

# A Verified Code Generator from Isabelle/HOL to CakeML

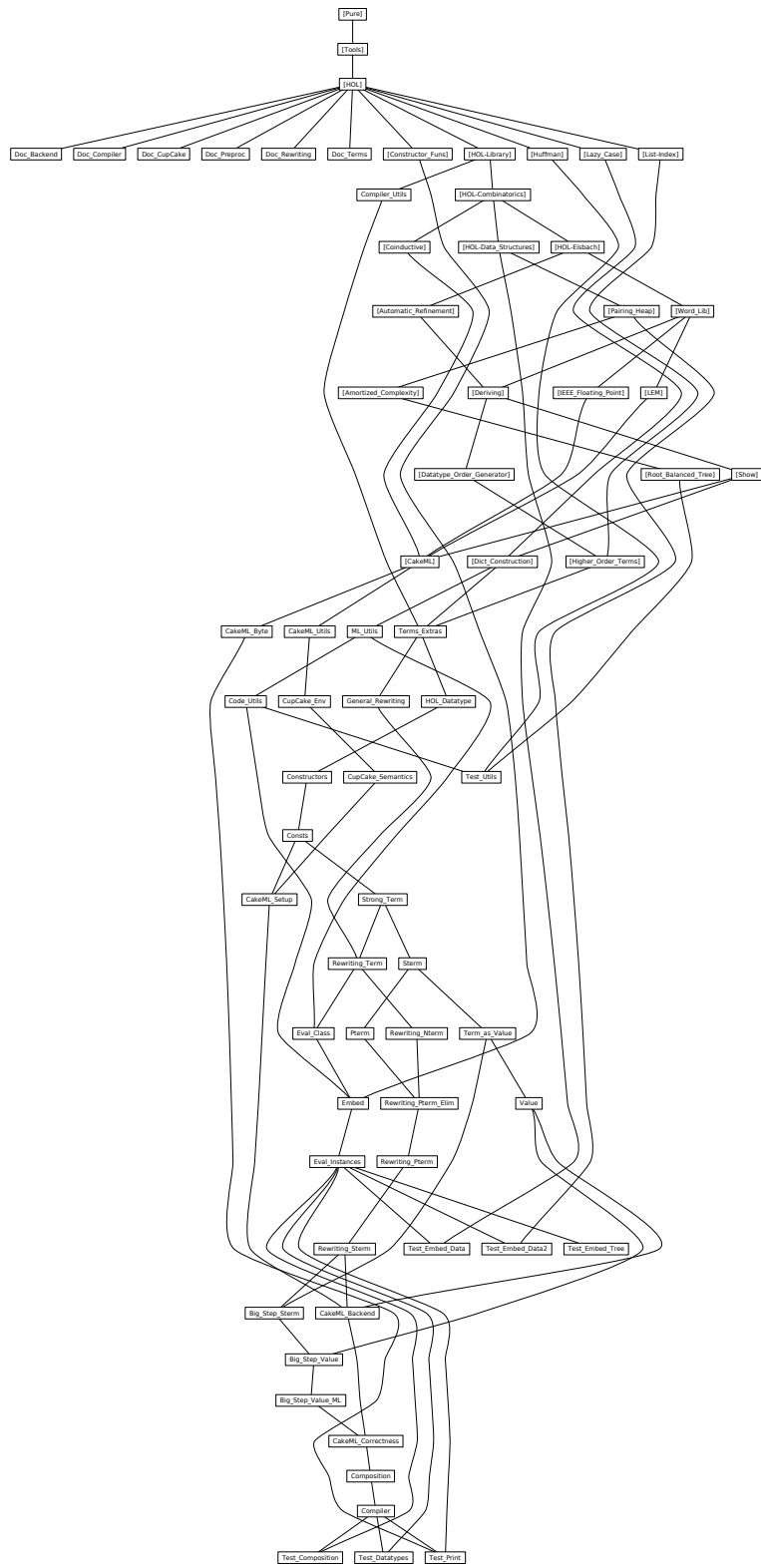
Lars Hupel

December 14, 2021

# Contents

<b>1</b>	<b>Terms</b>	<b>3</b>
1.1	Additional material over the <i>Higher-Order-Terms</i> AFP entry	3
1.2	Reflecting HOL datatype definitions	8
1.3	Constructor information	8
1.4	Special constants	9
1.5	Term algebra extended with wellformedness	10
1.6	Terms with sequential pattern matching	12
1.7	Terms with explicit pattern matching	16
1.8	Irreducible terms (values)	20
1.9	Viewing <i>sterm</i> as values	20
1.10	A dedicated value type	21
<b>2</b>	<b>A smaller version of CakeML: <i>CupCakeML</i></b>	<b>30</b>
2.1	CupCake environments	30
2.2	CupCake semantics	32
<b>3</b>	<b>Term rewriting</b>	<b>40</b>
3.1	Higher-order term rewriting using de-Bruijn indices	41
3.1.1	Matching and rewriting	41
3.1.2	Wellformedness	41
3.2	Higher-order term rewriting using explicit bound variable names	42
3.2.1	Definitions	42
3.2.2	Matching and rewriting	43
3.2.3	Translation from <i>Term-Class.term</i> to <i>nterm</i>	43
3.2.4	Correctness of translation	44
3.2.5	Completeness of translation	44
3.2.6	Splitting into constants	45
3.2.7	Computability	46
3.3	Higher-order term rewriting with explicit pattern matching	46
3.3.1	Intermediate rule sets	46
3.3.2	Pure pattern matching rule sets	53
3.4	Sequential pattern matching	54
3.5	Big-step semantics	58

3.5.1	Big-step semantics evaluating to irreducible <i>sterms</i> . . .	59
3.5.2	Big-step semantics evaluating to <i>value</i> . . . . .	60
3.5.3	Big-step semantics with conflation of constants and variables . . . . .	63
<b>4</b>	<b>Preprocessing of code equations</b>	<b>67</b>
4.1	A type class for correspondence between HOL expressions and terms . . . . .	67
4.2	Deep embedding of Pure terms into term-rewriting logic . . .	70
4.3	Default instances . . . . .	71
<b>5</b>	<b>Final stage: Translation to CakeML</b>	<b>72</b>
5.1	Basic CakeML setup . . . . .	72
5.2	Constructors according to CakeML . . . . .	73
5.2.1	Running the generated type declarations through the semantics . . . . .	74
5.3	CakeML backend . . . . .	75
5.3.1	Compilation . . . . .	76
5.3.2	Computability . . . . .	78
5.3.3	Correctness of semantic functions . . . . .	78
5.3.4	Correctness of compilation . . . . .	81
5.4	Converting bytes to integers . . . . .	82
<b>6</b>	<b>Composition of phases and full compilation pipeline</b>	<b>84</b>
6.1	Composition of correctness results . . . . .	84
6.1.1	Reflexive-transitive closure of <i>irules.compile-correct</i> . . .	84
6.1.2	Reflexive-transitive closure of <i>prules.compile-correct</i> . . .	85
6.1.3	Reflexive-transitive closure of <i>rules.compile-correct</i> . . .	85
6.1.4	Reflexive-transitive closure of <i>irules.transform-correct</i> . . .	85
6.1.5	Iterated application of <i>transform-irule-set</i> . . . . .	86
6.1.6	Big-step semantics . . . . .	87
6.1.7	ML-style semantics . . . . .	88
6.1.8	CakeML . . . . .	89
6.1.9	Composition . . . . .	91
6.2	Executable compilation chain . . . . .	92



# 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)
```

$\langle ML \rangle$

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

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

*<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 (t1 $ t2) (u1 $ u2)  $\longleftrightarrow$  pattern-compatible t1 u1  $\wedge$  (t1 = u1  $\longrightarrow$   
pattern-compatible t2 u2) |  
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-compatible-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*  $\Rightarrow$  *term list*  $\Rightarrow$  *bool* **where**  
*patterns-compatible*  $\equiv$  *rev-accum-rel pattern-compatible*

**abbreviation** *patterns-compatibles* :: (*term list*  $\times$  'a) *fset*  $\Rightarrow$  *bool* **where**  
*patterns-compatibles*  $\equiv$  *fpairwise* ( $\lambda$ (*pats*<sub>1</sub>, -) (*pats*<sub>2</sub>, -). *patterns-compatible* *pats*<sub>1</sub> *pats*<sub>2</sub>)

**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*  $\longleftrightarrow$  *t* = *u*  $\vee$  *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**  $\neg$  *patterns-compatible* *pats1* *pats2*  
  ⟨*proof*⟩

**end**

**abbreviation** *pattern-compatibles* :: (*term*  $\times$  'a) *fset*  $\Rightarrow$  *bool* **where**  
*pattern-compatibles*  $\equiv$  *fpairwise* ( $\lambda$ (*lhs*<sub>1</sub>, -) (*lhs*<sub>2</sub>, -). *pattern-compatible* *lhs*<sub>1</sub> *lhs*<sub>2</sub>)

**corollary** *match-compatible-pat-eq*:  
  **assumes** *pattern-compatible* *t*<sub>1</sub> *t*<sub>2</sub> *linear* *t*<sub>1</sub> *linear* *t*<sub>2</sub>  
  **assumes** *match* *t*<sub>1</sub> *u* = *Some* *env*<sub>1</sub> *match* *t*<sub>2</sub> *u* = *Some* *env*<sub>2</sub>  
  **shows** *t*<sub>1</sub> = *t*<sub>2</sub>  
  ⟨*proof*⟩



**corollary** *match-compatible-env-eq*:

**assumes** *pattern-compatible*  $t_1$   $t_2$  *linear*  $t_1$  *linear*  $t_2$   
**assumes** *match*  $t_1$   $u = \text{Some } env_1$  *match*  $t_2$   $u = \text{Some } env_2$   
**shows**  $env_1 = env_2$

*<proof>*

**corollary** *matches-compatible-eq*:

**assumes** *patterns-compatible*  $ts_1$   $ts_2$  *linears*  $ts_1$  *linears*  $ts_2$   
**assumes** *matches*  $ts_1$   $us = \text{Some } env_1$  *matches*  $ts_2$   $us = \text{Some } env_2$   
**shows**  $ts_1 = ts_2$   $env_1 = env_2$

*<proof>*

**lemma** *compatible-find-match*:

**assumes** *pattern-compatibles*  $(\text{fset-of-list } cs)$  *list-all*  $(\text{linear} \circ \text{fst})$   $cs$  *is-fmap*  
 $(\text{fset-of-list } cs)$

**assumes** *match*  $pat$   $t = \text{Some } env$   $(pat, rhs) \in \text{set } cs$

**shows** *find-match*  $cs$   $t = \text{Some } (env, pat, rhs)$

*<proof>*

**context** *term begin*

**definition** *arity-compatible*  $:: 'a \Rightarrow 'a \Rightarrow \text{bool}$  **where**

*arity-compatible*  $t_1$   $t_2 =$  (  
  *let*  
     $(head_1, pats_1) = \text{strip-comb } t_1;$   
     $(head_2, pats_2) = \text{strip-comb } t_2$   
  *in*  $head_1 = head_2 \longrightarrow \text{length } pats_1 = \text{length } pats_2$   
)

**abbreviation** *arity-compatibles*  $:: ('a \times 'b)$  *fset*  $\Rightarrow \text{bool}$  **where**

*arity-compatibles*  $\equiv \text{fpairwise } (\lambda(lhs_1, -) (lhs_2, -). \text{arity-compatible } lhs_1 \text{ } lhs_2)$

**definition** *head*  $:: 'a \Rightarrow \text{name}$  **where**

*head*  $t \equiv \text{const-name } (\text{fst } (\text{strip-comb } t))$

**abbreviation** *heads-of*  $:: (\text{term} \times 'a)$  *fset*  $\Rightarrow \text{name fset}$  **where**

*heads-of*  $rs \equiv (\text{head} \circ \text{fst}) \upharpoonright rs$

**end**

**definition** *arity*  $:: ('a \text{ list} \times 'b)$  *fset*  $\Rightarrow \text{nat}$  **where**

*arity*  $rs = \text{fthe-elem}' ((\text{length} \circ \text{fst}) \upharpoonright rs)$

**lemma** *arityI*:

**assumes** *fBall*  $rs$   $(\lambda(pats, -). \text{length } pats = n)$   $rs \neq \{\}\}$

**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 |1 constructors |1 fmrn C-info)

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

end

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

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

end

```

## 1.4 Special constants

```

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

```

```

locale special-constants = constructors

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

definition all-consts :: name fset where
all-consts = heads  $\cup$  C

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

sublocale welldefined: simple-syntactic-and welldefined
   $\langle$ proof $\rangle$ 

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)
   $\langle$ proof $\rangle$ 

end

end

```

## 1.5 Term algebra extended with wellformedness

```

theory Strong-Term
imports Consts
begin

class pre-strong-term = term +
  fixes wellformed :: 'a  $\Rightarrow$  bool
  fixes all-frees :: 'a  $\Rightarrow$  name fset
  assumes wellformed-const[simp]: wellformed (const name)
  assumes wellformed-free[simp]: wellformed (free name)
  assumes wellformed-app[simp]: wellformed (app u1 u2)  $\longleftrightarrow$  wellformed u1  $\wedge$ 
wellformed u2
  assumes all-frees-const[simp]: all-frees (const name) = fempty
  assumes all-frees-free[simp]: all-frees (free name) = { | name | }

```

```

assumes all-frees-app[simp]: all-frees (app  $u_1$   $u_2$ ) = all-frees  $u_1$  | $\cup$ | all-frees  $u_2$ 
begin

abbreviation wellformed-env :: (name, 'a) fmap  $\Rightarrow$  bool where
wellformed-env  $\equiv$  fmpred ( $\lambda$ -. wellformed)

end

context pre-constants begin

definition shadows-consts :: 'a::pre-strong-term  $\Rightarrow$  bool where
shadows-consts  $t \iff \neg$  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  $\Rightarrow$  bool
where
not-shadows-consts-env  $\equiv$  fmpred ( $\lambda$ -.  $s$ .  $\neg$  shadows-consts  $s$ )

end

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

class strong-term = pre-strong-term +
  assumes raw-frees-all-frees: abs-pred ( $\lambda$  $t$ . frees  $t$  | $\subseteq$ | all-frees  $t$ )  $t$ 
  assumes raw-subst-wellformed: abs-pred ( $\lambda$  $t$ . wellformed  $t \longrightarrow (\forall$  env. wellformed-env
env  $\longrightarrow$  wellformed (subst  $t$  env)))  $t$ 
begin

lemma frees-all-frees: frees  $t$  | $\subseteq$ | all-frees  $t$ 
  <proof>

lemma subst-wellformed: wellformed  $t \implies$  wellformed-env env  $\implies$  wellformed
(subst  $t$  env)
  <proof>

end

global-interpretation wellformed: subst-syntactic-and wellformed :: 'a::strong-term
 $\Rightarrow$  bool
  <proof>

instantiation term :: strong-term begin

fun all-frees-term :: term  $\Rightarrow$  name fset where
all-frees-term (Free  $x$ ) = { $x$ } |
all-frees-term ( $t_1$  $  $t_2$ ) = all-frees-term  $t_1$  | $\cup$ | all-frees-term  $t_2$  |
all-frees-term ( $\Lambda$   $t$ ) = all-frees-term  $t$  |

```

*all-frees-term* - = {||}

**lemma** *frees-all-frees-term*[simp]: *all-frees t = frees (t::term)*  
<proof>

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

**instance** <proof>

**end**

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

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

**fun** *all-frees-nterm* :: *nterm*  $\Rightarrow$  *name fset* **where**  
*all-frees-nterm (Nvar x) = { | x | }* |  
*all-frees-nterm (t<sub>1</sub> \$<sub>n</sub> t<sub>2</sub>) = all-frees-nterm t<sub>1</sub> | $\cup$ | all-frees-nterm t<sub>2</sub> |*  
*all-frees-nterm ( $\Lambda_n x. t$ ) = 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**  $\neg$  *shadows-consts t*  $\Longrightarrow$  *fdisjnt all-consts (frees t)*  
<proof>

**abbreviation** *wellformed-clauses* :: -  $\Rightarrow$  *bool* **where**  
*wellformed-clauses cs*  $\equiv$  *list-all* ( $\lambda(pat, t). linear pat \wedge wellformed t$ ) *cs*  $\wedge distinct$   
*(map fst cs) \wedge cs \neq []*

**end**

## 1.6 Terms with sequential pattern matching

**theory** *Sterm*  
**imports** *Strong-Term*  
**begin**

**datatype** *stern* =  
  *Sconst name* |  
  *Svar name* |  
  *Sabs (clauses: (term  $\times$  stern) list)* |  
  *Sapp stern stern (infixl \$<sub>s</sub> 70)*

**datatype-compat** *term*

**derive** *linorder term*

**abbreviation** *Sabs-single* ( $\Lambda_s$  -. - [0, 50] 50) **where**  
*Sabs-single*  $x$  *rhs*  $\equiv$  *Sabs* [(*Free*  $x$ , *rhs*)]

