Pattern Matches in HOL: A New Representation and Improved Code Generation

Thomas Tuerk[1] Magnus O. Myreen[2,3] Ramana Kumar[3]

Independent Scholar

CSE Department, Chalmers University of Technology

Computer Laboratory, University of Cambridge

ITP 2015, 24th August 2015

Overview

1. State of the Art
2. New Representation
3. Integration with CakeML
4. Examples
5. Conclusion

Overview

- pattern matching ubiquitous in functional programming
- pattern matching is a powerful technique
- it helps to write concise, readable definitions
- very handy and frequently used for interactive theorem proving in higher-order logic (HOL)
- however, it is not directly supported by HOL
- common representations
  - decision trees
  - sets of (conditional) equations

Decision Trees

- all major higher order logic systems use decision trees based on case-constants
  - **HOL 4**
  - Isabelle/HOL
  - ProofPower
  - HOL Light
  - Coq
  - ...
- algebraic datatype definitions introduce these case-constants
- the parser contains a pattern compilation algorithm that turns pattern matches in the input into decision trees
- pretty printers try to present these decision trees nicely

Decision Trees (Zip Example)

List Case Constant

```
list_case Nil n f = n      list_case (Cons y ys) n f = f y ys
```

Zipping Lists

```
zip l1 l2 = case (l1, l2) of
 | (Cons x l1', Cons y l2') =>
     Cons (x,y) (zip l1' l2')
 | _ => Nil
```

Zipping Lists (1)

```
zip l1 l2 =
  list_case l1 Nil (λ x l1'.
    list_case l2 Nil (λ y l2'.
      (Cons (x, y) (zip l1' l2'))))
```

- parser uses pattern compilation to produce a decision tree

Decision Trees (Zip Example)

List Case Constant

```
list_case Nil n f = n      list_case (Cons y ys) n f = f y ys
```

Zipping Lists

```
zip l1 l2 = case (l1, l2) of
 | (Cons x l1', Cons y l2') =>
     Cons (x,y) (zip l1' l2')
 | _ => Nil
```

Zipping Lists (2)

```
zip l1 l2 =
  list_case l2 Nil (λ y l2'.
    list_case l1 Nil (λ x l1'.
      (Cons (x, y) (zip l1' l2'))))
```

- parser uses pattern compilation to produce a decision tree
- exact result depends on pattern compilation heuristics

Decision Trees (Zip Example)

List Case Constant

```
list_case Nil n f = n      list_case (Cons y ys) n f = f y ys
```

Zipping Lists

```
zip l1 l2 = case (l1, l2) of
 | (Cons x l1', Cons y l2') =>
     Cons (x,y) (zip l1' l2')
 | _ => Nil
```

Zipping Lists (Pretty Printed)

```
zip l1 l2 = case (l1, l2) of
  | (Nil, _) => Nil
  | (Cons x l1', Nil) => Nil
  | (Cons x l1', Cons y l2') =>
       Cons (x,y) (zip l1' l2')
```

- parser uses pattern compilation to produce a decision tree
- exact result depends on pattern compilation heuristics
- pretty printer hides details

Decision Trees (RBT balancing)

Red-Black-Tree Constructors

```
Empty    Red    Black
```

Balancing (Okasaki, C.: Purely Functional Data Structures)

```
case (a,b) of
  (Red (Red a x b) y c,d) => Red (Black a x b) y (Black c n d)
| (Red a x (Red b y c),d) => Red (Black a x b) y (Black c n d)
| (a,Red (Red b y c) z d) => Red (Black a n b) y (Black c z d)
| (a,Red b y (Red c z d)) => Red (Black a n b) y (Black c z d)
| other => Black a n b
```

- optimal decision tree contains 57 cases
- naive compilation likely to produce more
- last case alone apears 36 times in result

Decision Trees (Discussion)

Issues

- big gap between user's view and internal representation
- pattern compilation trustworthy?
- non-trivial pretty printer trustworthy?
- size explosion and obfuscated structure
  - extracted code often complicated and slow
  - proofs lengthy and artificial

Function Definition Packages I

- various function definition packages provide an alternative to decision trees
- for top-level pattern-matches, they produce a set of (conditional) equations

