* (*mk-con S t*)
⟨*proof*⟩

context begin

private lemma *semantic-correctness0*:
fixes *exp*
assumes *cupcake-evaluate-single env exp r is-cupcake-all-env env*
assumes *fmrel-on-fset* (*ids t*) *related-v* Γ (*fmap-of-ns* (*sem-env.v env*))
assumes *related-exp t exp*
assumes *wellformed t wellformed-venv* Γ
assumes *closed-venv* Γ *closed-except t* (*fmdom* Γ)
assumes *fmpred* (λ-. *vwelldefined′*) Γ *consts t* |⊆| *fmdom* Γ |∪| *C*
assumes *fdisjnt C* (*fmdom* Γ)

assumes ¬ *shadows-consts t not-shadows-vconsts-env* Γ
shows *if-rval* (λ*ml-v*. ∃ *v*. $\Gamma \vdash_v t \downarrow v$ ∧ *related-v v ml-v*) *r*
⟨*proof*⟩

theorem *semantic-correctness*:
fixes *exp*
assumes *cupcake-evaluate-single env exp* (*Rval ml-v*) *is-cupcake-all-env env*
assumes *fmrel-on-fset* (*ids t*) *related-v* Γ (*fmap-of-ns* (*sem-env.v env*))
assumes *related-exp t exp*
assumes *wellformed t wellformed-venv* Γ
assumes *closed-venv* Γ *closed-except t* (*fmdom* Γ)
assumes *fmpred* (λ-. *vwelldefined′*) Γ *consts t* |⊆| *fmdom* Γ |∪| *C*
assumes *fdisjnt C* (*fmdom* Γ)
assumes ¬ *shadows-consts t not-shadows-vconsts-env* Γ
obtains *v* **where** $\Gamma \vdash_v t \downarrow v$ *related-v v ml-v*
⟨*proof*⟩

end end

end

5.4 Converting bytes to integers

theory *CakeML-Byte*
imports
CakeML.Evaluate-Single
Show.Show-Instances
begin

definition *pat* :: *Ast.pat* **where**
pat = *Ast.Pcon* (*Some* (*Short* ″*String-char-Char*″)) (*map* (λ*n*. *Ast.Pvar* (″*b*″ @ *show n*)) [*0*..<*8*])

definition *cake-plus* :: *exp* ⇒ *exp* ⇒ *exp* **where**
cake-plus t u = *Ast.App* (*Ast.Opn Ast.Plus*) [*t*, *u*]

lemma *cake-plus-correct*:
assumes *evaluate env s u* = (*s*′, *Rval* (*Litv* (*IntLit y*)))
assumes *evaluate env s*′ *t* = (*s*″, *Rval* (*Litv* (*IntLit x*)))
shows *evaluate env s* (*cake-plus t u*) = (*s*″, *Rval* (*Litv* (*IntLit* (*x* + *y*))))
⟨*proof*⟩

definition *cake-times* :: *exp* ⇒ *exp* ⇒ *exp* **where**
cake-times t u = *Ast.App* (*Ast.Opn Ast.Times*) [*t*, *u*]

lemma *cake-times-correct*:
assumes *evaluate env s u* = (*s*′, *Rval* (*Litv* (*IntLit y*)))
assumes *evaluate env s*′ *t* = (*s*″, *Rval* (*Litv* (*IntLit x*)))
shows *evaluate env s* (*cake-times t u*) = (*s*″, *Rval* (*Litv* (*IntLit* (*x* ∗ *y*))))

⟨*proof*⟩

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

definition *summands* :: *exp list* **where**
summands = *map* (λ*n*. *cake-times* (*Lit* (*IntLit* (*2* ^ *n*))) (*cake-int-of-bool* (*Ast.Var* (*Short* (''*b*'' @ *show n*))))) [*0*..<*8*]

definition *cake-int-of-byte* :: *exp* **where**
cake-int-of-byte =
Ast.Fun ''*x*'' (*Ast.Mat* (*Ast.Var* (*Short* ''*x*'')) [(*pat*, *foldl cake-plus* (*Lit* (*IntLit 0*)) *summands*)])

end

Chapter 6

Composition of phases and full compilation pipeline

theory *Doc-Compiler*
imports *Main*
begin

end

6.1 Composition of correctness results

theory *Composition*
imports *../Backend/CakeML-Correctness*
begin

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

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

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

lemma (**in** *prules*) *prewrite-closed*:
  assumes $rs \vdash_p t \longrightarrow t'$ *closed* t
  shows *closed* t'
⟨*proof*⟩