**type-synonym** *sclauses* = (*term*  $\times$  *term*) *list*

**lemma** *term-induct*[*case-names Sconst Svar Sabs Sapp*]:

**assumes**  $\bigwedge x. P$  (*Sconst*  $x$ )

**assumes**  $\bigwedge x. P$  (*Svar*  $x$ )

**assumes**  $\bigwedge cs. (\bigwedge pat\ t. (pat, t) \in set\ cs \implies P\ t) \implies P$  (*Sabs*  $cs$ )

**assumes**  $\bigwedge t\ u. P\ t \implies P\ u \implies P$  ( $t\ \$_s\ u$ )

**shows**  $P\ t$

*<proof>*

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

**definition** *app-term where*

*app-term*  $t\ u = t\ \$_s\ u$

**fun** *unapp-term where*

*unapp-term* ( $t\ \$_s\ u$ ) = *Some* ( $t, u$ ) |

*unapp-term* - = *None*

**definition** *const-term where*

*const-term* = *Sconst*

**fun** *unconst-term where*

*unconst-term* (*Sconst* *name*) = *Some* *name* |

*unconst-term* - = *None*

**fun** *unfree-term where*

*unfree-term* (*Svar* *name*) = *Some* *name* |

*unfree-term* - = *None*

**definition** *free-term where*

*free-term* = *Svar*

**fun** *frees-term where*

*frees-term* (*Svar* *name*) = {*|name|*}

*frees-term* (*Sconst* -) = {*|*}

*frees-term* (*Sabs*  $cs$ ) = *ffUnion* (*fset-of-list* (*map* ( $\lambda(pat, rhs). frees-term\ rhs -$

*frees* *pat*)  $cs$ )) |

*frees-term* ( $t\ \$_s\ u$ ) = *frees-term*  $t$   $\cup$  *frees-term*  $u$

**fun** *subst-term where*

```

subst-term (Svar s) env = (case fmllookup env s of Some t => t | None => Svar s)
|
subst-term (t1 $s t2) env = subst-term t1 env $s subst-term t2 env |
subst-term (Sabs cs) env = Sabs (map (λ(pat, rhs). (pat, subst-term rhs (fmdrop-fset
(frees pat) env))) cs) |
subst-term t env = t

```

```

fun consts-term :: term => name fset where
consts-term (Svar -) = {||} |
consts-term (Sconst name) = {|name|} |
consts-term (Sabs cs) = ffUnion (fset-of-list (map (λ(-, rhs). consts-term rhs)
cs)) |
consts-term (t $s u) = consts-term t |∪| consts-term u

```

```

instance
⟨proof⟩

```

```

end

```

```

instantiation term :: term begin

```

```

definition abs-pred-term :: (term => bool) => term => 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-termI[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 (t1 $s t2) S <=> closed-except t1 S ∧ closed-except t2 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** *term :: strong-term begin*

**fun** *wellformed-term :: term  $\Rightarrow$  bool where*  
*wellformed-term (t<sub>1</sub> \$<sub>s</sub> t<sub>2</sub>)  $\longleftrightarrow$  wellformed-term t<sub>1</sub>  $\wedge$  wellformed-term t<sub>2</sub> |*  
*wellformed-term (Sabs cs)  $\longleftrightarrow$  list-all ( $\lambda(pat, t). linear\ pat \wedge wellformed-term$*   
*t) cs  $\wedge$  distinct (map fst cs)  $\wedge$  cs  $\neq$  [] |*  
*wellformed-term -  $\longleftrightarrow$  True*

**primrec** *all-frees-term :: term  $\Rightarrow$  name fset where*  
*all-frees-term (Svar x) = {|x|} |*  
*all-frees-term (t<sub>1</sub> \$<sub>s</sub> t<sub>2</sub>) = all-frees-term t<sub>1</sub>  $\cup$  all-frees-term t<sub>2</sub> |*  
*all-frees-term (Sabs cs) = ffUnion (fset-of-list (map ( $\lambda(P, T). P \cup T$ ) (map*  
*(map-prod frees all-frees-term) cs))) |*  
*all-frees-term (Sconst -) = {|}|*

**instance** *<proof>*

**end**

**lemma** *match-sabs[simp]:  $\neg is-free\ t \implies match\ t\ (Sabs\ cs) = None$*   
*<proof>*

**context** *pre-constants begin*

**lemma** *welldefined-sabs: welldefined (Sabs cs)  $\longleftrightarrow$  list-all ( $\lambda(-, t). welldefined\ t$ )*  
*cs*  
*<proof>*

**lemma** *shadows-consts-term-simps[simp]:*  
*shadows-consts (t<sub>1</sub> \$<sub>s</sub> t<sub>2</sub>)  $\longleftrightarrow$  shadows-consts t<sub>1</sub>  $\vee$  shadows-consts t<sub>2</sub>*  
*shadows-consts (Svar name)  $\longleftrightarrow$  name  $\in$  all-consts*  
*shadows-consts (Sabs cs)  $\longleftrightarrow$  list-ex ( $\lambda(pat, t). \neg fdisjnt\ all-consts\ (frees\ pat) \vee$*   
*shadows-consts t) cs*  
*shadows-consts (Sconst name)  $\longleftrightarrow$  False*  
*<proof>*

**lemma** *subst-shadows:*

**assumes**  $\neg shadows-consts\ (t::term)\ not-shadows-consts-env\ \Gamma$   
**shows**  $\neg shadows-consts\ (subst\ t\ \Gamma)$   
*<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 \$<sub>p</sub> 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 \$<sub>s</sub> u) = sterm-to-pterm t \$<sub>p</sub> 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**  $\bigwedge cs. (\bigwedge pat t. (pat, t) \in cs \implies P t) \implies P (Pabs cs)$

**assumes**  $\bigwedge t u. P t \implies P u \implies P (t $<sub>p</sub> u)$

**shows**  $P t$

*<proof>*

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

**definition** *app-pterm* **where**

*app-pterm t u = t \$<sub>p</sub> u*

**fun** *unapp-pterm* **where**

*unapp-pterm (t \$<sub>p</sub> u) = Some (t, u)* |

*unapp-pterm - = None*

**definition** *const-pterm* **where**

$const\text{-}pterm = Pconst$

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

**definition**  $free\text{-}pterm$  **where**  
 $free\text{-}pterm = Pvar$

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

**function** (*sequential*)  $subst\text{-}pterm$  **where**  
 $subst\text{-}pterm (Pvar s) env = (case fmlookup env s of Some t \Rightarrow t \mid None \Rightarrow Pvar$   
 $s) \mid$   
 $subst\text{-}pterm (t_1 \$p t_2) env = subst\text{-}pterm t_1 env \$p subst\text{-}pterm t_2 env \mid$   
 $subst\text{-}pterm (Pabs cs) env = Pabs ((\lambda(pat, rhs). (pat, subst\text{-}pterm rhs (fmdrop\text{-}fset$   
 $(frees pat) env))) \mid \uparrow cs) \mid$   
 $subst\text{-}pterm t - = t$   
 $\langle proof \rangle$

**termination**  
 $\langle proof \rangle$   
**including**  $fset.liftng \langle proof \rangle$

**primrec**  $consts\text{-}pterm :: pterm \Rightarrow name fset$  **where**  
 $consts\text{-}pterm (Pconst x) = \{|x|\} \mid$   
 $consts\text{-}pterm (t_1 \$p t_2) = consts\text{-}pterm t_1 \mid \cup \mid consts\text{-}pterm t_2 \mid$   
 $consts\text{-}pterm (Pabs cs) = \text{ffUnion} (snd \mid \uparrow map\text{-}prod id consts\text{-}pterm \mid \uparrow cs) \mid$   
 $consts\text{-}pterm (Pvar -) = \{|\}\}$

**primrec**  $frees\text{-}pterm :: pterm \Rightarrow name fset$  **where**  
 $frees\text{-}pterm (Pvar x) = \{|x|\} \mid$   
 $frees\text{-}pterm (t_1 \$p t_2) = frees\text{-}pterm t_1 \mid \cup \mid frees\text{-}pterm t_2 \mid$   
 $frees\text{-}pterm (Pabs cs) = \text{ffUnion} ((\lambda(pv, tv). tv - frees pv) \mid \uparrow map\text{-}prod id frees\text{-}pterm$   
 $\mid \uparrow cs) \mid$   
 $frees\text{-}pterm (Pconst -) = \{|\}\}$

**instance**  
 $\langle proof \rangle$

**end**

**corollary**  $subst\text{-}pabs\text{-}id$ :

**assumes**  $\bigwedge pat rhs. (pat, rhs) \mid \in \mid cs \Longrightarrow subst rhs (fmdrop\text{-}fset (frees pat) env)$   
 $= rhs$

**shows**  $subst (Pabs cs) env = Pabs cs$   
 $\langle proof \rangle$

**corollary** *frees-pabs-alt-def*:

*frees* (*Pabs cs*) = *ffUnion* (( $\lambda(pat, rhs). \text{frees } rhs - \text{frees } pat$ ) | $\in$ | *cs*)  
*<proof>*

**lemma** *stern-to-pterm-frees[simp]*: *frees* (*stern-to-pterm t*) = *frees t*  
*<proof>*

**lemma** *stern-to-pterm-consts[simp]*: *consts* (*stern-to-pterm t*) = *consts t*  
*<proof>*

**lemma** *subst-stern-to-pterm*:

*subst* (*stern-to-pterm t*) (*fmap stern-to-pterm env*) = *stern-to-pterm* (*subst t env*)  
*<proof>*

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

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

**context** **begin**

**private lemma** *abs-pred-trivI0*: *P t*  $\Longrightarrow$  *abs-pred P (t::pterm)*  
*<proof>*

**instance** *<proof>*

**end**

**end**

**lemma** *no-abs-abs[simp]*:  $\neg$  *no-abs* (*Pabs cs*)  
*<proof>*

**lemma** *stern-to-pterm*:

**assumes** *no-abs t*

**shows** *stern-to-pterm t* = *convert-term t*

*<proof>*

**abbreviation** *Pabs-single* ( $\Lambda_p \cdot - [0, 50] 50$ ) **where**

*Pabs-single x rhs*  $\equiv$  *Pabs* { $|$  (*Free x, rhs*)  $|$ }

**lemma** *closed-except-simps*:

*closed-except* (*Pvar x S*)  $\longleftrightarrow$   $x | \in | S$

*closed-except* ( $t_1 \ \$_p \ t_2$ ) *S*  $\longleftrightarrow$  *closed-except*  $t_1$  *S*  $\wedge$  *closed-except*  $t_2$  *S*

*closed-except* (*Pabs cs*) *S*  $\longleftrightarrow$  *fBall cs* ( $\lambda(pat, t). \text{closed-except } t (S | \cup | \text{frees } pat)$ )

*closed-except* (*Pconst name*) *S*  $\longleftrightarrow$  *True*

*<proof>*

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

**function** (*sequential*) *wellformed-pterm* :: *pterm*  $\Rightarrow$  *bool* **where**

*wellformed-pterm* ( $t_1 \ \$_p \ t_2$ )  $\longleftrightarrow$  *wellformed-pterm*  $t_1 \wedge$  *wellformed-pterm*  $t_2 \mid$

*wellformed-pterm* (*Pabs cs*)  $\longleftrightarrow$  *fBall cs* ( $\lambda(pat, t). \text{linear } pat \wedge \text{wellformed-pterm } t$ )  $\wedge$  *is-fmap cs*  $\wedge$  *pattern-compatibles cs*  $\wedge$   $cs \neq \{\mid\}$   $\mid$

*wellformed-pterm -*  $\longleftrightarrow$  *True*

*<proof>*

**termination**

*<proof>*

**including** *fset.lifting* *<proof>*

**primrec** *all-frees-pterm* :: *pterm*  $\Rightarrow$  *name fset* **where**

*all-frees-pterm* (*Pvar x*) =  $\{|x|\}$   $\mid$

*all-frees-pterm* ( $t_1 \ \$_p \ t_2$ ) = *all-frees-pterm*  $t_1 \ \mid \cup \mid$  *all-frees-pterm*  $t_2 \ \mid$

*all-frees-pterm* (*Pabs cs*) = *ffUnion* ( $(\lambda(P, T). P \ \mid \cup \mid T) \ \mid \uparrow \mid$  *map-prod frees all-frees-pterm*  $\mid \uparrow \mid$  *cs*)  $\mid$

*all-frees-pterm* (*Pconst -*) =  $\{\mid\}$

**instance**

*<proof>*

**end**

**lemma** *stern-to-pterm-all-frees[simp]*: *all-frees* (*stern-to-pterm t*) = *all-frees t*

*<proof>*

**instance** *pterm* :: *strong-term* *<proof>*

**lemma** *wellformed-PabsI*:

**assumes** *is-fmap cs pattern-compatibles cs cs*  $\neq \{\mid\}$

**assumes**  $\bigwedge pat \ t. (pat, t) \ \mid \in \mid \ cs \implies \text{linear } pat$

**assumes**  $\bigwedge pat \ t. (pat, t) \ \mid \in \mid \ cs \implies \text{wellformed } t$

**shows** *wellformed* (*Pabs cs*)

*<proof>*

**corollary** *subst-closed-pabs*:

**assumes** (*pat, rhs*)  $\mid \in \mid \ cs$  *closed* (*Pabs cs*)

**shows** *subst rhs* (*fmdrop-fset* (*frees pat*) *env*) = *rhs*

*<proof>*

**lemma** (**in constants**) *shadows-consts-pterm-simps[simp]*:

*shadows-consts* ( $t_1 \ \$_p \ t_2$ )  $\longleftrightarrow$  *shadows-consts*  $t_1 \ \vee$  *shadows-consts*  $t_2$

*shadows-consts* (*Pvar name*)  $\longleftrightarrow$  *name*  $\mid \in \mid$  *all-consts*

*shadows-consts* (*Pabs cs*)  $\longleftrightarrow$  *fBex cs* ( $\lambda(pat, t). \text{shadows-consts } pat \ \vee \ \text{shadows-consts } t$ )

*shadows-consts* (*Pconst name*)  $\longleftrightarrow$  *False*  
 <proof>

**end**

## 1.8 Irreducible terms (values)

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

### 1.9 Viewing *stern* as values

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

**context** *constructors* **begin**

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

**lemma** *value-distinct*:  
*Sabs cs*  $\neq$  *name* \$\$ *ts* (**is** ?*P*)  
*name* \$\$ *ts*  $\neq$  *Sabs cs* (**is** ?*Q*)  
 <proof>

**abbreviation** *value-env* :: (*name, stern*) *fmap*  $\Rightarrow$  *bool* **where**  
*value-env*  $\equiv$  *fmpred* ( $\lambda$ -. *is-value*)

**lemma** *svar-value*[*simp*]:  $\neg$  *is-value* (*Svar name*)  
 <proof>

**lemma** *value-cases*:  
**obtains** (*comb*) *name vs* **where** *list-all is-value vs t = name* \$\$ *vs name*  $\in$  *C*  
 | (*abs*) *cs* **where** *t = Sabs cs*  
 | (*nonvalue*)  $\neg$  *is-value t*  
 <proof>

**end**

**fun** *smatch'* :: *pat*  $\Rightarrow$  *stern*  $\Rightarrow$  (*name, stern*) *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*)  $\Rightarrow$   
 (*if name = name'*  $\wedge$  *length ps = length vs* *then*  
*map-option* (*foldl* (*++<sub>f</sub>*) *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 (++) 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  $fmpred$  ( $\lambda$ -.  $pred$ )  $env$  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**  $fmpred\ (\lambda$ -.  $pred)\ env$   
 $\langle proof \rangle$

**end**

**primrec** *value-to-term* ::  $value \Rightarrow term$  **where**

$value\text{-to-term}\ (Vconst\ name\ vs) = name\ \$\$ map\ value\text{-to-term}\ vs\ |$   
 $value\text{-to-term}\ (Vabs\ cs\ \Gamma) = Sabs\ (map\ (\lambda(pat, t). (pat, subst\ t\ (fmdrop\text{-fset}\ (frees\ pat)\ (fmmap\ value\text{-to-term}\ \Gamma))))\ cs)\ |$   
 $value\text{-to-term}\ (Vrecabs\ css\ name\ \Gamma) =$   
 $Sabs\ (map\ (\lambda(pat, t). (pat, subst\ t\ (fmdrop\text{-fset}\ (frees\ pat)\ (fmmap\ value\text{-to-term}\ \Gamma))))\ (the\ (fmlookup\ css\ name))))$

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

**locale** *pre-value-term-pred* =  $value\text{-pred} +$

**fixes**  $S$   
**assumes**  $value\text{-to-term}: pred\ v \Longrightarrow S\ (value\text{-to-term}\ v)$   
**begin**

**corollary** *value-to-term-env*:

**assumes**  $fmpred\ (\lambda$ -.  $pred)\ \Gamma$   
**shows**  $fmpred\ (\lambda$ -.  $S)\ (fmmap\ value\text{-to-term}\ \Gamma)$   
 $\langle proof \rangle$

**end**

**locale** *value-term-pred* =  $value\text{-pred} + S$ : *simple-syntactic-and S* **for**  $S +$

**assumes**  $const: \bigwedge name. Q\ name \Longrightarrow S\ (const\ name)$   
**assumes**  $abs: \bigwedge \Gamma\ cs.$   
 $(\bigwedge n\ v. fmlookup\ \Gamma\ n = Some\ v \Longrightarrow pred\ v \Longrightarrow S\ (value\text{-to-term}\ v)) \Longrightarrow$   
 $fmpred\ (\lambda$ -.  $pred)\ \Gamma \Longrightarrow$   
 $P\ \Gamma\ cs \Longrightarrow$   
 $S\ (Sabs\ (map\ (\lambda(pat, t). (pat, subst\ t\ (fmmap\ value\text{-to-term}\ (fmdrop\text{-fset}\ (frees\ pat)\ \Gamma))))\ cs))$   
**begin**

**sublocale** *pre-value-term-pred*

$\langle proof \rangle$

**end**

**global-interpretation** *vwellformed*:

*value-term-pred*

$\lambda$ -. *wellformed-clauses*

$\lambda$ -. *True*

$\lambda$ -. *True*

*wellformed*

**defines** *vwellformed* = *vwellformed.pred*

*<proof>*

**abbreviation** *wellformed-venv*  $\equiv$  *fmpred* ( $\lambda$ -. *vwellformed*)

**global-interpretation** *vclosed*:

*value-term-pred*

$\lambda\Gamma$  *cs*. *list-all* ( $\lambda(pat, t)$ . *closed-except* *t* (*fmdom*  $\Gamma$   $\cup$  *frees pat*)) *cs*

$\lambda$ -. *True*

$\lambda$ -. *True*

*closed*

**defines** *vclosed* = *vclosed.pred*

*<proof>*

**abbreviation** *closed-venv*  $\equiv$  *fmpred* ( $\lambda$ -. *vclosed*)

**context** *pre-constants* **begin**

**sublocale** *vwelldefined*:

*value-term-pred*

$\lambda$ - *cs*. *list-all* ( $\lambda(-, t)$ . *welldefined* *t*) *cs*

$\lambda$ *name*. *name*  $\in$  *C*

$\lambda$ *dom*. *dom*  $\subseteq$  *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-term-pred*

$\lambda$ - -. *True*

$\lambda$ *name*. *name*  $\in$  *C*

$\lambda$ -. *True*

*is-value*

**defines** *vconstructor-value* = *vconstructor-value.pred*

*<proof>*

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

**abbreviation** *vconstructor-value-env*  $\equiv$  *fmpred* ( $\lambda$ -. *vconstructor-value*)

**definition** *vconstructor-value-rs* :: *vrule list*  $\Rightarrow$  *bool* **where**

*vconstructor-value-rs* *rs*  $\longleftrightarrow$   
  *list-all* ( $\lambda$ (-, *rhs*). *vconstructor-value* *rhs*) *rs*  $\wedge$   
  *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*

$\lambda$ - *cs*. *list-all* ( $\lambda$ (*pat*, *t*). *fdisjnt all-consts* (*frees pat*)  $\wedge$   $\neg$  *shadows-consts t*) *cs*

$\lambda$ -. *True*

$\lambda$ -. *True*

$\lambda$ *t*.  $\neg$  *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*  $\equiv$  *fmpred* ( $\lambda$ - *s*. *not-shadows-vconsts s*)

**end**

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

**fun** *term-to-value* :: *sterm*  $\Rightarrow$  *value* **where**

*term-to-value t* =

  (*case strip-comb t of*

    (*Sconst name, args*)  $\Rightarrow$  *Vconstr name* (*map term-to-value args*)

  | (*Sabs cs, []*)  $\Rightarrow$  *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*  $\Rightarrow$  *value*  $\Rightarrow$  ((*name*, *value*) *fmap*  $\times$  *term*  $\times$  *sterm*)  
*option where*  
*vfind-match* [] - = *None* |  
*vfind-match* ((*pat*, *rhs*) # *cs*) *t* =  
 (*case vmatch* (*mk-pat pat*) *t of*  
 *Some env*  $\Rightarrow$  *Some* (*env*, *pat*, *rhs*)  
 | *None*  $\Rightarrow$  *vfind-match cs t*)

**lemma** *vfind-match-elim*:

**assumes** *vfind-match cs t = Some* (*env*, *pat*, *rhs*)  
**shows** (*pat*, *rhs*)  $\in$  *set cs vmatch* (*mk-pat pat*) *t = Some env*  
*<proof>*

**inductive** *veq-structure* :: *value*  $\Rightarrow$  *value*  $\Rightarrow$  *bool where*

*abs-abs*: *veq-structure* (*Vabs* - -) (*Vabs* - -) |  
*recabs-recabs*: *veq-structure* (*Vrecabs* - - -) (*Vrecabs* - - -) |  
*constr-constr*: *list-all2 veq-structure ts us*  $\Longrightarrow$  *veq-structure* (*Vconstr name ts*) (*Vconstr name us*)

**lemma** *veq-structure-simps*[*code*, *simp*]:

*veq-structure* (*Vabs cs*<sub>1</sub>  $\Gamma$ <sub>1</sub>) (*Vabs cs*<sub>2</sub>  $\Gamma$ <sub>2</sub>)  
*veq-structure* (*Vrecabs css*<sub>1</sub> *name*<sub>1</sub>  $\Gamma$ <sub>1</sub>) (*Vrecabs css*<sub>2</sub> *name*<sub>2</sub>  $\Gamma$ <sub>2</sub>)  
*veq-structure* (*Vconstr name*<sub>1</sub> *ts*) (*Vconstr name*<sub>2</sub> *us*)  $\longleftrightarrow$  *name*<sub>1</sub> = *name*<sub>2</sub>  $\wedge$   
*list-all2 veq-structure ts us*  
*<proof>*

**lemma** *veq-structure-refl*[*simp*]: *veq-structure t t*

*<proof>*

**global-interpretation** *vno-abs*: *value-pred*  $\lambda$ - -. *False*  $\lambda$ -. *True*  $\lambda$ -. *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*  $\Rightarrow$  *value*  $\Rightarrow$  (*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' us*  $\Rightarrow$