Zip example revisited

```
|- ∀ l2. zip Nil l2 = Nil
|- ∀ l1. zip l1 Nil = Nil
|- ∀ x y l1’ l2’. zip (Cons x l1’, Cons y l2’) =
     Cons (x,y) (zip l1’ l2’)
```

Function Definition Packages II

Advantages

- no fancy pretty printer needed
- patterns not restricted
  - non-constructor functions
  - multiple occurences of bound variable
  - arithmetic expressions
  - ...
- conditional equations can represent guards

Function Definition Packages II

Advantages

- no fancy pretty printer needed
- patterns not restricted
  - non-constructor functions
  - multiple occurences of bound variable
  - arithmetic expressions
  - ...
- conditional equations can represent guards

Limitations

- no precedence of rows
  - either add complicated guards
  - or have many equations
- only possible at top-level
- code extraction tricky

Overview

- we provide a powerful, direct representation of case-expressions in HOL 4
- available with HOL 4 in directory `examples/pattern_matches`
- idea very similar to concepts used by function definition packages
- main challange: engineering
  - decent parsing and pretty printing support
  - good automation (about 5000 lines of ML)
- CakeML (`cakeml.org`) has been extended to use new representation
- this representation allows to improve code generation

New Representation I

- a row is a function `r : `$\alpha \to \beta$` option`
- it matches a value `v : `α, if `r v` returns a value
- a case-expression searches for the first row matching
- if no row matches, it returns `PMATCH_INCOMPLETE`

Case Expressions

```
PMATCH v [] := PMATCH_INCOMPLETE := ARB
PMATCH v (r::rs) := case r v of
                      SOME result => result
                    | NONE => PMATCH v rs
```

New Representation II

- a row consists of a
  - a pattern `p` $: \gamma \to \alpha$
  - a guard `g` $: \gamma \to$ `bool`
  - a right-hand-side `r` $: \gamma \to \beta$
- a value `v` $: \alpha$ matches the row, if there setting `x` $: \gamma$ of the variables bound by `p` with `p x = v` and `g x`
- it then evalutes to `r` applied to the variable setting

Row Semantics

```
PMATCH_ROW p g r := λv.
  if (∃x. (p x = v) ∧ g x) then
    SOME (r (@x. (p x = v) ∧ g x))
  else
    NONE
```

Example List Membership

Raw Syntax (for MEM x l)

```
PMATCH l [
  PMATCH_ROW (λ(uv:unit). []) (λuv. T) (λuv. F);
  PMATCH_ROW (λ(y,ys). y::ys) (λ(y,ys). x = y) (λ(y,ys). T);
  PMATCH_ROW (λ(_0,ys). _0::ys) (λ(_0,ys). T) (λ(_0,ys). mem x ys)
]
```

Pretty Printer and Parser Syntax (for MEM x l)

```
CASE l OF [
  ||. [] ~> F;
  || (y,ys). y::ys when (x = y) ~> T;
  || ys. _::ys ~> mem x ys
]
```

Features

- guards
- patterns not restricted to constructors
- full control over variables
  - free variables in pattern
  - multiple occurences of bound variable
- non-injective patterns

Example

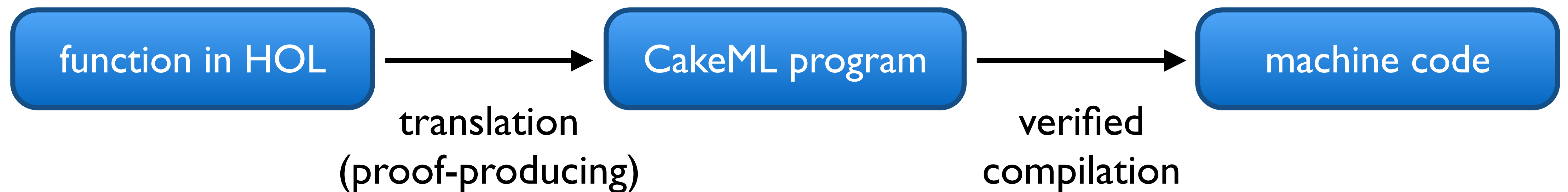
```
my_divmod n c :=
  CASE n OF [
    || (q, r). q * c + r when r < c ~> (q,r)
  ]
```

Tools

- conversions from and to decision trees
- congruence rules for automatic well-foundedness proofs
- (partial) evaluation and simplification tools
- removing by lifting to nearest boolean level
  - i.e. at top-level introducing equations like function definition packages
- powerful, extendable pattern compilation
- removal of redundant rows
- exhaustiveness checks
- removing guards / multiple variable bindings
- ...