corollary (**in** *prules*) *prewrite-rt-closed*:
  assumes $rs \vdash_p t \longrightarrow* t'$ *closed* t
  shows *closed* t'
⟨*proof*⟩

corollary (**in** *irules*) *compile-correct-rt*:
  assumes *Rewriting-Pterm.compile* $rs \vdash_p t \longrightarrow* t'$ *finished* rs
  shows $rs \vdash_i t \longrightarrow* t'$
⟨*proof*⟩

6.1.2 Reflexive-transitive closure of *prules.compile-correct*.

lemma (**in** *prules*) *compile-correct-rt*:
assumes *Rewriting-Sterm.compile rs* $\vdash_s$ *u* $\longrightarrow*$ *u'*
shows *rs* $\vdash_p$ *sterm-to-pterm u* $\longrightarrow*$ *sterm-to-pterm u'*
⟨*proof*⟩

lemma *srewrite-stepD*:
assumes *srewrite-step rs name t*
shows (*name*, *t*) $\in$ *set rs*
⟨*proof*⟩

lemma (**in** *srules*) *srewrite-wellformed*:
assumes *rs* $\vdash_s$ *t* $\longrightarrow$ *t'* *wellformed t*
shows *wellformed t'*
⟨*proof*⟩

lemma (**in** *srules*) *srewrite-wellformed-rt*:
assumes *rs* $\vdash_s$ *t* $\longrightarrow*$ *t'* *wellformed t*
shows *wellformed t'*
⟨*proof*⟩

lemma *vno-abs-value-to-sterm*: *no-abs* (*value-to-sterm v*) $\longleftrightarrow$ *vno-abs v* **for** *v*
⟨*proof*⟩

6.1.3 Reflexive-transitive closure of *rules.compile-correct*.

corollary (**in** *rules*) *compile-correct-rt*:
assumes *compile* $\vdash_n$ *u* $\longrightarrow*$ *u'* *closed u*
shows *rs* $\vdash$ *nterm-to-term' u* $\longrightarrow*$ *nterm-to-term' u'*
⟨*proof*⟩

6.1.4 Reflexive-transitive closure of *irules.transform-correct*.

corollary (**in** *irules*) *transform-correct-rt*:
assumes *transform-irule-set rs* $\vdash_i$ *u* $\longrightarrow*$ *u''* *t* $\approx_p$ *u closed t*
obtains *t''* **where** *rs* $\vdash_i$ *t* $\longrightarrow*$ *t''* *t''* $\approx_p$ *u''*
⟨*proof*⟩

corollary (**in** *irules*) *transform-correct-rt-no-abs*:
assumes *transform-irule-set rs* $\vdash_i$ *t* $\longrightarrow*$ *u closed t no-abs u*
shows *rs* $\vdash_i$ *t* $\longrightarrow*$ *u*
⟨*proof*⟩

corollary *transform-correct-rt-n-no-abs0*:
assumes *irules C rs* (*transform-irule-set* ^^ *n*) *rs* $\vdash_i$ *t* $\longrightarrow*$ *u closed t no-abs u*
shows *rs* $\vdash_i$ *t* $\longrightarrow*$ *u*
⟨*proof*⟩

corollary (**in** *irules*) *transform-correct-rt-n-no-abs*:

assumes (*transform-irule-set* ^^ *n*) *rs* $\vdash_i$ *t* ⟶* *u* *closed t* *no-abs u*
shows *rs* $\vdash_i$ *t* ⟶* *u*
⟨*proof*⟩

hide-fact *transform-correct-rt-n-no-abs0*

6.1.5 Iterated application of *transform-irule-set*.

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

lemma *rules-transform-iter0*:
assumes *irules C-info rs*
shows *irules C-info* ((*transform-irule-set* ^^ *n*) *rs*)
⟨*proof*⟩

lemma (**in** *irules*) *rules-transform-iter*: *irules C-info* ((*transform-irule-set* ^^ *n*) *rs*)
⟨*proof*⟩

lemma *transform-irule-set-n-heads*: *fst* |‘| ((*transform-irule-set* ^^ *n*) *rs*) = *fst* |‘| *rs*
⟨*proof*⟩