```

      (if name = name'  $\wedge$  length ps = length vs then
        map-option (foldl (++f) fmempty) (those (map2 vmatch' ps vs))
      else
        None)
  | -  $\Rightarrow$  None)

```

**lemma** *vmatch-vmatch'-eq*:  $vmatch\ p\ v = vmatch'\ p\ v$   
 $\langle proof \rangle$

**locale** *value-struct-rel* =  
**fixes**  $Q :: value \Rightarrow value \Rightarrow bool$   
**assumes** *Q-impl-struct*:  $Q\ t_1\ t_2 \Longrightarrow veq\ structure\ t_1\ t_2$   
**assumes** *Q-def[simp]*:  $Q\ (Vconstr\ name\ ts)\ (Vconstr\ name'\ us) \longleftrightarrow name = name' \wedge list\ all2\ Q\ ts\ us$   
**begin**

**lemma** *eq-left*:  $Q\ t\ u \Longrightarrow vno\ abs\ t \Longrightarrow t = u$   
 $\langle proof \rangle$

**lemma** *eq-right*:  $Q\ t\ u \Longrightarrow vno\ abs\ u \Longrightarrow t = u$   
 $\langle proof \rangle$

**context begin**

**private lemma** *vmatch'-rel*:  
**assumes**  $Q\ t_1\ t_2$   
**shows** *rel-option* (*fmrel*  $Q$ ) (*vmatch'*  $p\ t_1$ ) (*vmatch'*  $p\ t_2$ )  
 $\langle proof \rangle$

**lemma** *vmatch-rel*:  $Q\ t_1\ t_2 \Longrightarrow rel\ option\ (fmrel\ Q)\ (vmatch\ p\ t_1)\ (vmatch\ p\ t_2)$   
 $\langle proof \rangle$

**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*  $cs_2\ t_2$ )  
 $\langle proof \rangle$

**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**  
 $env-eq \equiv fmlrel (\lambda v t. t = value-to-stern v)$

**lemma** *env-eq-eq*:  
  **assumes** *env-eq venv senv*  
  **shows**  $senv = fmmmap value-to-stern venv$   
{proof}

**context** *constructors* **begin**

**context** **begin**

**private lemma** *vmatch-eq0*: *rel-option env-eq (vmatch p v) (smatch' p (value-to-stern 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-stern v))*  
{proof}

**end**

**end**

**abbreviation** *match-related* **where**  
 $match-related \equiv (\lambda(\Gamma_1, pat_1, rhs_1) (\Gamma_2, pat_2, rhs_2). rhs_1 = rhs_2 \wedge pat_1 = pat_2 \wedge env-eq \Gamma_1 \Gamma_2)$

**lemma** (**in** *constructors*) *find-match-eq*:  
  **assumes** *list-all (linear o fst) cs vconstructor-value v*  
  **shows** *rel-option match-related (vfind-match cs v) (find-match cs (value-to-stern v))*  
{proof}

**inductive** *erelated* :: *value*  $\Rightarrow$  *value*  $\Rightarrow$  *bool* ( $- \approx_e -$ ) **where**  
*constr*:  $list-all2\ erelated\ ts\ us \Longrightarrow Vconstr\ name\ ts \approx_e Vconstr\ name\ us \mid$   
*abs*:  $fmlrel-on-fset\ (ids\ (Sabs\ cs))\ erelated\ \Gamma_1\ \Gamma_2 \Longrightarrow Vabs\ cs\ \Gamma_1 \approx_e Vabs\ cs\ \Gamma_2 \mid$   
*rec-abs*:  
   $pred-fmap\ (\lambda cs. fmlrel-on-fset\ (ids\ (Sabs\ cs))\ erelated\ \Gamma_1\ \Gamma_2)\ css \Longrightarrow$   
     $Vrecabs\ css\ name\ \Gamma_1 \approx_e Vrecabs\ css\ name\ \Gamma_2$

**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-stern 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 ( $\lambda(p, e).$  cupcake-pmatch cenv p v0 [] = No-match  $\wedge$  distinct  
 (pat-bindings p [])) init  
**and** cupcake-pmatch cenv p v0 [] = Match env'  
 <proof>

**lemma** cupcake-pmatch-eq:

is-cupcake-pat pat  $\Longrightarrow$  pmatch-single envC s pat v0 env = cupcake-pmatch envC  
 pat v0 env  
 <proof>

**lemma** cupcake-match-result-eq:

cupcake-clauses pes  $\Longrightarrow$   
 match-result env s v pes err-v =  
 map-result ( $\lambda(e, -, env').$  (e, env')) id (cupcake-match-result (c env) v pes  
 err-v)  
 <proof>

**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)  
 <proof>

**lemma** cupcake-build-rec-preserve:

**assumes** is-cupcake-all-env cl-env is-cupcake-ns env list-all ( $\lambda(-, -, e).$  is-cupcake-exp  
 e) fs  
**shows** is-cupcake-ns (build-rec-env fs cl-env env)  
 <proof>

**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)  
 <proof>

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

**lemma** *cupcake-alist-to-ns-preserve*:  $list-all\ (is-cupcake-value\ \circ\ snd)\ env \implies is-cupcake-ns\ (alist-to-ns\ env)$   
 ⟨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-neg*:  
 $length\ vs \neq length\ ps \implies Matching.fold2(\lambda p\ v\ m.\ case\ m\ of$   
 $\quad Match\ env \Rightarrow cupcake-pmatch\ cenv\ p\ v\ env$   
 $\quad | m \Rightarrow m)\ Match-type-error\ ps\ vs\ m = Match-type-error$   
 ⟨proof⟩

**lemma** *cup-pmatch-list-nomatch*:  
 $length\ vs = length\ ps \implies Matching.fold2(\lambda p\ v\ m.\ case\ m\ of$   
 $\quad Match\ env \Rightarrow cupcake-pmatch\ cenv\ p\ v\ env$   
 $\quad | m \Rightarrow m)\ Match-type-error\ ps\ vs\ No-match = No-match$   
 ⟨proof⟩

**lemma** *cup-pmatch-list-tyerr*:  
 $length\ ps = length\ ps \implies Matching.fold2(\lambda p\ v\ m.\ case\ m\ of$   
 $\quad Match\ env \Rightarrow cupcake-pmatch\ cenv\ p\ v\ env$   
 $\quad | m \Rightarrow m)\ Match-type-error\ ps\ vs\ Match-type-error = Match-type-error$   
 ⟨proof⟩ **lemma** *cupcake-pmatch-list-preserve*:  
**assumes**  $\bigwedge p\ v\ env.\ p \in set\ ps \wedge v \in set\ vs \longrightarrow list-all\ (is-cupcake-value\ \circ\ snd)\ env \longrightarrow if-match\ (list-all\ (is-cupcake-value\ \circ\ snd))\ (cupcake-pmatch\ cenv\ p\ v\ env)$   
 $list-all\ (is-cupcake-value\ \circ\ snd)\ env$   
**shows**  $if-match\ (list-all\ (\lambda a.\ is-cupcake-value\ (snd\ a)))\ (Matching.fold2$   
 $\quad (\lambda p\ v\ m.\ case\ m\ of$   
 $\quad\quad Match\ env \Rightarrow cupcake-pmatch\ cenv\ p\ v\ env$   
 $\quad\quad | m \Rightarrow m)$   
 $\quad Match-type-error\ ps\ vs\ (Match\ env))$   
 ⟨proof⟩ **lemma** *cupcake-pmatch-preserve0*:  
 $is-cupcake-pat\ pat \implies$   
 $is-cupcake-value\ v0 \implies$   
 $list-all\ (is-cupcake-value\ \circ\ snd)\ env \implies$   
 $cupcake-c-ns\ envC \implies$   
 $if-match\ (list-all\ (is-cupcake-value\ \circ\ snd))\ (cupcake-pmatch\ envC\ pat\ v0\ env)$   
 ⟨proof⟩

**lemma** *cupcake-pmatch-preserve*:

*is-cupcake-pat pat*  $\implies$   
*is-cupcake-value v0*  $\implies$   
*list-all (is-cupcake-value  $\circ$  snd) env*  $\implies$   
*cupcake-c-ns envC*  $\implies$   
*cupcake-pmatch envC pat v0 env = Match env'*  $\implies$   
*list-all (is-cupcake-value  $\circ$  snd) env'*

$\langle$ *proof* $\rangle$

**end**

**lemma** *cupcake-match-result-preserve*:

*cupcake-c-ns envC*  $\implies$   
*cupcake-clauses pes*  $\implies$   
*is-cupcake-value v*  $\implies$   
*if-rval* ( $\lambda(e, p, env').$  *is-cupcake-pat p*  $\wedge$  *is-cupcake-exp e*  $\wedge$  *list-all (is-cupcake-value*  
 *$\circ$  snd) env')*  
*(cupcake-match-result envC v pes err-v)*

$\langle$ *proof* $\rangle$

**lemma** *static-cenv-lookup*:

**assumes** *cupcake-nsLookup static-cenv i = Some (len, b)*  
**obtains** *name where b = TypeId (Short name)*

$\langle$ *proof* $\rangle$

**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*

$\langle$ *proof* $\rangle$

**lemma** *cupcake-nsLookup-preserve*:

**assumes** *is-cupcake-ns ns nsLookup ns n = Some v0*  
**shows** *is-cupcake-value v0*

$\langle$ *proof* $\rangle$

**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  $\circ$  snd) env' is-cupcake-exp e is-cupcake-pat p*

$\langle$ *proof* $\rangle$

**end**

**fun** *list-all2-shortcircuit where*

*list-all2-shortcircuit P (x # xs) (y # ys)  $\longleftrightarrow$  (case y of Rval -  $\implies$  P x y  $\wedge$*   
*list-all2-shortcircuit P xs ys | Rerr -  $\implies$  P x y) |*

*list-all2-shortcircuit*  $P \ [] \ [] \longleftrightarrow \text{True}$  |  
*list-all2-shortcircuit*  $P \ - \ - \longleftrightarrow \text{False}$

**lemma** *list-all2-shortcircuit-induct*[consumes 1, case-names nil cons-val cons-err]:  
**assumes** *list-all2-shortcircuit*  $P \ xs \ ys$   
**assumes**  $R \ [] \ []$   
**assumes**  $\bigwedge x \ xs \ y \ ys. P \ x \ (Rval \ y) \Longrightarrow \text{list-all2-shortcircuit } P \ xs \ ys \Longrightarrow R \ xs \ ys$   
 $\Longrightarrow R \ (x \ \# \ xs) \ (Rval \ y \ \# \ ys)$   
**assumes**  $\bigwedge x \ xs \ y \ ys. P \ x \ (Rerr \ y) \Longrightarrow R \ (x \ \# \ xs) \ (Rerr \ y \ \# \ ys)$   
**shows**  $R \ xs \ ys$   
<proof>

**lemma** *list-all2-shortcircuit-mono*[mono]:  
**assumes**  $R \leq Q$   
**shows** *list-all2-shortcircuit*  $R \leq \text{list-all2-shortcircuit } Q$   
<proof>

**lemma** *list-all2-shortcircuit-weaken*: *list-all2-shortcircuit*  $P \ xs \ ys \Longrightarrow (\bigwedge xs \ ys. P \ xs \ ys \Longrightarrow Q \ xs \ ys) \Longrightarrow \text{list-all2-shortcircuit } Q \ xs \ ys$   
<proof>

**lemma** *list-all2-shortcircuit-rval*[simp]:  
*list-all2-shortcircuit*  $P \ xs \ (\text{map } Rval \ ys) \longleftrightarrow \text{list-all2} \ (\lambda x \ y. P \ x \ (Rval \ y)) \ xs \ ys$   
**(is ?lhs  $\longleftrightarrow$  ?rhs)**  
<proof>

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

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

*app3:*  
 $list\text{-}all2\text{-}shortcircuit (cupcake\text{-}evaluate\text{-}single\ env) (rev\ es)\ rs \implies$   
 $sequence\text{-}result\ rs = Rval\ vs \implies$   
 $do\text{-}opapp (rev\ vs) = None \implies$   
 $cupcake\text{-}evaluate\text{-}single\ env\ (App\ Opapp\ es)\ (Rerr\ (Rabort\ Rtype\text{-}error)) \mid$

*app6:*  
 $list\text{-}all2\text{-}shortcircuit (cupcake\text{-}evaluate\text{-}single\ env) (rev\ es)\ rs \implies$   
 $sequence\text{-}result\ rs = Rerr\ err \implies$   
 $cupcake\text{-}evaluate\text{-}single\ env\ (App\ op0\ es)\ (Rerr\ err) \mid$

*mat1:*  
 $cupcake\text{-}evaluate\text{-}single\ env\ e\ (Rval\ v0) \implies$   
 $cupcake\text{-}match\text{-}result\ (c\ env)\ v0\ pes\ Bindv = Rval\ (e',\ -, env') \implies$   
 $cupcake\text{-}evaluate\text{-}single\ (env\ (\mid\ sem\text{-}env.v := nsAppend\ (alist\text{-}to\text{-}ns\ env'))\ (sem\text{-}env.v\ env)\ \mid))\ e'\ bv \implies$   
 $cupcake\text{-}evaluate\text{-}single\ env\ (Mat\ e\ pes)\ bv \mid$

*mat1error:*  
 $cupcake\text{-}evaluate\text{-}single\ env\ e\ (Rval\ v0) \implies$   
 $cupcake\text{-}match\text{-}result\ (c\ env)\ v0\ pes\ Bindv = Rerr\ err \implies$   
 $cupcake\text{-}evaluate\text{-}single\ env\ (Mat\ e\ pes)\ (Rerr\ err) \mid$

*mat2:*  
 $cupcake\text{-}evaluate\text{-}single\ env\ e\ (Rerr\ err) \implies$   
 $cupcake\text{-}evaluate\text{-}single\ env\ (Mat\ e\ pes)\ (Rerr\ err)$

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

**context** **begin**

**private lemma** *cupcake-list-preserve0:*

$list\text{-}all2\text{-}shortcircuit$   
 $(\lambda e\ r.\ cupcake\text{-}evaluate\text{-}single\ env\ e\ r \wedge (is\text{-}cupcake\text{-}all\text{-}env\ env \implies is\text{-}cupcake\text{-}exp\ e \implies if\text{-}rval\ is\text{-}cupcake\text{-}value\ r))\ es\ rs \implies$   
 $is\text{-}cupcake\text{-}all\text{-}env\ env \implies list\text{-}all\ is\text{-}cupcake\text{-}exp\ es \implies sequence\text{-}result\ rs = Rval\ vs \implies list\text{-}all\ is\text{-}cupcake\text{-}value\ vs$   
 $\langle proof \rangle$  **lemma** *cupcake-single-preserve0:*  
 $cupcake\text{-}evaluate\text{-}single\ env\ e\ res \implies is\text{-}cupcake\text{-}all\text{-}env\ env \implies is\text{-}cupcake\text{-}exp\ e \implies if\text{-}rval\ is\text{-}cupcake\text{-}value\ res$   
 $\langle proof \rangle$

**lemma** *cupcake-single-preserve:*

$cupcake\text{-}evaluate\text{-}single\ env\ e\ (Rval\ res) \implies is\text{-}cupcake\text{-}all\text{-}env\ env \implies is\text{-}cupcake\text{-}exp\ e \implies is\text{-}cupcake\text{-}value\ res$   
 $\langle proof \rangle$

**lemma** *cupcake-list-preserve*:

*list-all2-shortcircuit (cupcake-evaluate-single env) es rs*  $\implies$   
*is-cupcake-all-env env*  $\implies$  *list-all is-cupcake-exp es*  $\implies$  *sequence-result rs = Rval*  
*vs*  $\implies$  *list-all is-cupcake-value vs*

*<proof>* **lemma** *cupcake-list-correct-rval*:

**assumes** *list-all2-shortcircuit*

( $\lambda e r.$

*cupcake-evaluate-single env e r*  $\wedge$

(*is-cupcake-all-env env*  $\longrightarrow$  *is-cupcake-exp e*  $\longrightarrow$  ( $\forall (s::'a \text{ state}). \exists s'. \text{evaluate}$   
*env s e (s', r)*)))

*es rs is-cupcake-all-env env list-all is-cupcake-exp es sequence-result rs = Rval*  
*vs*

**shows**  $\exists s'. \text{evaluate-list (evaluate env) (s::'a state) es (s',Rval vs)}$

*<proof>* **lemma** *cupcake-list-correct-rerr*:

**assumes** *list-all2-shortcircuit*

( $\lambda e r.$

*cupcake-evaluate-single env e r*  $\wedge$

(*is-cupcake-all-env env*  $\longrightarrow$  *is-cupcake-exp e*  $\longrightarrow$  ( $\forall (s::'a \text{ state}). \exists s'. \text{evaluate}$   
*env s e (s', r)*)))

*es rs is-cupcake-all-env env list-all is-cupcake-exp es sequence-result rs = Rerr*  
*err*

**shows**  $\exists s'. \text{evaluate-list (evaluate env) (s::'a state) es (s',Rerr err)}$

*<proof>* **lemma** *cupcake-list-correct0*:

**assumes** *list-all2-shortcircuit*

( $\lambda e r.$

*cupcake-evaluate-single env e r*  $\wedge$

(*is-cupcake-all-env env*  $\longrightarrow$  *is-cupcake-exp e*  $\longrightarrow$  ( $\forall (s::'a \text{ state}). \exists s'. \text{evaluate}$   
*env s e (s', r)*)))

*es rs is-cupcake-all-env env list-all is-cupcake-exp es*

**shows**  $\exists s'. \text{evaluate-list (evaluate env) (s::'a state) es (s',sequence-result rs)}$

*<proof>*

**lemma** *cupcake-single-correct*:

**assumes** *cupcake-evaluate-single env e res is-cupcake-all-env env is-cupcake-exp*  
*e*

**shows**  $\exists s'. \text{Big-Step-Unclocked-Single.evaluate env s e (s',res)}$

*<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**  $\exists s'. \text{evaluate-list (evaluate env) (s::'a state) es (s',sequence-result rs)}$

*<proof>* **lemma** *cupcake-list-complete0*:

*evaluate-list*

( $\lambda s e r. \text{evaluate env s e r} \wedge (\text{is-cupcake-all-env env} \longrightarrow \text{is-cupcake-exp e} \longrightarrow$   
*cupcake-evaluate-single env e (snd r)*)) *s1 es res*  $\implies$

*is-cupcake-all-env env*  $\implies$  *list-all is-cupcake-exp es*  $\implies$   $\exists rs. \text{list-all2-shortcircuit}$   
*(cupcake-evaluate-single env) es rs*  $\wedge$  *sequence-result rs = (snd res)*

*<proof>* **lemma** *cupcake-single-complete0*:



$evaluate\ env\ s\ e\ res \implies is-cupcake-all-env\ env \implies is-cupcake-exp\ e \implies cupcake-evaluate-single\ env\ e\ (snd\ res)$

$\langle proof \rangle$

**lemma** *cupcake-single-complete*:

$evaluate\ env\ s\ e\ (s',\ res) \implies is-cupcake-all-env\ env \implies is-cupcake-exp\ e \implies cupcake-evaluate-single\ env\ e\ res$

$\langle proof \rangle$

**lemma** *cupcake-list-complete*:

$evaluate-list\ (evaluate\ env)\ s1\ es\ res \implies$

$is-cupcake-all-env\ env \implies list-all\ is-cupcake-exp\ es \implies \exists rs.\ list-all2-shortcircuit\ (cupcake-evaluate-single\ env)\ es\ rs \wedge sequence-result\ rs = (snd\ res)$

$\langle proof \rangle$  **lemma** *cupcake-list-state-preserve0*:

**assumes**  $evaluate-list\ (\lambda s\ e\ res.\ Big-Step-Unclocked-Single.evaluate\ env\ s\ e\ res \wedge (is-cupcake-all-env\ env \longrightarrow is-cupcake-exp\ e \longrightarrow s = fst\ res))\ s\ es\ res$

$list-all\ is-cupcake-exp\ es\ is-cupcake-all-env\ env$

**shows**  $s = (fst\ res)$

$\langle proof \rangle$

**lemma** *cupcake-state-preserve*:

**assumes**  $Big-Step-Unclocked-Single.evaluate\ env\ s\ e\ res\ is-cupcake-all-env\ env\ is-cupcake-exp\ e$

**shows**  $s = (fst\ res)$

$\langle proof \rangle$

**corollary** *cupcake-single-correct-strong*:

**assumes**  $cupcake-evaluate-single\ env\ e\ res\ is-cupcake-all-env\ env\ is-cupcake-exp\ e$

**shows**  $Big-Step-Unclocked-Single.evaluate\ env\ s\ e\ (s,\ res)$

$\langle proof \rangle$

**corollary** *cupcake-single-complete-weak*:

$evaluate\ env\ s\ e\ (s,\ res) \implies is-cupcake-all-env\ env \implies is-cupcake-exp\ e \implies cupcake-evaluate-single\ env\ e\ res$

$\langle proof \rangle$

**end end**

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

**end**

## Chapter 3

# Term rewriting