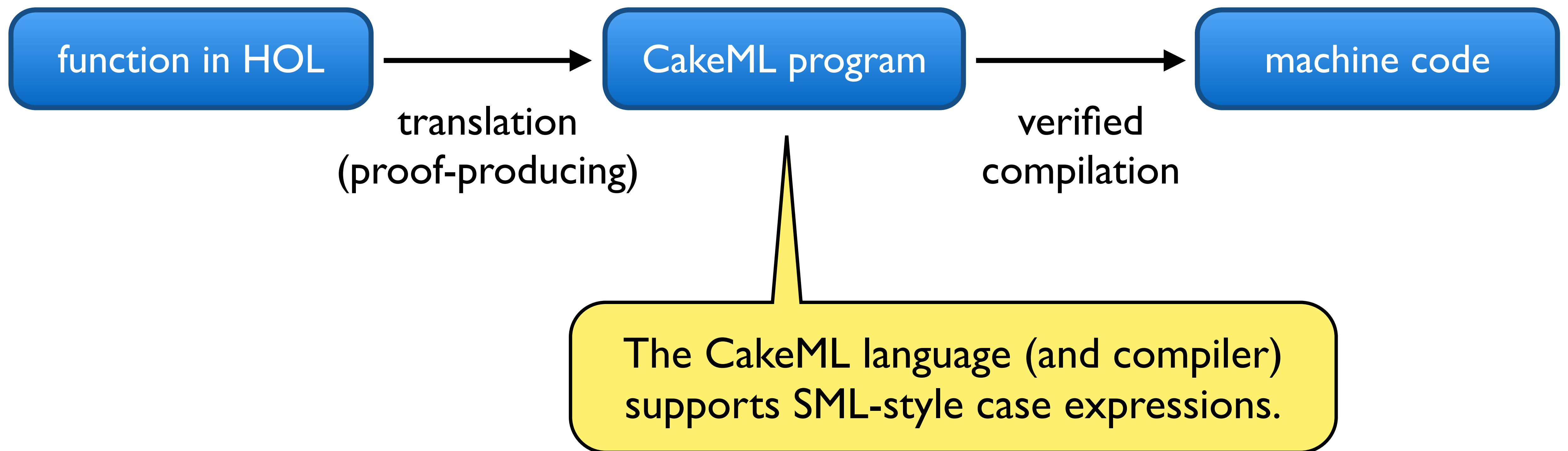
CakeML

The CakeML project provides proof-producing code generation:



CakeML

The CakeML project provides proof-producing code generation:



Code generation

User-input for a term that causes bad code generation:
(assuming a code generator without a fancy pattern compiler)

```
case (a,b) of
  (Red (Red a x b) y c,d) => Red (Black a x b) y (Black c n d)
| (Red a x (Red b y c),d) => Red (Black a x b) y (Black c n d)
| (a,Red (Red b y c) z d) => Red (Black a n b) y (Black c z d)
| (a,Red b y (Red c z d)) => Red (Black a n b) y (Black c z d)
| other => Black a n b
```

Without PMATCH, this turns into (at least) 57 rows in generated SML code.