hide-fact *rules-transform-iter0*

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

lemma *transform-irule-set-iter-heads*: *fst* |‘| *transform-irule-set-iter rs* = *fst* |‘| *rs*
⟨*proof*⟩

lemma (**in** *irules*) *finished-alt-def*: *finished rs* ⟷ *max-arity rs* = *0*
⟨*proof*⟩

lemma (**in** *irules*) *transform-finished-id*: *finished rs* ⟹ *transform-irule-set rs* = *rs*
⟨*proof*⟩

lemma (**in** *irules*) *max-arity-decr*: *max-arity* (*transform-irule-set rs*) = *max-arity rs* − *1*
⟨*proof*⟩

lemma *max-arity-decr'0*:
assumes *irules C rs*
shows *max-arity* ((*transform-irule-set* ^^ *n*) *rs*) = *max-arity rs* − *n*
⟨*proof*⟩

lemma (**in** *irules*) *max-arity-decr'*: *max-arity* ((*transform-irule-set* ^^ *n*) *rs*) =

max-arity rs $-$ *n*
⟨*proof*⟩

hide-fact *max-arity-decr′0*

lemma (**in** *irules*) *transform-finished*: *finished* (*transform-irule-set-iter rs*)
⟨*proof*⟩

Trick as described in § *7.1* in the locale manual.

locale *irules′* = *irules*

sublocale *irules′* ⊆ *irules′-as-irules*: *irules C-info transform-irule-set-iter rs*
⟨*proof*⟩

sublocale *crules* ⊆ *crules-as-irules′*: *irules′ C-info Rewriting-Pterm-Elim.compile rs*
⟨*proof*⟩

sublocale *irules′* ⊆ *irules′-as-prules*: *prules C-info Rewriting-Pterm.compile* (*transform-irule-set-iter rs*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *as-vrules-fst′*[*simp*]: *map fst as-vrules* = *map fst rs*
⟨*proof*⟩

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*⟩
    including *fset.lifting* ⟨*proof*⟩

lemma *srules-as-vrules*: *vrules C-info as-vrules*

⟨*proof*⟩

sublocale *srules-as-vrules*: *vrules C-info as-vrules*
⟨*proof*⟩

lemma *rs′-rs-eq*: *srules-as-vrules.rs′* = *rs*
⟨*proof*⟩

lemma *veval-correct*:
  fixes *v*
  assumes *as-vrules*, *fmempty* $\vdash_v$ *t* ↓ *v wellformed t closed t*
  shows *rs*, *fmempty* $\vdash_s$ *t* ↓ *value-to-sterm v*
⟨*proof*⟩

end

6.1.7 ML-style semantics

context *srules* **begin**

lemma *as-vrules-mk-rec-env*: *fmap-of-list as-vrules* = *mk-rec-env global-css fmempty*
⟨*proof*⟩

abbreviation (*input*) *vrelated* ≡ *srules-as-vrules.vrelated*
notation *srules-as-vrules.vrelated* ($\vdash_v$/ - ≈ - [*0*, *50*] *50*)

lemma *vrecabs-global-css-refl*:
  assumes *name* |∈| *fmdom global-css*
  shows $\vdash_v$ *Vrecabs global-css name fmempty* ≈ *Vrecabs global-css name fmempty*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *as-vrules-refl-C*: *fmrel-on-fset C vrelated* (*fmap-of-list as-vrules*) (*fmap-of-list as-vrules*)
⟨*proof*⟩

lemma *veval′-correct″*:
  fixes *t v*
  assumes *fmap-of-list as-vrules* $\vdash_v$ *t* ↓ *v*
  assumes *wellformed t*
  assumes ¬ *shadows-consts t*
  assumes *welldefined t*
  assumes *closed t*
  assumes *vno-abs v*
  shows *as-vrules*, *fmempty* $\vdash_v$ *t* ↓ *v*
⟨*proof*⟩

end

6.1.8 CakeML

context *srules* **begin**

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

lemma *compile-sem-env*:
evaluate-dec ck mn env state (*compile-group all-consts rs*) (*state*, *Rval* (*as-sem-env env*))
⟨*proof*⟩

lemma *compile-sem-env′*:
fun-evaluate-decs mn state env [(*compile-group all-consts rs*)] = (*state*, *Rval* (*as-sem-env env*))
⟨*proof*⟩