```
theory Doc-Rewriting
imports Main
begin

end
theory General-Rewriting
imports Terms-Extras
begin

locale rewriting =
  fixes  $R :: 'a::term \Rightarrow 'a \Rightarrow bool$ 
  assumes  $R\text{-fun}: R\ t\ t' \Longrightarrow R\ (app\ t\ u)\ (app\ t'\ u)$ 
  assumes  $R\text{-arg}: R\ u\ u' \Longrightarrow R\ (app\ t\ u)\ (app\ t\ u')$ 
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  $list\text{-all}2\ R^{**}\ ts\ us\ R^{**}\ t\ u$ 
  shows  $R^{**}\ (list\text{-comb}\ t\ ts)\ (list\text{-comb}\ u\ us)$ 
  <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']β |
  fun: rs ⊢ t → t' ⇒ rs ⊢ t $ u → t' $ u |
  arg: rs ⊢ u → u' ⇒ rs ⊢ t $ u → t $ u'
```

```
global-interpretation rewrite: rewriting rewrite rs for rs
⟨proof⟩
```

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

```
lemma rewrite-beta-alt: t [t']β = u ⇒ wellformed t' ⇒ rs ⊢ (Λ t $ t') → u
⟨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:  $fBall\ rs\ (\lambda(lhs, -). \neg\ shadows-consts\ lhs)$ 
assumes welldefined-rs:  $fBall\ rs\ (\lambda(-, rhs).\ welldefined\ rhs)$ 
begin

lemma rewrite-wellformed:
  assumes  $rs \vdash t \longrightarrow t'$  wellformed t
  shows wellformed t'
  <proof>

lemma rewrite-rt-wellformed:  $rs \vdash t \longrightarrow^* t' \implies wellformed\ t \implies wellformed\ t'$ 
  <proof>

lemma rewrite-closed:  $rs \vdash t \longrightarrow t' \implies closed\ t \implies closed\ t'$ 
  <proof>

lemma rewrite-rt-closed:  $rs \vdash t \longrightarrow^* t' \implies closed\ t \implies closed\ t'$ 
  <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  $\times$  nterm

```

```

abbreviation nrule :: nrule  $\Rightarrow$  bool where
nrule  $\equiv$  basic-rule

```

```

fun (in constants) not-shadowing :: nrule  $\Rightarrow$  bool where
not-shadowing (lhs, rhs)  $\longleftrightarrow \neg\ shadows-consts\ lhs \wedge \neg\ shadows-consts\ rhs$ 

```

```

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

```

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

### 3.2.2 Matching and rewriting

**inductive** *nrewrite* :: *nrule fset ⇒ nterm ⇒ nterm ⇒ bool (-/ ⊢<sub>n</sub>/ - →/ -*  
*[50,0,50] 50)* **for** *rs* **where**  
*step*: *r |∈| rs ⇒ r ⊢ t → u ⇒ rs ⊢<sub>n</sub> t → u |*  
*beta*: *rs ⊢<sub>n</sub> ((Λ<sub>n</sub> x. t) \$<sub>n</sub> t') → subst t (fmap-of-list [(x, t')]) |*  
*fun*: *rs ⊢<sub>n</sub> t → t' ⇒ rs ⊢<sub>n</sub> t \$<sub>n</sub> u → t' \$<sub>n</sub> u |*  
*arg*: *rs ⊢<sub>n</sub> u → u' ⇒ rs ⊢<sub>n</sub> t \$<sub>n</sub> u → t \$<sub>n</sub> u'*

**global-interpretation** *nrewrite*: *rewriting nrewrite rs for rs*  
*<proof>*

**abbreviation** *nrewrite-rt* :: *nrule fset ⇒ nterm ⇒ nterm ⇒ bool (-/ ⊢<sub>n</sub>/ - →\*/*  
*- [50,0,50] 50)* **where**  
*nrewrite-rt rs ≡ (nrewrite rs)\*\**

**lemma** (**in** *nrules*) *nrewrite-closed*:  
**assumes** *rs ⊢<sub>n</sub> t → t' closed t*  
**shows** *closed t'*  
*<proof>*

**corollary** (**in** *nrules*) *nrewrite-rt-closed*:  
**assumes** *rs ⊢<sub>n</sub> t →\* 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**  $\exists T. \text{all-frees } (\text{fst } (\text{run-state } (\text{term-to-nterm } \Gamma \ t) \ x)) \mid \subseteq \mid \text{fset-of-list } \Gamma \mid \cup \mid$   
*frees t |∪| T ∧ fBall T (λy. 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 |∪| 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* | $\cup$ |  $S$ ))

**lemma** *translate-rule-alt-def*:

*translate-rule*  $S$  = ( $\lambda$ (*lhs*, *rhs*). (*lhs*, *fresh-frun* (*term-to-nterm* [] *rhs*) (*frees lhs* | $\cup$ |  $S$ )))  
 <proof>

**definition** *compile' where*

*compile'*  $C$ -info *rs* = *translate-rule* (*pre-constants.all-consts*  $C$ -info (*heads-of rs*) | $\uparrow$ | *rs*)

**context** *rules begin*

**definition** *compile :: nrule fset where*

*compile* = *translate-rule all-consts* | $\uparrow$ | *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*  $\vdash_n u \longrightarrow u'$  *closed u*

**shows** *rs*  $\vdash$  *nterm-to-term'*  $u \longrightarrow$  *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*  $\vdash t \longrightarrow t'$  *closed t wellformed t*

**obtains**  $u'$  **where** *compile*  $\vdash_n$  *fst* (*run-state* (*term-to-nterm* []  $t$ )  $s$ )  $\longrightarrow u'$   $u' \approx_\alpha$  *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*  $\times$  *nterm*) *fset*

**type-synonym** *crule-set* = (*name*  $\times$  *crules*) *fset*

**abbreviation** *arity-compatibles* :: (*term list*  $\times$  'a) *fset*  $\Rightarrow$  *bool* **where**

*arity-compatibles*  $\equiv$  *fpairwise* ( $\lambda$ (*pats*<sub>1</sub>, -) (*pats*<sub>2</sub>, -). *length* *pats*<sub>1</sub> = *length* *pats*<sub>2</sub>)

**lemma** *arity-compatible-length*:

**assumes** *arity-compatibles*  $rs$  (*pats*, *rhs*)  $|\in|$   $rs$

**shows** *length* *pats* = *arity*  $rs$

*<proof>*

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

**locale** *crules* = *pre-crules* +

**assumes** *fmap*: *is-fmap*  $rs$

**assumes** *nonempty*:  $rs \neq \{\}\}$

**assumes** *inner*:

*fBall*  $rs$  ( $\lambda$ (-, *crs*).

*arity-compatibles* *crs*  $\wedge$

*is-fmap* *crs*  $\wedge$

*patterns-compatibles* *crs*  $\wedge$

$crs \neq \{\}\}$   $\wedge$

*fBall* *crs* ( $\lambda$ (*pats*, *rhs*).

*linears* *pats*  $\wedge$

*pats*  $\neq \square$   $\wedge$

*fdisjnt* (*freess* *pats*) *all-consts*  $\wedge$

$\neg$  *shadows-consts* *rhs*  $\wedge$

*frees* *rhs*  $|\subseteq|$  *freess* *pats*  $\wedge$

*welldefined* *rhs*)

**lemma** (**in** *pre-crules*) *crulesI*:

**assumes**  $\bigwedge$  *name* *crs*. (*name*, *crs*)  $|\in|$   $rs \Longrightarrow$  *arity-compatibles* *crs*

**assumes**  $\bigwedge$  *name* *crs*. (*name*, *crs*)  $|\in|$   $rs \Longrightarrow$  *is-fmap* *crs*

**assumes**  $\bigwedge$  *name* *crs*. (*name*, *crs*)  $|\in|$   $rs \Longrightarrow$  *patterns-compatibles* *crs*

**assumes**  $\bigwedge$  *name* *crs*. (*name*, *crs*)  $|\in|$   $rs \Longrightarrow$   $crs \neq \{\}\}$

**assumes**  $\bigwedge$  *name* *crs* *pats* *rhs*. (*name*, *crs*)  $|\in|$   $rs \Longrightarrow$  (*pats*, *rhs*)  $|\in|$  *crs*  $\Longrightarrow$