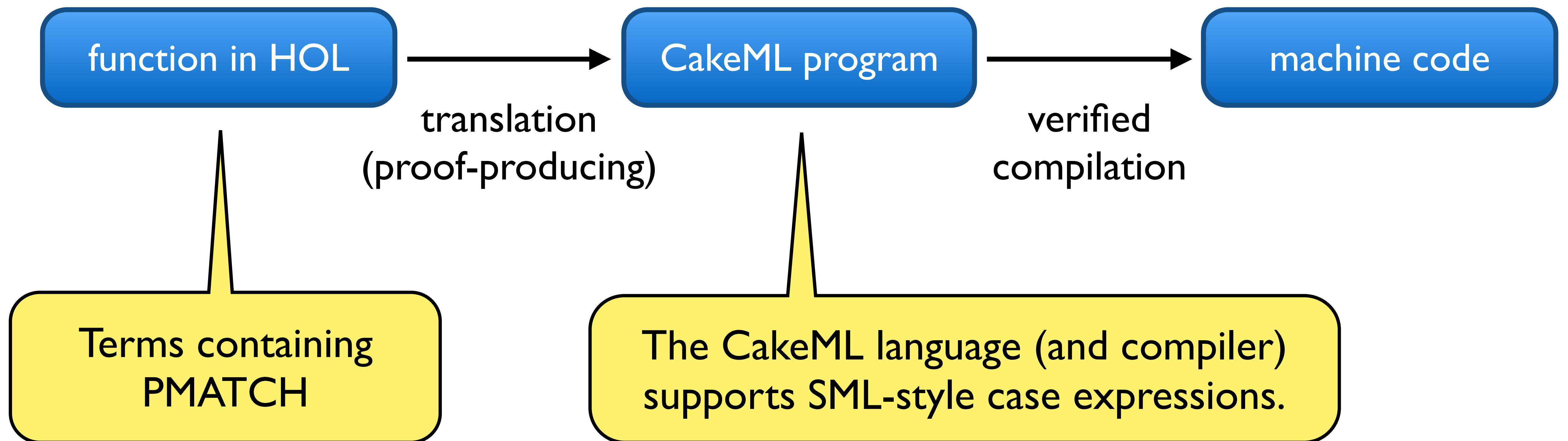
State-of-the-art compilers can still produce optimal code from such suboptimal source code.

With PMATCH, a simple code generator can translate the above expression into 5 rows.

Even naive compilers (e.g. CakeML's) can produce reasonable code from this.