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* ⦈))
⟨*proof*⟩

lemma *compile-prog′*[*unfolded combine-dec-result.simps*, *simplified*]:
fun-evaluate-prog state env (*compile rs*) = (*state*, *combine-dec-result* (*as-sem-env env*) (*Rval* ⦇ *sem-env.v* = *nsEmpty*, *sem-env.c* = *nsEmpty* ⦈))
⟨*proof*⟩

definition *sem-env* :: *v sem-env* **where**
sem-env ≡ *extend-dec-env* (*as-sem-env empty-sem-env*) *empty-sem-env*

lemma *cupcake-sem-env*: *is-cupcake-all-env sem-env*
⟨*proof*⟩

lemma *sem-env-refl*: *fmrel related-v* (*fmap-of-list as-vrules*) (*fmap-of-ns* (*sem-env.v sem-env*))
⟨*proof*⟩

including *fmap.lifting*
⟨*proof*⟩

lemma *semantic-correctness′*:
assumes *cupcake-evaluate-single sem-env* (*mk-con all-consts t*) (*Rval ml-v*)
assumes *welldefined t closed t* ¬ *shadows-consts t wellformed t*
obtains *v* **where** *fmap-of-list as-vrules* $\vdash_v$ *t* ↓ *v related-v v ml-v*

⟨*proof*⟩

end

fun *cake-to-value* :: *v* ⇒ *value* **where**
cake-to-value (*Conv* (*Some* (*name*, -)) *vs*) = *Vconstr* (*Name name*) (*map cake-to-value vs*)

context *cakeml′* **begin**

lemma *cake-to-value-abs-free*:
  assumes *is-cupcake-value v cake-no-abs v*
  shows *vno-abs* (*cake-to-value v*)
  ⟨*proof*⟩

lemma *cake-to-value-related*:
  assumes *cake-no-abs v is-cupcake-value v*
  shows *related-v* (*cake-to-value v*) *v*
⟨*proof*⟩

lemma *related-v-abs-free-uniq*:
  assumes *related-v* v_1 *ml-v related-v* v_2 *ml-v cake-no-abs ml-v*
  shows $v_1 = v_2$
⟨*proof*⟩

corollary *related-v-abs-free-cake-to-value*:
  assumes *related-v v ml-v cake-no-abs ml-v is-cupcake-value ml-v*
  shows *v* = *cake-to-value ml-v*
⟨*proof*⟩

end

context *srules* **begin**

lemma *cupcake-sem-env-preserve*:
  assumes *cupcake-evaluate-single sem-env* (*mk-con S t*) (*Rval ml-v*) *wellformed t*
  shows *is-cupcake-value ml-v*
⟨*proof*⟩

lemma *semantic-correctness″*:
  assumes *cupcake-evaluate-single sem-env* (*mk-con all-consts t*) (*Rval ml-v*)
  assumes *welldefined t closed t* ¬ *shadows-consts t wellformed t*
  assumes *cake-no-abs ml-v*
  shows *fmap-of-list as-vrules* $\vdash_v$ *t* ↓ *cake-to-value ml-v*
⟨*proof*⟩

end

6.1.9 Composition

context *rules* **begin**

abbreviation *term-to-nterm* **where**
term-to-nterm t ≡ *fresh-frun* (*Term-to-Nterm.term-to-nterm* [] *t*) *all-consts*

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

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

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

definition *compiled* ≡ *rules-as-nrules.crules-as-irules′.irules′-as-prules.prules-as-srules.as-vrules*

lemma *fmdom-compiled*: *fmdom* (*fmap-of-list compiled*) = *heads-of rs*
⟨*proof*⟩

lemma *cake-semantic-correctness*:
  assumes *cupcake-evaluate-single cake-sem-env* (*sterm-to-cake t*) (*Rval ml-v*)
  assumes *welldefined t closed t* ¬ *shadows-consts t wellformed t*
  assumes *cake-no-abs ml-v*
  shows *fmap-of-list compiled* $\vdash_v$ *t* ↓ *cake-to-value ml-v*
⟨*proof*⟩