```

linears pats
assumes  $\bigwedge name\ crs\ pats\ rhs. (name, crs) \in rs \implies (pats, rhs) \in crs \implies pats \neq []$ 
assumes  $\bigwedge name\ crs\ pats\ rhs. (name, crs) \in rs \implies (pats, rhs) \in crs \implies fdisjnt (freess\ pats)\ all-consts$ 
assumes  $\bigwedge name\ crs\ pats\ rhs. (name, crs) \in rs \implies (pats, rhs) \in crs \implies \neg shadows-consts\ rhs$ 
assumes  $\bigwedge name\ crs\ pats\ rhs. (name, crs) \in rs \implies (pats, rhs) \in crs \implies frees\ rhs \sqsubseteq freess\ pats$ 
assumes  $\bigwedge name\ crs\ pats\ rhs. (name, crs) \in rs \implies (pats, rhs) \in crs \implies welldefined\ rhs$ 
assumes is-fmap rs rs  $\neq \{\{\}\}$ 
shows crules C-info rs
<proof>

```

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

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

**lemma** *consts-of-heads*: *fst*  $\mid^{\dagger}$  *consts-of rs* = *heads-of rs*  
<proof>

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

**sublocale** *nrules*  $\subseteq$  *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*  $\times$  *pterm*) *fset*



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

**locale** *pre-irules* = constants *C-info fst* |<sup>4</sup> *rs* **for** *C-info* **and** *rs* :: *irule-set*

**locale** *irules* = *pre-irules* +

**assumes** *fmap*: *is-fmap* *rs*

**assumes** *nonempty*: *rs* ≠ {||}

**assumes** *inner*:

*fBall* *rs* ( $\lambda(-, \textit{irs})$ ).

*arity-compatibles* *irs* ∧

*is-fmap* *irs* ∧

*patterns-compatibles* *irs* ∧

*irs* ≠ {||} ∧

*fBall* *irs* ( $\lambda(\textit{pats}, \textit{rhs})$ ).

*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**  $\bigwedge \textit{name irs}. (\textit{name}, \textit{irs}) \in | \textit{rs} \implies \textit{arity-compatibles irs}$

**assumes**  $\bigwedge \textit{name irs}. (\textit{name}, \textit{irs}) \in | \textit{rs} \implies \textit{is-fmap irs}$

**assumes**  $\bigwedge \textit{name irs}. (\textit{name}, \textit{irs}) \in | \textit{rs} \implies \textit{patterns-compatibles irs}$

**assumes**  $\bigwedge \textit{name irs}. (\textit{name}, \textit{irs}) \in | \textit{rs} \implies \textit{irs} \neq \{\|\}$

**assumes**  $\bigwedge \textit{name irs pats rhs}. (\textit{name}, \textit{irs}) \in | \textit{rs} \implies (\textit{pats}, \textit{rhs}) \in | \textit{irs} \implies \textit{linears pats}$

**assumes**  $\bigwedge \textit{name irs pats rhs}. (\textit{name}, \textit{irs}) \in | \textit{rs} \implies (\textit{pats}, \textit{rhs}) \in | \textit{irs} \implies \textit{abs-ish pats rhs}$

**assumes**  $\bigwedge \textit{name irs pats rhs}. (\textit{name}, \textit{irs}) \in | \textit{rs} \implies (\textit{pats}, \textit{rhs}) \in | \textit{irs} \implies \textit{fdisjnt (freess pats) all-consts}$

**assumes**  $\bigwedge \textit{name irs pats rhs}. (\textit{name}, \textit{irs}) \in | \textit{rs} \implies (\textit{pats}, \textit{rhs}) \in | \textit{irs} \implies \textit{closed-except rhs (freess pats)}$

**assumes**  $\bigwedge \textit{name irs pats rhs}. (\textit{name}, \textit{irs}) \in | \textit{rs} \implies (\textit{pats}, \textit{rhs}) \in | \textit{irs} \implies \textit{wellformed rhs}$

**assumes**  $\bigwedge \textit{name irs pats rhs}. (\textit{name}, \textit{irs}) \in | \textit{rs} \implies (\textit{pats}, \textit{rhs}) \in | \textit{irs} \implies \neg \textit{shadows-consts rhs}$

**assumes**  $\bigwedge \textit{name irs pats rhs}. (\textit{name}, \textit{irs}) \in | \textit{rs} \implies (\textit{pats}, \textit{rhs}) \in | \textit{irs} \implies \textit{welldefined rhs}$

**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*  $\Rightarrow$  *pterm* **where**

*nterm-to-pterm* (*Nvar* *s*) = *Pvar* *s* |

*nterm-to-pterm* (*Nconst* *s*) = *Pconst* *s* |

*nterm-to-pterm* (*t*<sub>1</sub> \$<sub>n</sub> *t*<sub>2</sub>) = *nterm-to-pterm* *t*<sub>1</sub> \$<sub>p</sub> *nterm-to-pterm* *t*<sub>2</sub> |

*nterm-to-pterm* ( $\Lambda_n$  *x*. *t*) = ( $\Lambda_p$  *x*. *nterm-to-pterm* *t*)

**lemma** *nterm-to-pterm-inj*: *nterm-to-pterm* *x* = *nterm-to-pterm* *y*  $\implies$  *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*  $\Rightarrow$  *irules* **where**

*translate-crules* = *fimage* (*map-prod* *id* *nterm-to-pterm*)

**definition** *compile* :: *crule-set*  $\Rightarrow$  *irule-set* **where**

*compile* = *fimage* (*map-prod* *id* *translate-crules*)

**lemma** *compile-heads*: *fst* |<sup>!</sup> *compile* *rs* = *fst* |<sup>!</sup> *rs*  
<proof>

**lemma** (**in** *crules*) *compile-rules*: *irules* *C-info* (*compile* *rs*)  
<proof>

**sublocale** *crules*  $\subseteq$  *crules-as-irules*: *irules* *C-info* *compile* *rs*  
<proof>

## Transformation of *irule-set*

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

*transform-irules* *rs* = (  
  if *arity* *rs* = 0 then *rs*  
  else *map-prod id Pabs* |<sup>q</sup> *fgroup-by* ( $\lambda$ (*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*  $\Rightarrow$  *irule-set* **where**

*transform-irule-set* = *fimage* (*map-prod id transform-irules*)

**lemma** *transform-irule-set-heads*: *fst* |<sup>q</sup> *transform-irule-set* *rs* = *fst* |<sup>q</sup> *rs*

*<proof>*

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

*<proof>*

## Matching and rewriting

**definition** *irewrite-step* :: *name*  $\Rightarrow$  *term list*  $\Rightarrow$  *pterm*  $\Rightarrow$  *pterm*  $\Rightarrow$  *pterm option*  
**where**

*irewrite-step* *name pats rhs t* = *map-option* (*subst rhs*) (*match* (*name* \$\$ *pats*) *t*)

**abbreviation** *irewrite-step'* :: *name*  $\Rightarrow$  *term list*  $\Rightarrow$  *pterm*  $\Rightarrow$  *pterm*  $\Rightarrow$  *pterm*  $\Rightarrow$   
*bool* (-, -, -<sub>i</sub>/ -  $\rightarrow$ / - [50,0,50] 50) **where**

*name, pats, rhs*  $\vdash_i$  *t*  $\rightarrow$  *u*  $\equiv$  *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*  $\rightarrow$  *u*

*<proof>*

**inductive** *irewrite* :: *irule-set*  $\Rightarrow$  *pterm*  $\Rightarrow$  *pterm*  $\Rightarrow$  *bool* (-<sub>i</sub>/ -  $\rightarrow$ / - [50,0,50] 50) **for** *irs* **where**

*step*:  $\llbracket$  (*name*, *rs*)  $\in$  *irs*; (*pats*, *rhs*)  $\in$  *rs*; *name, pats, rhs*  $\vdash_i$  *t*  $\rightarrow$  *t'*  $\rrbracket \Longrightarrow$  *irs*  $\vdash_i$   
*t*  $\rightarrow$  *t'* |

*beta*:  $\llbracket$  *c*  $\in$  *cs*; *c*  $\vdash$  *t*  $\rightarrow$  *t'*  $\rrbracket \Longrightarrow$  *irs*  $\vdash_i$  *Pabs cs*  $\$_p$  *t*  $\rightarrow$  *t'* |

*fun*: *irs*  $\vdash_i$  *t*  $\rightarrow$  *t'*  $\Longrightarrow$  *irs*  $\vdash_i$  *t*  $\$_p$  *u*  $\rightarrow$  *t'*  $\$_p$  *u* |

*arg*: *irs*  $\vdash_i$  *u*  $\rightarrow$  *u'*  $\Longrightarrow$  *irs*  $\vdash_i$  *t*  $\$_p$  *u*  $\rightarrow$  *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*  $\implies$  *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*  $rs \vdash_n t \longrightarrow^* t' t' \approx_i u'$   
 $\langle proof \rangle$

### 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'*  
 $\langle proof \rangle$

### Correctness of transformation

**abbreviation** *irules-deferred-matches*  $::$  *pterm list*  $\Rightarrow$  *irules*  $\Rightarrow$  (*term*  $\times$  *pterm*)  
*fset where*  
*irules-deferred-matches args*  $\equiv$  *fselect*  
 $(\lambda(\text{pats}, \text{rhs}). \text{map-option } (\lambda \text{env}. (\text{last pats}, \text{subst rhs env})) (\text{matches } (\text{butlast pats}) \text{ args}))$

### context *irules begin*

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

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

**lemma** *prelated-refl*[*intro!*]:  $t \approx_p t$   
 $\langle proof \rangle$

**sublocale** *prelated*: *term-struct-rel prelated*  
 $\langle proof \rangle$

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

**corollary** *prelated-closed*:  $t \approx_p u \Longrightarrow closed\ t \longleftrightarrow closed\ u$   
 $\langle proof \rangle$

**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:  $t \approx_p u \implies \text{rel-option } \text{prelated.P-env } (\text{match } x \ t) \ (\text{match } x \ u)$  If  $t$  and  $u$  are related because of the *prelated.defer* rule, they have completely different shapes. Establishing  $\text{is-abs } t = \text{is-abs } u$  as a precondition would rule out this case, but at the same time be too restrictive.

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

**lemma** *prelated-subst*:

**assumes**  $t_1 \approx_p t_2$  *prelated.P-env env<sub>1</sub> env<sub>2</sub>*  
**shows**  $\text{subst } t_1 \ \text{env}_1 \approx_p \text{subst } t_2 \ \text{env}_2$   
*<proof>*

**lemma** *prelated-step*:

**assumes**  $\text{name}, \text{pats}, \text{rhs} \vdash_i u \rightarrow u' \ t \approx_p u$   
**obtains**  $t'$  **where**  $\text{name}, \text{pats}, \text{rhs} \vdash_i t \rightarrow t' \ t' \approx_p u'$   
*<proof>*

**lemma** *prelated-beta*: — same problem as *prelated.related-match*

**assumes**  $(\text{pat}, \text{rhs}_2) \vdash t_2 \rightarrow u_2 \ \text{rhs}_1 \approx_p \text{rhs}_2 \ t_1 \approx_p t_2$   
**obtains**  $u_1$  **where**  $(\text{pat}, \text{rhs}_1) \vdash t_1 \rightarrow u_1 \ u_1 \approx_p u_2$   
*<proof>*

**theorem** *transform-correct*:

**assumes**  $\text{transform-irule-set } rs \vdash_i u \longrightarrow u' \ t \approx_p u \ \text{closed } t$   
**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**  $rs \vdash_i t \longrightarrow t' \ \text{closed } t$   
**shows**  $\text{transform-irule-set } rs \vdash_i t \longrightarrow^* t'$   
*<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 (-/ ⊢p/ - →/ -
[50,0,50] 50) for rs where
step: (name, rhs) |∈| rs ⇒ rs ⊢p Pconst name → rhs |
beta: c |∈| cs ⇒ c ⊢ t → t' ⇒ rs ⊢p Pabs cs $p t → t' |
fun: rs ⊢p t → t' ⇒ rs ⊢p t $p u → t' $p u |
arg: rs ⊢p u → u' ⇒ rs ⊢p t $p u → t $p u'
```

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

```
abbreviation prewrite-rt :: prule fset ⇒ pterm ⇒ pterm ⇒ bool (-/ ⊢p/ - →*/
- [50,0,50] 50) where
prewrite-rt rs ≡ (prewrite rs)**
```

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

```
definition finished :: irule-set ⇒ bool where
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*:  $fst \mid^{\dagger} compile\ rs = fst \mid^{\dagger} rs$   
(*proof*)

### Correctness of translation

**lemma** *arity-zero-shape*:  
  **assumes** *arity-compatibles*  $rs$  *arity*  $rs = 0$  *is-fmap*  $rs$   $rs \neq \{\}\}$   
  **obtains**  $t$  **where**  $rs = \{ \mid (\ [], t) \mid \}$   
(*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*  $\times$  *stern*

**abbreviation** *closed-srules* :: *srule list*  $\Rightarrow$  *bool* **where**  
*closed-srules*  $\equiv$  *list-all* (*closed*  $\circ$  *snd*)

**primrec** *srule* :: *srule*  $\Rightarrow$  *bool* **where**  
*srule* ( $\_$ , *rhs*)  $\iff$  *wellformed* *rhs*  $\wedge$  *closed* *rhs*  $\wedge$  *is-abs* *rhs*

**lemma** *sruleI*[*intro!*]: *wellformed* *rhs*  $\implies$  *closed* *rhs*  $\implies$  *is-abs* *rhs*  $\implies$  *srule* (*name*,  
*rhs*)  
(*proof*)



```

locale srules = constants C-info fst |4 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 ( $\lambda(-, rhs). \neg \text{shadows-consts } rhs$ ) rs
  assumes swelldefined-rs: list-all ( $\lambda(-, rhs). \text{welldefined } rhs$ ) rs
begin

lemma map: is-map (set rs)
<proof>

lemma clausesE:
  assumes (name, rhs)  $\in$  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  $\neq$  name'  $\implies$  srewrite-step rs name rhs  $\implies$  srewrite-step
  ((name', rhs') # rs) name rhs

lemma srewrite-stepI0:
  assumes (name, rhs)  $\in$  set rs is-map (set rs)
  shows srewrite-step rs name rhs
<proof>

lemma (in srules) srewrite-stepI: (name, rhs)  $\in$  set rs  $\implies$  srewrite-step rs name
rhs
<proof>

```

**hide-fact** srewrite-stepI0

```

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

```

**code-pred** srewrite <proof>

```

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

```

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

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

### Translation from *pterm* to *stern*

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

**primrec** *pterm-to-stern* :: *pterm*  $\Rightarrow$  *stern* **where**  
*pterm-to-stern* (*Pconst* *name*) = *Sconst* *name* |  
*pterm-to-stern* (*Pvar* *name*) = *Svar* *name* |  
*pterm-to-stern* (*t*  $\$$ <sub>*p*</sub> *u*) = *pterm-to-stern* *t*  $\$$ <sub>*s*</sub> *pterm-to-stern* *u* |  
*pterm-to-stern* (*Pabs* *cs*) = *Sabs* (*ordered-fmap* (*map-prod* *id* *pterm-to-stern* |  
*cs*))

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

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

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

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

**corollary** *pterm-to-stern-frees*: *wellformed* *t*  $\Longrightarrow$  *frees* (*pterm-to-stern* *t*) = *frees* *t*  
⟨*proof*⟩

**corollary** *pterm-to-stern-closed*:  
  *closed-except* *t* *S*  $\Longrightarrow$  *wellformed* *t*  $\Longrightarrow$  *closed-except* (*pterm-to-stern* *t*) *S*  
⟨*proof*⟩

**corollary** *pterm-to-stern-consts*: *wellformed* *t*  $\Longrightarrow$  *consts* (*pterm-to-stern* *t*) = *consts* *t*  
⟨*proof*⟩

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

*wellformed*  $t \implies \text{shadows-consts } t \iff \text{shadows-consts } (\text{pterm-to-sterm } t)$   
 ⟨proof⟩

**definition** *compile* :: *prule fset*  $\Rightarrow$  *srule list* **where**  
*compile*  $rs = \text{ordered-fmap } (\text{map-prod id pterm-to-sterm } |^| \text{ } rs)$

## Correctness of translation

**context** *prules* **begin**

**lemma** *compile-heads*: *fst*  $|^| \text{ fset-of-list } (\text{compile } rs) = \text{fst } |^| \text{ } rs$   
 ⟨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* ( $\lambda p s. p = \text{sterm-to-pterm } s$ )  
 ⟨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*  $rs' \text{ name } rhs \implies rs' \neq []$   
 ⟨proof⟩ **lemma** *compile-consE*:  
**assumes** (*name*,  $rhs'$ )  $\# \text{ rest} = \text{compile } rs \text{ is-fmap } rs$   
**obtains**  $rhs$  **where**  $rhs' = \text{pterm-to-sterm } rhs \text{ (name, rhs) } | \in | \text{ } rs \text{ rest} = \text{compile}$   
 ( $rs - \{ | \text{ (name, rhs) } | \}$ )  
 ⟨proof⟩ **lemma** *compile-correct-step*:  
**assumes** *srewrite-step* (*compile*  $rs$ ) *name*  $rhs \text{ is-fmap } rs \text{ fBall } rs \text{ prule}$   
**shows** (*name*, *sterm-to-pterm*  $rhs$ )  $| \in | \text{ } rs$   
 ⟨proof⟩

**lemma** *compile-correct0*:  
**assumes** *compile*  $rs \vdash_s u \longrightarrow u' \text{ prules } C \text{ } rs$   
**shows**  $rs \vdash_p \text{sterm-to-pterm } u \longrightarrow \text{sterm-to-pterm } u'$

<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*:  
 $srelated'.P\text{-env}\ penv\ senv \implies srelated'.P\text{-env}\ penv\ senv' \implies senv = senv'$   
 <proof>

**lemma** *srelated-subst'*:  
 assumes  $srelated'.P\text{-env}\ penv\ senv\ wellformed\ t$   
 shows  $pterm\text{-to}\text{-sterm}\ (subst\ t\ penv) = subst\ (pterm\text{-to}\text{-sterm}\ t)\ senv$   
 <proof>

**lemma** *srelated-find-match*:  
 assumes  $find\text{-match}\ cs\ t = Some\ (penv,\ pat,\ rhs)\ srelated'.P\text{-env}\ penv\ senv$   
 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 *sterms*