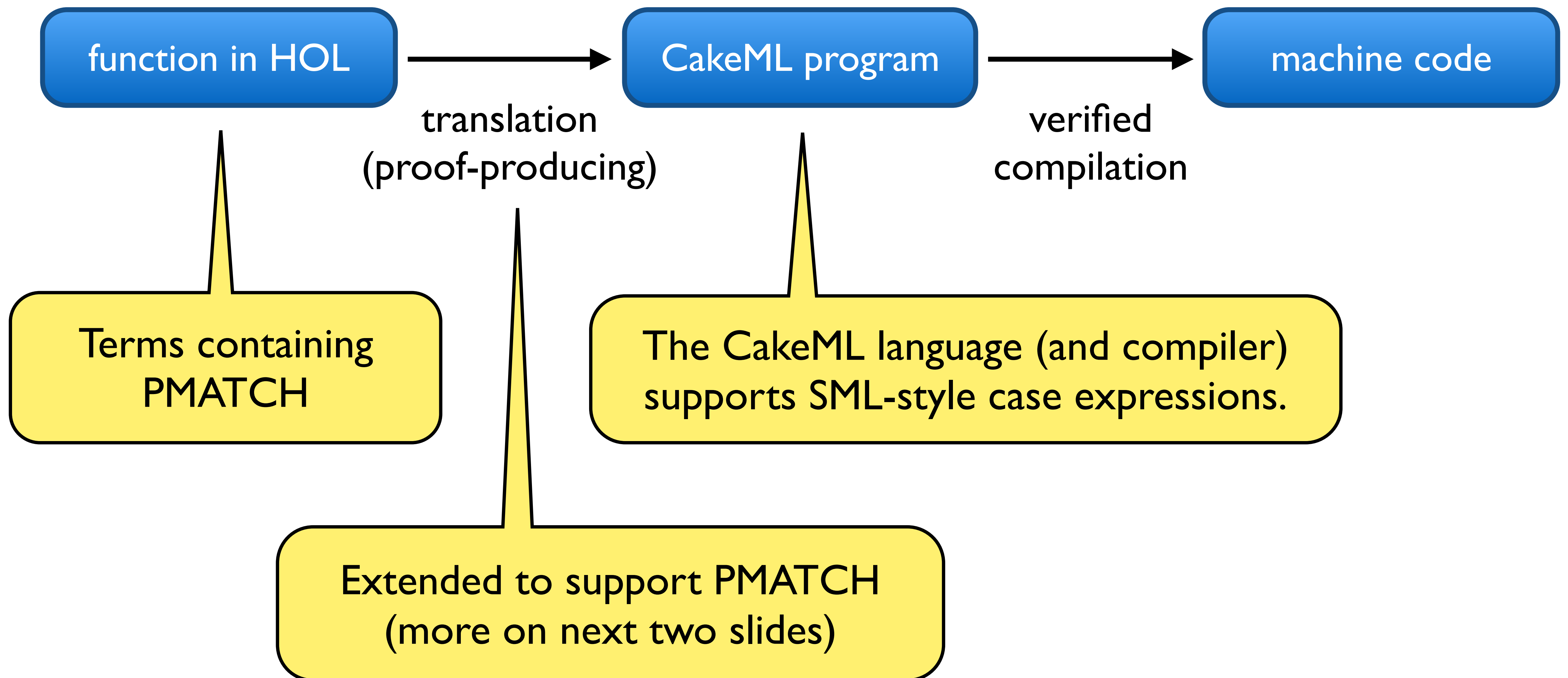
CakeML

The CakeML project provides proof-producing code generation:

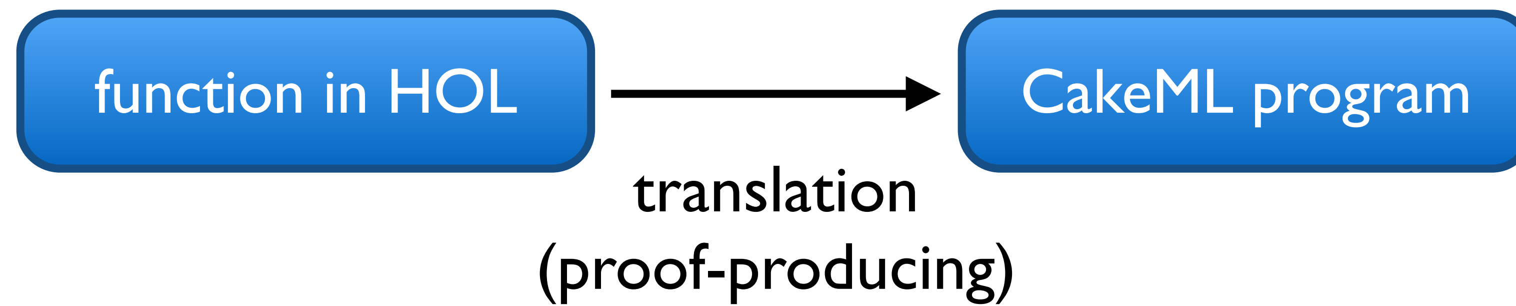


CakeML

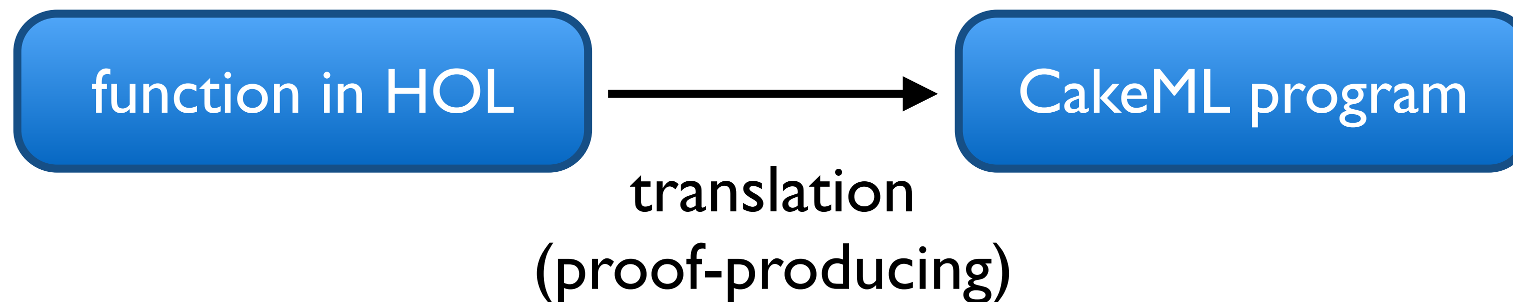
The CakeML project provides proof-producing code generation:



CakeML support for PMATCH



CakeML support for PMATCH

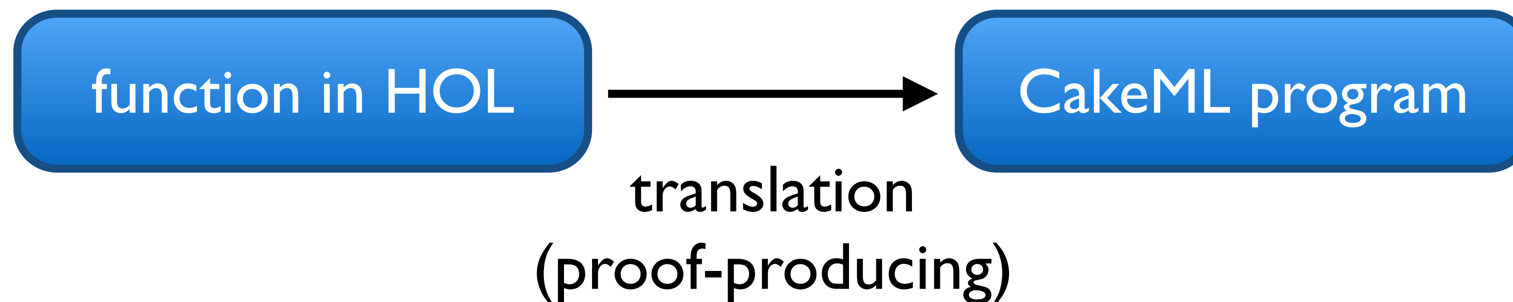


The translation derives theorems of the following form:

Eval cakeml_ast env (relation hol_term)

where Eval is: Eval ast env p = ∃v. (ast,env) ⇓ (Result v) ∧ p v

CakeML support for PMATCH



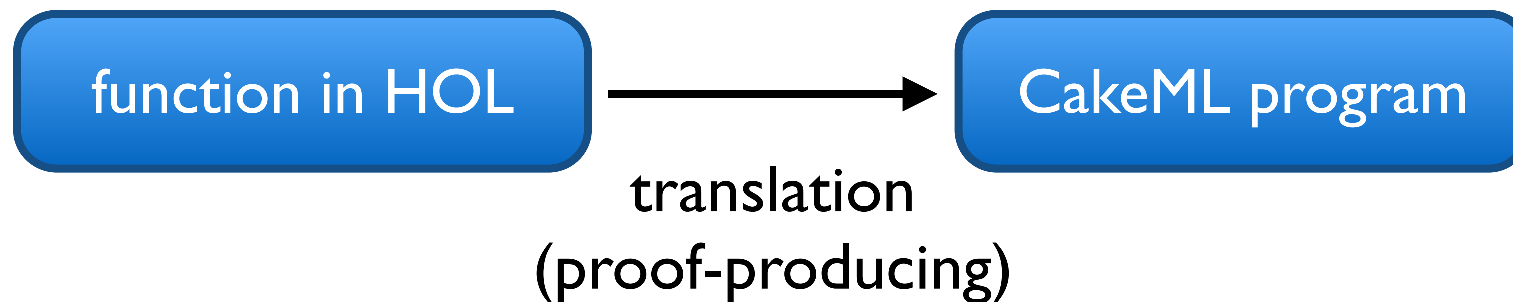
The translation derives theorems of the following form:

Eval cakeml_ast env (relation hol_term)

CakeML program

where Eval is: Eval ast env p = ∃v. (ast,env) ⇓ (Result v) ∧ p v

CakeML support for PMATCH



The translation derives theorems of the following form:

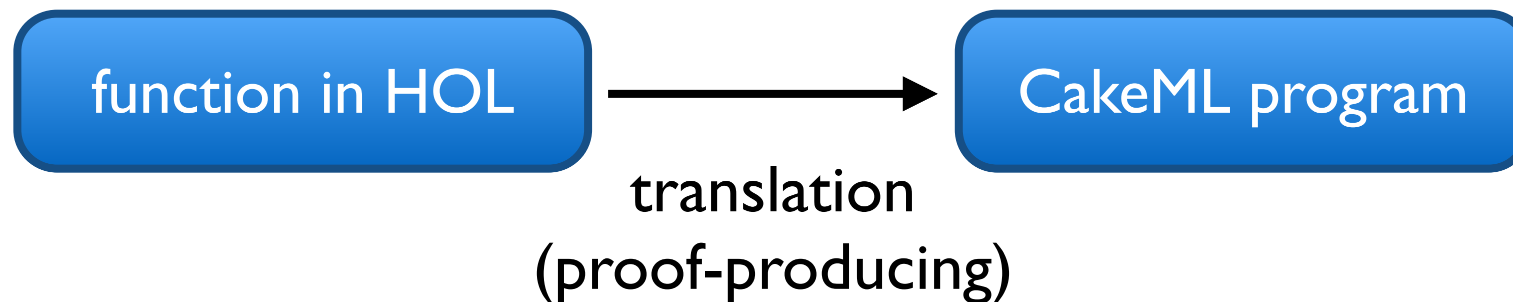
Eval cakeml_ast env (relation hol_term)