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*⟩

end

end

6.2 Executable compilation chain

theory *Compiler*
imports *Composition*
begin

definition *term-to-exp* :: *C-info* ⇒ *rule fset* ⇒ *term* ⇒ *exp* **where**
term-to-exp C-info rs t =
  cakeml.mk-con C-info (*heads-of rs* |∪| *constructors.C C-info*)
    (*pterm-to-sterm* (*nterm-to-pterm* (*fresh-frun* (*term-to-nterm* [] *t*) (*heads-of rs* |∪| *constructors.C C-info*))))

lemma (**in** *rules*) *Compiler.term-to-exp C-info rs* = *term-to-cake*
  ⟨*proof*⟩

primrec *compress-pterm* :: *pterm* ⇒ *pterm* **where**
compress-pterm (*Pabs cs*) = *Pabs* (*fcompress* (*map-prod id compress-pterm* |‘| *cs*))
|
compress-pterm (*Pconst name*) = *Pconst name* |
compress-pterm (*Pvar name*) = *Pvar name* |
compress-pterm (*t* $\$_p$ *u*) = *compress-pterm t* $\$_p$ *compress-pterm u*

lemma *compress-pterm-eq*[*simp*]: *compress-pterm t* = *t*
⟨*proof*⟩

definition *compress-crule-set* :: *crule-set* ⇒ *crule-set* **where**
compress-crule-set = *fcompress* ∘ *fimage* (*map-prod id fcompress*)

definition *compress-irule-set* :: *irule-set* ⇒ *irule-set* **where**
compress-irule-set = *fcompress* ∘ *fimage* (*map-prod id* (*fcompress* ∘ *fimage* (*map-prod id compress-pterm*)))

definition *compress-prule-set* :: *prule fset* ⇒ *prule fset* **where**
compress-prule-set = *fcompress* ∘ *fimage* (*map-prod id compress-pterm*)

lemma *compress-crule-set-eq*[*simp*]: *compress-crule-set rs* = *rs*
⟨*proof*⟩

lemma *compress-irule-set-eq*[*simp*]: *compress-irule-set rs* = *rs*
⟨*proof*⟩

lemma *compress-prule-set*[*simp*]: *compress-prule-set rs* = *rs*
⟨*proof*⟩

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

definition *as-sem-env* :: *C-info* ⇒ *srule list* ⇒ *v sem-env* ⇒ *v sem-env* **where**

as-sem-env C-info rs env =
⦇ *sem-env.v* =
build-rec-env (*cakeml.mk-letrec-body C-info* (*fset-of-list* (*map fst rs*) |∪| *constructors.C C-info*) *rs*) *env nsEmpty*,
sem-env.c =
nsEmpty ⦈

definition *empty-sem-env* :: *C-info* ⇒ *v sem-env* **where**
empty-sem-env C-info = ⦇ *sem-env.v* = *nsEmpty*, *sem-env.c* = *constructors.as-static-cenv C-info* ⦈

definition *sem-env* :: *C-info* ⇒ *srule list* ⇒ *v sem-env* **where**
sem-env C-info rs = *extend-dec-env* (*as-sem-env C-info rs* (*empty-sem-env C-info*)) (*empty-sem-env C-info*)

definition *compile* :: *C-info* ⇒ *rule fset* ⇒ *Ast.prog* **where**
compile C-info =
CakeML-Backend.compile′ C-info ∘
Rewriting-Sterm.compile ∘
compress-prule-set ∘
Rewriting-Pterm.compile ∘
transform-irule-set-iter ∘
compress-irule-set ∘
Rewriting-Pterm-Elim.compile ∘
compress-crule-set ∘
Rewriting-Nterm.consts-of ∘
fcompress ∘
Rewriting-Nterm.compile′ C-info ∘
fcompress

definition *compile-to-env* :: *C-info* ⇒ *rule fset* ⇒ *v sem-env* **where**
compile-to-env C-info =
sem-env C-info ∘
Rewriting-Sterm.compile ∘
compress-prule-set ∘
Rewriting-Pterm.compile ∘
transform-irule-set-iter ∘
compress-irule-set ∘
Rewriting-Pterm-Elim.compile ∘
compress-crule-set ∘
Rewriting-Nterm.consts-of ∘
fcompress ∘
Rewriting-Nterm.compile′ C-info ∘
fcompress

lemma (**in** *rules*) *Compiler.compile-to-env C-info rs* = *rules.cake-sem-env C-info rs*
⟨*proof*⟩

export-code
term-to-exp compile compile-to-env
checking *Scala*

end