```

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

```

**lemma** (**in constructors**) *seval-closed*:

```

assumes rs, \Gamma \vdash_s t \downarrow u closed-srules rs closed-env \Gamma closed-except t (fmdom \Gamma)
shows closed u
<proof>

```

**lemma** (**in srules**) *seval-wellformed*:

```

assumes rs, \Gamma \vdash_s t \downarrow u wellformed t wellformed-env \Gamma
shows wellformed u
<proof>

```

**lemma** (**in constants**) *seval-shadows*:

```

assumes rs, \Gamma \vdash_s t \downarrow u \neg shadows-consts t
assumes list-all (\lambda(-, rhs). \neg shadows-consts rhs) rs
assumes not-shadows-consts-env \Gamma
shows \neg shadows-consts u
<proof>

```

**lemma** (**in constructors**) *seval-list-comb-abs*:

```

assumes rs, \Gamma \vdash_s name $$ args \downarrow Sabs cs
shows name \in dom (map-of rs)
<proof>

```

**lemma** (**in constructors**) *is-value-eval-id*:

```

assumes is-value t closed t
shows rs, \Gamma \vdash_s t \downarrow t
<proof>

```

**lemma** (in *constructors*) *ssubst-eval*:  
**assumes**  $rs, \Gamma \vdash_s t \downarrow t' \Gamma' \subseteq_f \Gamma$  *closed-env*  $\Gamma$  *value-env*  $\Gamma$   
**shows**  $rs, \Gamma \vdash_s \text{subst } t \Gamma' \downarrow t'$   
 $\langle \text{proof} \rangle$

**lemma** (in *constructors*) *seval-agree-eq*:  
**assumes**  $rs, \Gamma \vdash_s t \downarrow u$  *fmrestrict-fset*  $S \Gamma = \text{fmrestrict-fset } S \Gamma'$  *closed-except*  $t$   
 $S$   
**assumes**  $S \subseteq_f \text{fmdom } \Gamma$  *closed-srules*  $rs$  *closed-env*  $\Gamma$   
**shows**  $rs, \Gamma' \vdash_s t \downarrow u$   
 $\langle \text{proof} \rangle$   
**including** *fmap.lifting*  
 $\langle \text{proof} \rangle$   
**including** *fmap.lifting fset.lifting*  
 $\langle \text{proof} \rangle$

**Correctness wrt *srewrite***

**context** *srules* **begin context** **begin**

**private lemma** *seval-correct0*:  
**assumes**  $rs, \Gamma \vdash_s t \downarrow u$  *closed-except*  $t$  (*fmdom*  $\Gamma$ ) *closed-env*  $\Gamma$   
**shows**  $rs \vdash_s \text{subst } t \Gamma \longrightarrow^* u$   
 $\langle \text{proof} \rangle$

**corollary** *seval-correct*:  
**assumes**  $rs, \text{fmempty} \vdash_s t \downarrow u$  *closed*  $t$   
**shows**  $rs \vdash_s t \longrightarrow^* u$   
 $\langle \text{proof} \rangle$

**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*  $\Rightarrow$  *bool* **where**  
*vrule*  $(-, \text{rhs}) \longleftrightarrow \text{vwellformed } \text{rhs} \wedge \text{vclosed } \text{rhs} \wedge \neg \text{is-Vconstr } \text{rhs}$

**locale** *vrules* = *constants* *C-info* *fst* |<sup>4</sup> *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*  $(\lambda(-, \text{rhs}). \text{not-shadows-vconsts } \text{rhs})$  *rs*

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

**lemma** *map*: *is-map* (*set rs*)  
 $\langle proof \rangle$

**end**

**abbreviation** *value-to-stern-rules* :: *vrule list*  $\Rightarrow$  *srule list* **where**  
*value-to-stern-rules*  $\equiv$  *map* (*map-prod id value-to-stern*)

**inductive** (**in** *special-constants*)

*veval* :: (*name*  $\times$  *value*) *list*  $\Rightarrow$  (*name, value*) *fmap*  $\Rightarrow$  *stern*  $\Rightarrow$  *value*  $\Rightarrow$  *bool* ( $-,$   
 $- / \vdash_v / - \downarrow / - [50,0,50] 50$ ) **for** *rs* **where**

*const*: (*name, rhs*)  $\in$  *set rs*  $\Rightarrow$  *rs,  $\Gamma \vdash_v$  Sconst name  $\downarrow$  rhs* |

*var*: *fmlookup*  $\Gamma$  *name* = *Some val*  $\Rightarrow$  *rs,  $\Gamma \vdash_v$  Svar name  $\downarrow$  val* |

*abs*: *rs,  $\Gamma \vdash_v$  Sabs cs  $\downarrow$  Vabs cs  $\Gamma$*  |

*comb*:

*rs,  $\Gamma \vdash_v$  t  $\downarrow$  Vabs cs  $\Gamma'$   $\Rightarrow$  rs,  $\Gamma \vdash_v$  u  $\downarrow$  u'  $\Rightarrow$*

*vfind-match cs u' = Some (env, -, rhs)  $\Rightarrow$*

*rs,  $\Gamma' ++_f$  env  $\vdash_v$  rhs  $\downarrow$  val  $\Rightarrow$*

*rs,  $\Gamma \vdash_v$  t  $\$s$  u  $\downarrow$  val* |

*rec-comb*:

*rs,  $\Gamma \vdash_v$  t  $\downarrow$  Vrecabs css name  $\Gamma'$   $\Rightarrow$*

*fmlookup css name = Some cs  $\Rightarrow$*

*rs,  $\Gamma \vdash_v$  u  $\downarrow$  u'  $\Rightarrow$*

*vfind-match cs u' = Some (env, -, rhs)  $\Rightarrow$*

*rs,  $\Gamma' ++_f$  env  $\vdash_v$  rhs  $\downarrow$  val  $\Rightarrow$*

*rs,  $\Gamma \vdash_v$  t  $\$s$  u  $\downarrow$  val* |

*constr*:

*name  $\in$  C  $\Rightarrow$*

*list-all2 (veval rs  $\Gamma$ ) ts us  $\Rightarrow$*

*rs,  $\Gamma \vdash_v$  name  $\$s$  ts  $\downarrow$  Vconstr name us*

**lemma** (**in** *vrules*) *veval-wellformed*:

**assumes** *rs,  $\Gamma \vdash_v$  t  $\downarrow$  v wellformed t wellformed-venv  $\Gamma$*

**shows** *vwellformed v*

$\langle proof \rangle$

**lemma** (**in** *vrules*) *veval-closed*:

**assumes** *rs,  $\Gamma \vdash_v$  t  $\downarrow$  v closed-except t (fmdom  $\Gamma$ ) closed-venv  $\Gamma$*

**assumes** *wellformed t wellformed-venv  $\Gamma$*

**shows** *vclosed v*

$\langle proof \rangle$

**lemma** (**in** *vrules*) *veval-constructor-value*:

**assumes** *rs,  $\Gamma \vdash_v$  t  $\downarrow$  v vconstructor-value-env  $\Gamma$*

**shows** *vconstructor-value v*

*<proof>*

**lemma** (in *vrules*) *veval-welldefined*:

**assumes** *rs*,  $\Gamma \vdash_v t \downarrow v$  *fmpred* ( $\lambda\cdot$ . *vwelldefined*)  $\Gamma$  *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-rules: srules C-info rs'*

*<proof>*

**end**

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

The proof idea is that we can always substitute with  $\Gamma'$  and obtain a regular *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  $\Gamma$  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*  $\Gamma$

**assumes** *closed-except* *t* (*fmdom*  $\Gamma$ ) *closed-venv*  $\Gamma$

**assumes** *vconstructor-value-env*  $\Gamma$

**shows** *rs'*, *fmap* *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*



*<proof>*

**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  $\Gamma'$  = *fmlookup-keys* ( $\lambda$ name cs. *Vrecabs* css name  $\Gamma'$ ) css

**context** *special-constants* **begin**

**inductive** *veval'* :: (name, value) fmap  $\Rightarrow$  *stern*  $\Rightarrow$  value  $\Rightarrow$  bool (-/  $\vdash_v$ / -  $\downarrow$ / - [50,0,50] 50) **where**

*const*: name  $\notin$  C  $\Longrightarrow$  *fmlookup*  $\Gamma$  name = *Some* val  $\Longrightarrow$   $\Gamma \vdash_v$  *Sconst* name  $\downarrow$  val |

*var*: *fmlookup*  $\Gamma$  name = *Some* val  $\Longrightarrow$   $\Gamma \vdash_v$  *Svar* name  $\downarrow$  val |

*abs*:  $\Gamma \vdash_v$  *Sabs* cs  $\downarrow$  *Vabs* cs  $\Gamma$  |

*comb*:

$\Gamma \vdash_v$  t  $\downarrow$  *Vabs* cs  $\Gamma'$   $\Longrightarrow$   $\Gamma \vdash_v$  u  $\downarrow$  u'  $\Longrightarrow$   
*vfind-match* cs u' = *Some* (env, -, rhs)  $\Longrightarrow$   
 $\Gamma' ++_f$  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  $\Gamma'$   $\Longrightarrow$   
*fmlookup* css name = *Some* cs  $\Longrightarrow$   
 $\Gamma \vdash_v$  u  $\downarrow$  u'  $\Longrightarrow$   
*vfind-match* cs u' = *Some* (env, -, rhs)  $\Longrightarrow$   
 $\Gamma' ++_f$  *mk-rec-env* css  $\Gamma'$   $++_f$  env  $\vdash_v$  rhs  $\downarrow$  val  $\Longrightarrow$   
 $\Gamma \vdash_v$  t  $\$s$  u  $\downarrow$  val |

*constr*: name  $\in$  C  $\Longrightarrow$  *list-all2* (*veval'*  $\Gamma$ ) ts us  $\Longrightarrow$   $\Gamma \vdash_v$  name  $\$s$  ts  $\downarrow$  *Vconstr* name us

**lemma** *veval'-sabs-svarE*:

**assumes**  $\Gamma \vdash_v$  *Sabs* cs  $\$s$  *Svar* n  $\downarrow$  v

**obtains** u' env pat rhs

**where** *fmlookup*  $\Gamma$  n = *Some* u'  
*vfind-match* cs u' = *Some* (env, pat, rhs)  
 $\Gamma ++_f$  env  $\vdash_v$  rhs  $\downarrow$  v

*<proof>*

**lemma** *veval'-wellformed*:

**assumes**  $\Gamma \vdash_v t \downarrow v$  *wellformed t wellformed-venv*  $\Gamma$   
**shows** *vwellformed v*  
*<proof>*

**lemma** (in *constants*) *veval'-shadows*:

**assumes**  $\Gamma \vdash_v t \downarrow v$  *not-shadows-vconsts-env*  $\Gamma \neg$  *shadows-consts t*  
**shows** *not-shadows-vconsts v*  
*<proof>*

**lemma** *veval'-closed*:

**assumes**  $\Gamma \vdash_v t \downarrow v$  *closed-except t (fmdom  $\Gamma$ ) closed-venv*  $\Gamma$   
**assumes** *wellformed t wellformed-venv*  $\Gamma$   
**shows** *vclosed v*  
*<proof>*

**primrec** *vwelldefined' :: value  $\Rightarrow$  bool* **where**

*vwelldefined' (Vconstr name vs)  $\longleftrightarrow$  list-all vwelldefined' vs |*  
*vwelldefined' (Vabs cs  $\Gamma$ )  $\longleftrightarrow$*   
*pred-fmap id (fmmap vwelldefined'  $\Gamma$ )  $\wedge$*   
*list-all ( $\lambda(pat, t). \text{consts } t \mid\subseteq\mid (fmdom \Gamma \mid\cup\mid C)$ ) cs  $\wedge$*   
*fdisjnt C (fmdom  $\Gamma$ ) |*  
*vwelldefined' (Vrecabs css name  $\Gamma$ )  $\longleftrightarrow$*   
*pred-fmap id (fmmap vwelldefined'  $\Gamma$ )  $\wedge$*   
*pred-fmap ( $\lambda cs.$*   
*list-all ( $\lambda(pat, t). \text{consts } t \mid\subseteq\mid fmdom \Gamma \mid\cup\mid (C \mid\cup\mid fmdom css)$ ) cs  $\wedge$*   
*fdisjnt C (fmdom  $\Gamma$ )) css  $\wedge$*   
*name  $\mid\in\mid fmdom css \wedge$*   
*fdisjnt C (fmdom css)*

**lemma** *vmatch-welldefined'*:

**assumes** *vmatch pat v = Some env vwelldefined' v*  
**shows** *fmpred ( $\lambda-. vwelldefined'$ ) env*  
*<proof>*

**lemma** *sconsts-list-comb*:

*consts (list-comb f xs)  $\mid\subseteq\mid S \longleftrightarrow \text{consts } f \mid\subseteq\mid S \wedge \text{list-all } (\lambda x. \text{consts } x \mid\subseteq\mid S) xs$*   
*<proof>*

**lemma** *sconsts-sabs*:

*consts (Sabs cs)  $\mid\subseteq\mid S \longleftrightarrow \text{list-all } (\lambda(-, t). \text{consts } t \mid\subseteq\mid S) cs$*   
*<proof>*

**lemma** (in *constants*) *veval'-welldefined'*:

**assumes**  $\Gamma \vdash_v t \downarrow v$  *fdisjnt C (fmdom  $\Gamma$ )*  
**assumes** *consts t  $\mid\subseteq\mid fmdom \Gamma \mid\cup\mid C$  fmpred ( $\lambda-. vwelldefined'$ )  $\Gamma$*   
**assumes** *wellformed t wellformed-venv*  $\Gamma$   
**assumes**  $\neg$  *shadows-consts t not-shadows-vconsts-env*  $\Gamma$   
**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  $\Rightarrow$  value  $\Rightarrow$  bool ( $\vdash_v / - \approx - [0, 50] 50$ ) where**

***constr*: list-all2 *vrelated* *ts us*  $\implies \vdash_v Vconstr\ name\ ts \approx Vconstr\ name\ us$  |**

***abs*:**

***fmrel-on-fset* (*frees* (*Sabs cs*)) *vrelated*  $\Gamma_1 \Gamma_2 \implies$**

***fmrel-on-fset* (*consts* (*Sabs cs*)) *vrelated* (*fmap-of-list rs*)  $\Gamma_2 \implies$**

**$\vdash_v Vabs\ cs\ \Gamma_1 \approx Vabs\ cs\ \Gamma_2$  |**

***rec-abs*:**

***pred-fmap* ( $\lambda cs.$**

***fmrel-on-fset* (*frees* (*Sabs cs*)) *vrelated*  $\Gamma_1 \Gamma_2 \wedge$**

***fmrel-on-fset* (*consts* (*Sabs cs*)) *vrelated* (*fmap-of-list rs*) ( $\Gamma_2 ++_f\ mk-rec-env$**

***css*  $\Gamma_2$ ) *css*  $\implies$**

***name*  $\in$  | *fmdom* *css*  $\implies$**

**$\vdash_v Vrecabs\ css\ name\ \Gamma_1 \approx Vrecabs\ css\ name\ \Gamma_2$**

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

**sublocale *vrelated*: value-struct-rel *vrelated***

*<proof>*

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

**lemma *veval'-correct*:**

**assumes  $\Gamma_2 \vdash_v t \downarrow v_2$  *wellformed t wellformed-venv*  $\Gamma_2$**

**assumes  $\neg$  *shadows-consts t not-shadows-vconsts-env*  $\Gamma_2$**

**assumes *welldefined t***

**assumes *fmpr* ( $\lambda-. vwelldefined$ )  $\Gamma_1$**

**assumes *fmrel-on-fset* (*frees t*) *vrelated*  $\Gamma_1 \Gamma_2$**

**assumes** *fmrel-on-fset* (*consts t*) *vrelated* (*fmap-of-list rs*)  $\Gamma_2$   
**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*  $\Gamma_2$   
**assumes**  $\neg$  *shadows-consts t not-shadows-vconsts-env*  $\Gamma_2$   
**assumes** *welldefined t*  
**assumes** *closed t*  
**assumes** *fmrel-on-fset* (*consts t*) *vrelated* (*fmap-of-list rs*)  $\Gamma_2$   
**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*  $\Gamma' \Gamma$   
**assumes** *closed-venv*  $\Gamma$  *closed-except t* (*fmdom*  $\Gamma$ )  
**assumes** *wellformed t wellformed-venv*  $\Gamma$  *fdisjnt C* (*fmdom*  $\Gamma$ )  
**assumes** *consts t*  $|\subseteq|$  *fmdom*  $\Gamma$   $|\cup|$  *C fmpred* ( $\lambda\cdot$ . *vwelldefined'*)  $\Gamma$   
**assumes**  $\neg$  *shadows-consts t not-shadows-vconsts-env*  $\Gamma$   
**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  $\Rightarrow$  term  $\Rightarrow$  'a  $\Rightarrow$  bool (-/  $\vdash$ / (-  $\approx$ / -) [50,0,50] 50)
  assumes eval-wellformed: rs  $\vdash$  t  $\approx$  a  $\implies$  wellformed t
begin

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

lemma eval'I[intro]:
```