CakeML program

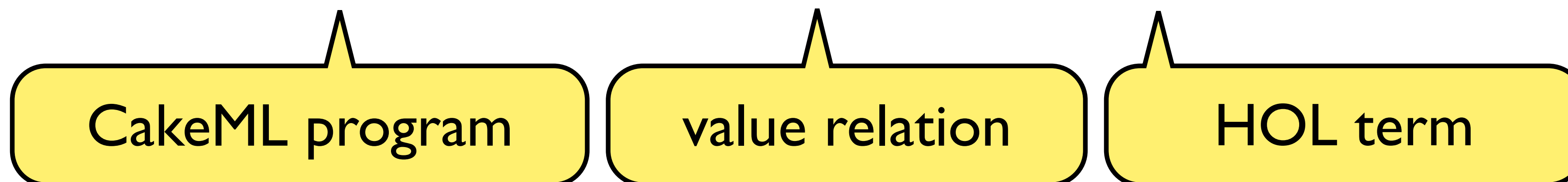
value relation

where Eval is: Eval ast env p = ∃v. (ast,env) ⇓ (Result v) ∧ p v

CakeML support for PMATCH



The translation derives theorems of the following form:



where Eval is: Eval ast env p = ∃v. (ast,env) ⇓ (Result v) ∧ p v

CakeML support for PMATCH

The translation derives theorems of the following form:

Eval cakeml_ast env (relation hol_term)

CakeML program

value relation

HOL term

CakeML support for PMATCH

Implementation:

We supply two new theorems: one for translating PMATCH with no pattern rows, and one for adding a row to a PMATCH.

Custom automation written to apply the above theorems when PMATCH is to be translated.

The translation derives theorems of the following form:

Eval cakeml_ast env (relation hol_term)

CakeML program

value relation

HOL term

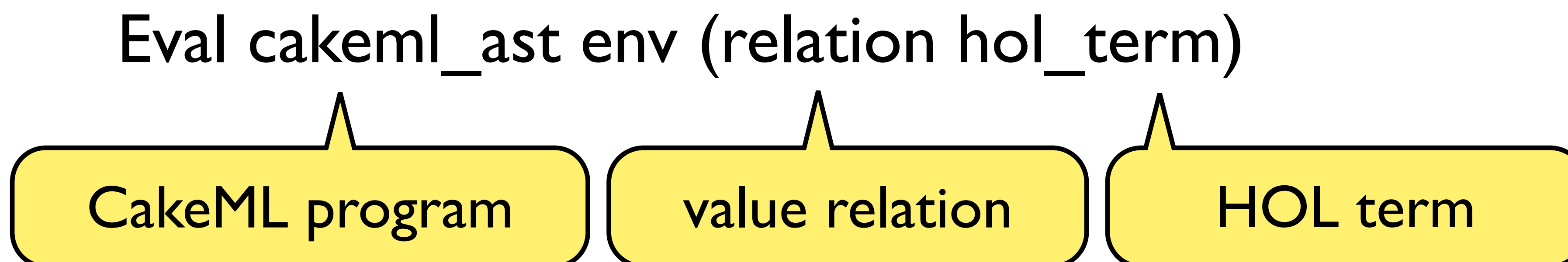
CakeML support for PMATCH

Implementation:

We supply two new theorems: one for translating PMATCH with no pattern rows, and one for adding a row to a PMATCH.

Custom automation written to apply the above theorems when PMATCH is to be translated.

The translation derives theorems of the following form:



Experiments suggest that our implementation can translate all SML-style PMATCH terms of interest.