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

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

**lemma**  $eval-trivI$ :  $rs \vdash t \approx a \implies rs \vdash t \downarrow a$   
 $\langle proof \rangle$

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

**end**

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

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

**instance**  
 $\langle proof \rangle$

**end**

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

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

**lemma**  $eval-ext$ :  
**assumes**  $wellformed\ f \wedge x\ t. rs \vdash t \downarrow x \implies rs \vdash f \$ t \downarrow a\ x$   
**shows**  $rs \vdash f \approx a$   
 $\langle proof \rangle$

**lemma**  $eval'-ext$ :  
**assumes**  $wellformed\ f \wedge x\ t. rs \vdash t \downarrow x \implies rs \vdash f \$ t \downarrow a\ x$   
**shows**  $rs \vdash f \downarrow a$   
 $\langle proof \rangle$

**lemma** *eval'-ext-alt*:

**fixes**  $f :: 'a::evaluate \Rightarrow 'b::evaluate$

**assumes**  $wellformed' \ 1 \ t \ \wedge \ u \ x. \ rs \vdash \ u \downarrow \ x \Longrightarrow rs \vdash \ t [u]_{\beta} \downarrow \ f \ x$

**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-elim*s

**context** *begin*

**private definition** *rewrite-step-term* ::  $term \times term \Rightarrow term \Rightarrow term \ option$

**where**

$rewrite-step-term = rewrite-step$

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

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

*<ML>*

**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*<sub>1</sub> \$ *t*<sub>2</sub>) (*u*<sub>1</sub> \$ *u*<sub>2</sub>)  $\longleftrightarrow$  *non-overlapping'* *t*<sub>1</sub> *u*<sub>1</sub>  $\vee$  *non-overlapping'* *t*<sub>2</sub> *u*<sub>2</sub> |

*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*<sub>1</sub> \$ *t*<sub>2</sub>) (*u*<sub>1</sub> \$ *u*<sub>2</sub>)  $\longleftrightarrow$  *pattern-compatible'* *t*<sub>1</sub> *u*<sub>1</sub>  $\wedge$  (*t*<sub>1</sub> = *u*<sub>1</sub>  $\longrightarrow$  *pattern-compatible'* *t*<sub>2</sub> *u*<sub>2</sub>) |

*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* ( $\lambda$ (*lhs*<sub>1</sub>, -) (*lhs*<sub>2</sub>, -). *pattern-compatible'* *lhs*<sub>1</sub> *lhs*<sub>2</sub>)

**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* ( $\lambda$ (*lhs*, -).  $\neg$  *pre-constants.shadows-consts* *C-info* (*heads-of* *rs*) *lhs*)  $\wedge$



*fdisjnt (heads-of rs) (constructors.C C-info)  $\wedge$   
 fBall rs ( $\lambda(-, rhs). \text{pre-constants.welldefined C-info (heads-of rs) rhs}$ )  $\wedge$   
 distinct (constructors.all-constructors C-info)*

**lemma** *rules-approx:*

**assumes** *rules' C-info rs*

**shows** *rules C-info rs*

*<proof>*

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

*<proof>*

*<ML>*

**consts** *lift-term :: 'a  $\Rightarrow$  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*  
++<sub>f</sub> *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*<sub>Λ</sub> *funs'* *f*)) *funs* *env*) =  
*fmap-of-ns* *env* ++<sub>f</sub> *fmap-of-list* (map (λ(*f*, -). (*Name* *f*, *Recclosure* *env*<sub>Λ</sub> *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*<sub>Λ</sub> *env*) = *fmap-of-ns* *env* ++<sub>f</sub> *cake-mk-rec-env*  
*funs* *env*<sub>Λ</sub>  
⟨*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* ( $\lambda(\text{name}, \text{dt-def}). (\text{map } \text{as-string } (\text{tparams } \text{dt-def}), \text{as-string name},$

$\text{map } (\lambda(C, \text{params}). (\text{as-string } C, \text{map } \text{typ-to-t } \text{params}))$

$(\text{sorted-list-of-fmap } (\text{constructors } \text{dt-def})))$

$(\text{sorted-list-of-fmap } C\text{-info})$ )

**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 ( $\lambda$ -. cake-all-types  $\cup$  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 ( $\lambda$ name. TypeId (Short (as-string name)))) (sorted-list-of-fset (fmdom
C-info)))
<proof> lemma cakeml-convoluted1: foldr ( $\lambda$ (n, ts). ( $\#$ ) n) ys xs = map fst ys @
xs
<proof> lemma cakeml-convoluted2: foldr ( $\lambda$ x y. f x @ y) xs ys = concat (map f
xs) @ ys
<proof> lemma check-dup-ctors-alt-def: check-dup-ctors tds  $\longleftrightarrow$  distinct (tds  $\gg$ 
( $\lambda$ (-, -, 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' (t1 $ t2) = frees' t2 @ frees' t1 |
  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  $\Rightarrow$  stern  $\Rightarrow$  exp and
mk-clauses :: name fset  $\Rightarrow$  (term  $\times$  stern) list  $\Rightarrow$  (Ast.pat  $\times$  exp) list and
mk-con :: name fset  $\Rightarrow$  stern  $\Rightarrow$  exp where
mk-exp - (Svar s) = Ast.Var (Short (as-string s)) |
mk-exp - (Sconst s) = (if s  $\in$  C then Ast.Con (Some (Short (as-string s))) [] else
Ast.Var (Short (as-string s))) |
mk-exp S (t1 $s t2) = Ast.App Ast.Opapp [mk-con S t1, mk-con S t2] |
mk-exp S (Sabs cs) = (
  let n = fresh-fNext S in
  Ast.Fun (as-string n) (Ast.Mat (Ast.Var (Short (as-string n))) (mk-clauses S
cs))) |
mk-con S t =
  (case strip-comb t of
    (Sconst c, args)  $\Rightarrow$ 
      if c  $\in$  C then Ast.Con (Some (Short (as-string c))) (map (mk-con S) args)
  else mk-exp S t
  | -  $\Rightarrow$  mk-exp S t) |
mk-clauses S cs = map ( $\lambda$ (pat, t). (mk-ml-pat (mk-pat pat), mk-con (frees pat  $\cup$ 
S) t)) cs
```

**context begin**

**private lemma** *mk-exp-cupcake0*:

```
wellformed t  $\implies$  is-cupcake-exp (mk-exp S t)
wellformed-clauses cs  $\implies$  cupcake-clauses (mk-clauses S cs)  $\wedge$  cake-linear-clauses
(mk-clauses S cs)
wellformed t  $\implies$  is-cupcake-exp (mk-con S t)
<proof>
```

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

**lemma** *mk-exp-cupcake*:

```
wellformed t  $\implies$  is-cupcake-exp (mk-exp S t)
wellformed t  $\implies$  is-cupcake-exp (mk-con S t)
<proof>
```

**end**

**definition** *mk-letrec-body* **where**

```
mk-letrec-body S rs = (
  map ( $\lambda$ (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 |<sup>†</sup> 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 |<sup>†</sup> 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* | $\notin$ | *C*  $\implies$  *related-exp* (*Sconst name*) (*Ast.Var* (*Short* (*as-string name*)))  
|  
*constr*: *name* | $\in$ | *C*  $\implies$  *list-all2* *related-exp* *ts es*  $\implies$   
*related-exp* (*name* \$\$ *ts*) (*Ast.Con* (*Some* (*Short* (*as-string name*))) *es*) |  
*app*: *related-exp* *t*<sub>1</sub> *u*<sub>1</sub>  $\implies$  *related-exp* *t*<sub>2</sub> *u*<sub>2</sub>  $\implies$  *related-exp* (*t*<sub>1</sub> \$<sub>s</sub> *t*<sub>2</sub>) (*Ast.App*  
*Ast.Opapp* [*u*<sub>1</sub>, *u*<sub>2</sub>]) |  
*fun*: *list-all2* (*rel-prod* *related-pat* *related-exp*) *cs ml-cs*  $\implies$   
*n* | $\notin$ | *ids* (*Sabs cs*)  $\implies$  *n* | $\notin$ | *all-consts*  $\implies$   
*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*  $\implies$   
*related-exp* *scr ml-scr*  $\implies$   
*related-exp* (*Sabs cs* \$<sub>s</sub> *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*  $\times$  *stern*) *list*  $\Rightarrow$  *name*  $\Rightarrow$  *exp*  $\Rightarrow$  *bool* **where**  
*related-fun cs n e*  $\longleftrightarrow$   
*n* | $\notin$ | *ids* (*Sabs cs*)  $\wedge$  *n* | $\notin$ | *all-consts*  $\wedge$  (*case e of*  
(*Ast.Mat* (*Ast.Var* (*Short n'*)) *ml-cs*)  $\Rightarrow$   
*n* = *Name n'*  $\wedge$  *list-all2* (*rel-prod* *related-pat* *related-exp*) *cs ml-cs*  
| -  $\Rightarrow$  *False*)

**lemma** *related-fun-alt-def*:  
*related-fun cs n* (*Ast.Mat* (*Ast.Var* (*Short* (*as-string n*))) *ml-cs*)  $\longleftrightarrow$   
*list-all2* (*rel-prod* *related-pat* *related-exp*) *cs ml-cs*  $\wedge$   
*n* | $\notin$ | *ids* (*Sabs cs*)  $\wedge$  *n* | $\notin$ | *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* | $\notin$ | *ids* (*Sabs cs*)  
*n* | $\notin$ | *all-consts*  
**and** *list-all2* (*rel-prod* *related-pat* *related-exp*) *cs ml-cs*  
<*proof*>

**lemma** *related-exp-fun*:  
*related-fun cs n e*  $\longleftrightarrow$  *related-exp* (*Sabs cs*) (*Ast.Fun* (*as-string n*) *e*)  $\wedge$  *n* | $\notin$ | *ids*  
(*Sabs cs*)  $\wedge$  *n* | $\notin$ | *all-consts*  
(**is** *?lhs*  $\longleftrightarrow$  *?rhs*)  
<*proof*>

**inductive** *related-v* :: *value*  $\Rightarrow$  *v*  $\Rightarrow$  *bool* **where**  
*conv*:

*list-all2 related-v us vs*  $\implies$   
*related-v* (*Vconstr name us*) (*Conv* (*Some* (*as-string name, -*) *vs*) |  
*closure*:  
*related-fun cs n e*  $\implies$   
*fmrel-on-fset* (*ids* (*Sabs cs*)) *related-v*  $\Gamma$  (*fmap-of-ns* (*sem-env.v env*))  $\implies$   
*related-v* (*Vabs cs*  $\Gamma$ ) (*Closure env* (*as-string n*) *e*) |  
*rec-closure*:  
*fmrel-on-fset* (*fbind* (*fmran css*) (*ids*  $\circ$  *Sabs*)) *related-v*  $\Gamma$  (*fmap-of-ns* (*sem-env.v env*))  $\implies$   
*fmrel* ( $\lambda cs. \lambda(n, e). \text{related-fun } cs \ n \ e$ ) *css* (*fmap-of-list* (*map* (*map-prod Name* (*map-prod Name id*)) *exps*))  $\implies$   
*related-v* (*Vrecabs css name*  $\Gamma$ ) (*Recclosure env exps* (*as-string name*))

**abbreviation** *var-env* :: (*name, value*) *fmap*  $\Rightarrow$  (*string*  $\times$  *v*) *list*  $\Rightarrow$  *bool* **where**  
*var-env*  $\Gamma$  *ns*  $\equiv$  *fmrel related-v*  $\Gamma$  (*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*  
 $\langle \text{proof} \rangle$

**context begin**

**private inductive** *match-result-related* :: (*string*  $\times$  *v*) *list*  $\Rightarrow$  (*string*  $\times$  *v*) *list*  
*match-result*  $\Rightarrow$  (*name, value*) *fmap option*  $\Rightarrow$  *bool* **for** *eenv* **where**  
*no-match*: *match-result-related eenv No-match None* |  
*error*: *match-result-related eenv Match-type-error -* |  
*match*: *var-env*  $\Gamma$  *eenv-m*  $\implies$  *match-result-related eenv* (*Match* (*eenv-m*  $\text{@}$  *eenv*))  
(*Some*  $\Gamma$ )

**private corollary** *match-result-related-empty*: *match-result-related eenv* (*Match eenv*) (*Some fmempty*)  
 $\langle \text{proof} \rangle$  **fun** *is-Match* :: '*a* *match-result*  $\Rightarrow$  *bool* **where**  
*is-Match* (*Match -*)  $\longleftrightarrow$  *True* |  
*is-Match -*  $\longleftrightarrow$  *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*)  
 $\langle \text{proof} \rangle$

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

**obtains** *rhs pat*  $\Gamma$  **where**  
*ml-pat* = *mk-ml-pat* (*mk-pat pat*)  
*related-exp rhs ml-rhs*  
*vfind-match cs v* = *Some* ( $\Gamma$ , *pat*, *rhs*)  
*var-env*  $\Gamma$  *envv*  
 <*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* ( $\lambda cs (n, e). \text{related-fun } cs \ n \ e$ ) *css* (*fmap-of-list* (*map* (*map-prod* *Name* (*map-prod Name id*)) *funs*))

**assumes** *fmrel-on-fset* (*fbind* (*fmrans* *css*) (*ids*  $\circ$  *Sabs*)) *related-v*  $\Gamma_\Lambda$  (*fmap-of-ns* (*sem-env.v env* $_\Lambda$ ))

**shows** *fmrel related-v* (*mk-rec-env* *css*  $\Gamma_\Lambda$ ) (*cake-mk-rec-env* *funs* *env* $_\Lambda$ )

<*proof*>

**lemma** *mk-exp-correctness*:

*ids t*  $| \subseteq | S \implies \text{all-consts } | \subseteq | S \implies \neg \text{shadows-consts } t \implies \text{related-exp } t$  (*mk-exp* *S t*)

*ids* (*Sabs cs*)  $| \subseteq | S \implies \text{all-consts } | \subseteq | S \implies \neg \text{shadows-consts } (Sabs \ cs) \implies \text{list-all2 } (\text{rel-prod } \text{related-pat } \text{related-exp}) \ cs$  (*mk-clauses S cs*)

*ids t*  $| \subseteq | S \implies \text{all-consts } | \subseteq | S \implies \neg \text{shadows-consts } t \implies \text{related-exp } t$  (*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*  $\Gamma$  (*fmap-of-ns* (*sem-env.v env*))

**assumes** *related-exp t exp*

**assumes** *wellformed t wellformed-venv*  $\Gamma$

**assumes** *closed-venv*  $\Gamma$  *closed-except t* (*fmdom*  $\Gamma$ )

**assumes** *fmpred* ( $\lambda-. \text{vwelldefined'}$ )  $\Gamma$  *consts t*  $| \subseteq | \text{fmdom } \Gamma \cup | C$

**assumes** *fdisjnt C* (*fmdom*  $\Gamma$ )

**assumes**  $\neg$  *shadows-consts t not-shadows-vconsts-env*  $\Gamma$   
**shows** *if-rval*  $(\lambda ml-v. \exists v. \Gamma \vdash_v t \downarrow v \wedge \text{related-}v\ v\ ml-v)$   $r$   
 $\langle$ *proof* $\rangle$

**theorem** *semantic-correctness*:

**fixes** *exp*  
**assumes** *cupcake-evaluate-single env exp*  $(Rval\ ml-v)$  *is-cupcake-all-env env*  
**assumes** *fmrel-on-fset*  $(ids\ t)$  *related-v*  $\Gamma$   $(fmap-of-ns\ (sem-env.v\ env))$   
**assumes** *related-exp t exp*  
**assumes** *wellformed t wellformed-venv*  $\Gamma$   
**assumes** *closed-venv*  $\Gamma$  *closed-except t*  $(fmdom\ \Gamma)$   
**assumes** *fmpred*  $(\lambda-. vwelldefined')$   $\Gamma$  *consts t*  $|\subseteq|$   $fmdom\ \Gamma$   $|\cup|$   $C$   
**assumes** *fdisjnt C*  $(fmdom\ \Gamma)$   
**assumes**  $\neg$  *shadows-consts t not-shadows-vconsts-env*  $\Gamma$   
**obtains**  $v$  **where**  $\Gamma \vdash_v t \downarrow v$  *related-v v ml-v*  
 $\langle$ *proof* $\rangle$

**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*  $(Some\ (Short\ "String-char-Char"))$   $(map\ (\lambda n. Ast.Pvar\ ("b" @ show\ n))\ [0..<8])$

**definition** *cake-plus*  $::$  *exp*  $\Rightarrow$  *exp*  $\Rightarrow$  *exp* **where**

*cake-plus t u* = *Ast.App*  $(Ast.Opn\ Ast.Plus)$   $[t, u]$

**lemma** *cake-plus-correct*:

**assumes** *evaluate env s u* =  $(s', Rval\ (Litv\ (IntLit\ y)))$

**assumes** *evaluate env s' t* =  $(s'', Rval\ (Litv\ (IntLit\ x)))$

**shows** *evaluate env s (cake-plus t u)* =  $(s'', Rval\ (Litv\ (IntLit\ (x + y))))$

$\langle$ *proof* $\rangle$

**definition** *cake-times*  $::$  *exp*  $\Rightarrow$  *exp*  $\Rightarrow$  *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*  $\Rightarrow$  *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* ( $\lambda n.$  *cake-times* (*Lit* (*IntLit* ( $2 \wedge n$ ))) (*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**  $rs \vdash_p \text{stern-to-pterm } u \longrightarrow^* \text{stern-to-pterm } u'$

*<proof>*

**lemma** *srewrite-stepD*:

**assumes** *srewrite-step*  $rs \text{ name } t$

**shows**  $(\text{name}, t) \in \text{set } rs$

*<proof>*

**lemma** (in *srules*) *srewrite-wellformed*:

**assumes**  $rs \vdash_s t \longrightarrow t' \text{ wellformed } t$

**shows** *wellformed*  $t'$

*<proof>*

**lemma** (in *srules*) *srewrite-wellformed-rt*:

**assumes**  $rs \vdash_s t \longrightarrow^* t' \text{ wellformed } t$

**shows** *wellformed*  $t'$

*<proof>*

**lemma** *vno-abs-value-to-stern*: *no-abs* (*value-to-stern*  $v$ )  $\longleftrightarrow$  *vno-abs*  $v$  **for**  $v$

*<proof>*

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

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

**assumes** *compile*  $\vdash_n u \longrightarrow^* u' \text{ closed } u$

**shows**  $rs \vdash \text{nterm-to-term}' u \longrightarrow^* \text{nterm-to-term}' u'$

*<proof>*

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

**corollary** (in *irules*) *transform-correct-rt*:

**assumes** *transform-irule-set*  $rs \vdash_i u \longrightarrow^* u'' t \approx_p u \text{ closed } t$

**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*  $rs \vdash_i t \longrightarrow^* u \text{ closed } t \text{ no-abs } u$

**shows**  $rs \vdash_i t \longrightarrow^* u$

*<proof>*

**corollary** *transform-correct-rt-n-no-abs0*:

**assumes** *irules*  $C \text{ } rs$  (*transform-irule-set*  $\widehat{\sim} n$ )  $rs \vdash_i t \longrightarrow^* u \text{ closed } t \text{ no-abs } u$

**shows**  $rs \vdash_i t \longrightarrow^* u$

*<proof>*

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

**assumes**  $(\text{transform-irule-set } \overset{\sim}{\sim} n) \text{ rs } \vdash_i t \longrightarrow^* u \text{ closed } t \text{ no-abs } u$   
**shows**  $\text{rs } \vdash_i t \longrightarrow^* u$   
 $\langle \text{proof} \rangle$

**hide-fact**  $\text{transform-correct-rt-n-no-abs0}$

### 6.1.5 Iterated application of $\text{transform-irule-set}$ .

**definition**  $\text{max-arity} :: \text{irule-set} \Rightarrow \text{nat}$  **where**  
 $\text{max-arity } \text{rs} = \text{fMax } ((\text{arity} \circ \text{snd}) \mid^{\dagger} \text{rs})$

**lemma**  $\text{rules-transform-iter0}$ :  
**assumes**  $\text{irules } C\text{-info } \text{rs}$   
**shows**  $\text{irules } C\text{-info } ((\text{transform-irule-set } \overset{\sim}{\sim} n) \text{rs})$   
 $\langle \text{proof} \rangle$

**lemma** **(in**  $\text{irules}$ **)**  $\text{rules-transform-iter}$ :  $\text{irules } C\text{-info } ((\text{transform-irule-set } \overset{\sim}{\sim} n) \text{rs})$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{transform-irule-set-n-heads}$ :  $\text{fst } \mid^{\dagger} ((\text{transform-irule-set } \overset{\sim}{\sim} n) \text{rs}) = \text{fst } \mid^{\dagger} \text{rs}$   
 $\langle \text{proof} \rangle$

**hide-fact**  $\text{rules-transform-iter0}$

**definition**  $\text{transform-irule-set-iter} :: \text{irule-set} \Rightarrow \text{irule-set}$  **where**  
 $\text{transform-irule-set-iter } \text{rs} = (\text{transform-irule-set } \overset{\sim}{\sim} \text{max-arity } \text{rs}) \text{rs}$

**lemma**  $\text{transform-irule-set-iter-heads}$ :  $\text{fst } \mid^{\dagger} \text{transform-irule-set-iter } \text{rs} = \text{fst } \mid^{\dagger} \text{rs}$   
 $\langle \text{proof} \rangle$

**lemma** **(in**  $\text{irules}$ **)**  $\text{finished-alt-def}$ :  $\text{finished } \text{rs} \longleftrightarrow \text{max-arity } \text{rs} = 0$   
 $\langle \text{proof} \rangle$

**lemma** **(in**  $\text{irules}$ **)**  $\text{transform-finished-id}$ :  $\text{finished } \text{rs} \Longrightarrow \text{transform-irule-set } \text{rs} = \text{rs}$   
 $\langle \text{proof} \rangle$

**lemma** **(in**  $\text{irules}$ **)**  $\text{max-arity-decr}$ :  $\text{max-arity } (\text{transform-irule-set } \text{rs}) = \text{max-arity } \text{rs} - 1$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{max-arity-decr}'0$ :  
**assumes**  $\text{irules } C \text{rs}$   
**shows**  $\text{max-arity } ((\text{transform-irule-set } \overset{\sim}{\sim} n) \text{rs}) = \text{max-arity } \text{rs} - n$   
 $\langle \text{proof} \rangle$

**lemma** **(in**  $\text{irules}$ **)**  $\text{max-arity-decr}'$ :  $\text{max-arity } ((\text{transform-irule-set } \overset{\sim}{\sim} n) \text{rs}) =$



*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 |<sup>q</sup> 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 |<sup>q</sup> fset-of-list as-vrules = fst |<sup>q</sup> 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 |<sup>q</sup> 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*  $\vdash_v t \downarrow v$  *wellformed t closed t*

**shows** *rs, fmempty*  $\vdash_s t \downarrow$  *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*  $\equiv$  *srules-as-vrules.vrelated*

**notation** *srules-as-vrules.vrelated* ( $\vdash_v / - \approx - [0, 50] 50$ )

**lemma** *vrecabs-global-css-refl:*

**assumes** *name*  $|\in|$  *fmdom global-css*

**shows**  $\vdash_v$  *Vrecabs global-css name fmempty*  $\approx$  *Vrecabs global-css name fmempty*

*<proof>*

**lemma** *as-vrules-refl-rs: fmrel-on-fset (fst | $\uparrow$ | 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*  $\vdash_v t \downarrow v$

**assumes** *wellformed t*

**assumes**  $\neg$  *shadows-consts t*

**assumes** *welldefined t*

**assumes** *closed t*

**assumes** *vno-abs v*

**shows** *as-vrules, fmempty*  $\vdash_v t \downarrow v$

*<proof>*

**end**

### 6.1.8 CakeML

**context** *srules* **begin**

**definition** *as-sem-env* :: *v sem-env*  $\Rightarrow$  *v sem-env* **where**

*as-sem-env env* = ( $\lfloor$  *sem-env.v* = *build-rec-env* (*mk-letrec-body all-consts rs*) *env*  
*nsEmpty*, *sem-env.c* = *nsEmpty*  $\rfloor$ )

**lemma** *compile-sem-env*:

*evaluate-dec ck mn env state* (*compile-group all-consts rs*) (*state*, *Rval* (*as-sem-env env*))  
 $\langle$ *proof* $\rangle$

**lemma** *compile-sem-env'*:

*fun-evaluate-decs mn state env* [(*compile-group all-consts rs*)] = (*state*, *Rval* (*as-sem-env env*))  
 $\langle$ *proof* $\rangle$

**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* ( $\lfloor$  *sem-env.v* = *nsEmpty*, *sem-env.c* = *nsEmpty*  $\rfloor$ )))  
 $\langle$ *proof* $\rangle$

**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* ( $\lfloor$  *sem-env.v* = *nsEmpty*, *sem-env.c* = *nsEmpty*  $\rfloor$ )))  
 $\langle$ *proof* $\rangle$

**definition** *sem-env* :: *v sem-env* **where**

*sem-env*  $\equiv$  *extend-dec-env* (*as-sem-env empty-sem-env*) *empty-sem-env*

**lemma** *cupcake-sem-env*: *is-cupcake-all-env sem-env*

$\langle$ *proof* $\rangle$

**lemma** *sem-env-reft*: *fmrel related-v* (*fmap-of-list as-vrules*) (*fmap-of-ns* (*sem-env.v sem-env*))

$\langle$ *proof* $\rangle$

**including** *fmap.lifting*

$\langle$ *proof* $\rangle$

**lemma** *semantic-correctness'*:

**assumes** *cupcake-evaluate-single sem-env* (*mk-con all-consts t*) (*Rval ml-v*)

**assumes** *welldefined t closed t*  $\neg$  *shadows-consts t wellformed t*

**obtains** *v* **where** *fmap-of-list as-vrules*  $\vdash_v$  *t*  $\downarrow$  *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 v1 ml-v related-v v2 ml-v cake-no-abs ml-v
  shows v1 = v2
  ⟨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 ⊢v t ↓ cake-to-value ml-v
  ⟨proof⟩

end

```

## 6.1.9 Composition

**context** *rules* **begin**

**abbreviation** *term-to-nterm* **where**

*term-to-nterm*  $t \equiv \text{fresh-frun } (\text{Term-to-Nterm.term-to-nterm } [] t) \text{ all-consts}$

**abbreviation** *stern-to-cake* **where**

*stern-to-cake*  $\equiv \text{rules-as-nrules.crules-as-irules'.irules'-as-prules.prules-as-srules.mk-con all-consts}$

**abbreviation** *term-to-cake*  $t \equiv \text{stern-to-cake } (\text{pterm-to-stern } (\text{nterm-to-pterm } (\text{term-to-nterm } t)))$

**abbreviation** *cake-to-term*  $t \equiv (\text{convert-term } (\text{value-to-stern } (\text{cake-to-value } t)) :: \text{term})$

**abbreviation** *cake-sem-env*  $\equiv \text{rules-as-nrules.crules-as-irules'.irules'-as-prules.prules-as-srules.sem-env}$

**definition** *compiled*  $\equiv \text{rules-as-nrules.crules-as-irules'.irules'-as-prules.prules-as-srules.as-vrules}$

**lemma** *fndom-compiled*:  $\text{fndom } (\text{fmap-of-list } \text{compiled}) = \text{heads-of } \text{rs}$

*<proof>*

**lemma** *cake-semantic-correctness*:

**assumes** *cupcake-evaluate-single* *cake-sem-env*  $(\text{stern-to-cake } t) \text{ (Rval ml-v)}$

**assumes** *welldefined*  $t \text{ closed } t \neg \text{shadows-consts } t \text{ wellformed } t$

**assumes** *cake-no-abs*  $\text{ml-v}$

**shows**  $\text{fmap-of-list } \text{compiled} \vdash_v t \downarrow \text{cake-to-value } \text{ml-v}$

*<proof>*

Lo and behold, this is the final correctness theorem!

**theorem** *compiled-correct*:

— If CakeML evaluation of a term succeeds ...

**assumes**  $\exists k. \text{Evaluate-Single.evaluate } \text{cake-sem-env } (s \text{ (| clock := k |)}) (\text{term-to-cake } t) = (s', \text{Rval ml-v})$

— ... producing a constructor term without closures ...

**assumes** *cake-no-abs*  $\text{ml-v}$

— ... and some syntactic properties of the involved terms hold ...

**assumes**  $\text{closed } t \neg \text{shadows-consts } t \text{ welldefined } t \text{ wellformed } t$

— ... then this evaluation can be reproduced in the term-rewriting semantics

**shows**  $\text{rs} \vdash t \longrightarrow^* \text{cake-to-term } \text{ml-v}$

*<proof>*

**end**

**end**

## 6.2 Executable compilation chain

```
theory Compiler
imports Composition
begin
```

```
definition term-to-exp :: C-info  $\Rightarrow$  rule fset  $\Rightarrow$  term  $\Rightarrow$  exp where
term-to-exp C-info rs t =
  cakeml.mk-con C-info (heads-of rs  $\cup$  | constructors.C C-info)
  (pterm-to-stern (nterm-to-pterm (fresh-frun (term-to-nterm [] t) (heads-of rs
 $\cup$  | constructors.C C-info))))
```

```
lemma (in rules) Compiler.term-to-exp C-info rs = term-to-cake
   $\langle$ proof $\rangle$ 
```

```
primrec compress-pterm :: pterm  $\Rightarrow$  pterm where
compress-pterm (Pabs cs) = Pabs (fcompress (map-prod id compress-pterm | $\uparrow$  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
   $\langle$ proof $\rangle$ 
```

```
definition compress-crule-set :: crule-set  $\Rightarrow$  crule-set where
compress-crule-set = fcompress  $\circ$  fimage (map-prod id fcompress)
```

```
definition compress-irule-set :: irule-set  $\Rightarrow$  irule-set where
compress-irule-set = fcompress  $\circ$  fimage (map-prod id (fcompress  $\circ$  fimage (map-prod
id compress-pterm))))
```

```
definition compress-prule-set :: prule fset  $\Rightarrow$  prule fset where
compress-prule-set = fcompress  $\circ$  fimage (map-prod id compress-pterm)
```

```
lemma compress-crule-set-eq[simp]: compress-crule-set rs = rs
   $\langle$ proof $\rangle$ 
```

```
lemma compress-irule-set-eq[simp]: compress-irule-set rs = rs
   $\langle$ proof $\rangle$ 
```

```
lemma compress-prule-set[simp]: compress-prule-set rs = rs
   $\langle$ proof $\rangle$ 
```

```
definition transform-irule-set-iter :: irule-set  $\Rightarrow$  irule-set where
transform-irule-set-iter rs = ((transform-irule-set  $\circ$  compress-irule-set)  $\rightsquigarrow$  max-arity
rs) rs
```

```
definition as-sem-env :: C-info  $\Rightarrow$  srule list  $\Rightarrow$  v sem-env  $\Rightarrow$  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*) |*U*| *con-*  
*structors.C* *C-info*) *rs*) *env* *nsEmpty*,  
   *sem-env.c* =  
     *nsEmpty* | )

**definition** *empty-sem-env* :: *C-info*  $\Rightarrow$  *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*  $\Rightarrow$  *srule list*  $\Rightarrow$  *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*  $\Rightarrow$  *rule fset*  $\Rightarrow$  *Ast.prog* **where**

*compile* *C-info* =  
*CakeML-Backend.compile'* *C-info*  $\circ$   
*Rewriting-Sterm.compile*  $\circ$   
*compress-prule-set*  $\circ$   
*Rewriting-Pterm.compile*  $\circ$   
*transform-irule-set-iter*  $\circ$   
*compress-irule-set*  $\circ$   
*Rewriting-Pterm-Elim.compile*  $\circ$   
*compress-crule-set*  $\circ$   
*Rewriting-Nterm.consts-of*  $\circ$   
*fcompress*  $\circ$   
*Rewriting-Nterm.compile'* *C-info*  $\circ$   
*fcompress*

**definition** *compile-to-env* :: *C-info*  $\Rightarrow$  *rule fset*  $\Rightarrow$  *v* *sem-env* **where**

*compile-to-env* *C-info* =  
*sem-env* *C-info*  $\circ$   
*Rewriting-Sterm.compile*  $\circ$   
*compress-prule-set*  $\circ$   
*Rewriting-Pterm.compile*  $\circ$   
*transform-irule-set-iter*  $\circ$   
*compress-irule-set*  $\circ$   
*Rewriting-Pterm-Elim.compile*  $\circ$   
*compress-crule-set*  $\circ$   
*Rewriting-Nterm.consts-of*  $\circ$   
*fcompress*  $\circ$   
*Rewriting-Nterm.compile'* *C-info*  $\circ$   
*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
```