Partial Evaluation

```
|- CASE (a,Black b' m c') OF [
    || (a,x,b,y,c,d). (Red (Red a x b) y c,d) ~>
         (Red (Black a x b) y (Black c n d));
    || (a,x,b,y,c,d). (Red a x (Red b y c),d) ~>
         (Red (Black a x b) y (Black c n d));
    || (a,b,y,c,z,d). (a,Red (Red b y c) z d) ~>
         (Red (Black a n b) y (Black c z d));
    || (a,b,y,c,z,d). (a,Red b y (Red c z d)) ~>
         (Red (Black a n b) y (Black c z d));
    || other. other ~> (Black a n (Black b' m c'))
  ] =
  CASE a OF [
    || (a,x,b,y,c). Red (Red a x b) y c ~>
         (Red (Black a x b) y (Black c n (Black b' m c')));
    || (a,x,b,y,c). Red a x (Red b y c) ~>
         (Red (Black a x b) y (Black c n (Black b' m c')));
    ||. _ ~> (Black a n (Black b' m c'))
  ]
```

Removing Double Binds

```
|- REMOVE_DUPS l =
   CASE l OF [
     ||. [] ~> [];
     || (x,xs). x::x::xs ~> (REMOVE_DUPS (x::xs));
     || (x,xs). x::xs ~> (x::REMOVE_DUPS xs)
   ]
```

becomes by removing double binds

```
|- REMOVE_DUPS l =
   CASE l OF [
     ||. [] ~> [];
     || (x,xs,x_1). x::x_1::xs when (x_1 = x) ~>
           (REMOVE_DUPS (x::xs));
     || (x,xs). x::xs ~> (x::REMOVE_DUPS xs)
   ]
```

Removing Guards

```
|- REMOVE_DUPS l =
   CASE l OF [
     ||. [] ~> [];
     || (x,xs,x_1). x::x_1::xs when (x_1 = x) ~>
          (REMOVE_DUPS (x::xs));
     || (x,xs). x::xs ~> (x::REMOVE_DUPS xs)
   ]
```

becomes by removing guards

```
|- REMOVE_DUPS l =
   CASE l OF [
     ||. [] ~> [];
     || (x,xs,x_1). x::x_1::xs ~>
          if x_1 = x then REMOVE_DUPS (x::xs)
          else x::REMOVE_DUPS (x_1::xs);
     || (x,xs). x::xs ~> (x::REMOVE_DUPS xs)
   ]
```

Introducing Equations

```
|- (REMOVE_DUPS [] = []) ∧
   (∀x xs. REMOVE_DUPS (x::x::xs) = REMOVE_DUPS (x::xs)) ∧
   (∀x xs.
      (∀xs_1. xs ≠ x::xs_1) ⇒
      (REMOVE_DUPS (x::xs) = x::REMOVE_DUPS xs)) ∧
   (∀l. l ≠ [] ∧ (∀x xs. l ≠ x::xs) ⇒ (REMOVE_DUPS l = ARB))
```

```
|- (REMOVE_DUPS [] = []) ∧
   (∀x xs x_1. REMOVE_DUPS (x::x_1::xs) =
      if x_1 = x then REMOVE_DUPS (x::xs)
      else x::REMOVE_DUPS (x_1::xs)) ∧
   (∀x xs.
      (∀xs_1 x_1. xs ≠ x_1::xs_1) ⇒
      (REMOVE_DUPS (x::xs) = x::REMOVE_DUPS xs)) ∧
   (∀l. l ≠ [] ∧ (∀x xs. l ≠ x::xs) ⇒ (REMOVE_DUPS l = ARB))
```

Exhaustiveness

```
|- ∀l1 l2.
   ZIP_PART l1 l2 =
   CASE (l1,l2) OF [
     || (x,xs,y,ys). (x::xs,y::ys) ~> ((x,y)::ZIP_PART xs ys);
     ||. ([],[]) ~> []
   ]
```

becomes with automatically adding missing cases

```
|- ∀l1 l2.
   ZIP_PART l1 l2 =
   CASE (l1,l2) OF [
     || (x,xs,y,ys). (x::xs,y::ys) ~> ((x,y)::ZIP_PART xs ys);
     ||. ([],[]) ~> [];
     || (v4,v5). ([],v4::v5) ~> ARB;
     || (v2,v3). (v2::v3,[]) ~> ARB
   ]
```

We can prove exhaustiveness of this pattern-match automatically.

Pattern Compilation I

- one reason for blowup of RBT-balancing: 3 constructors
- only 2 cases needed: red or not
- our pattern compilation allows to do that
- resulting decision tree is about 20% of the size of original decision tree

```
val tree_red_CASE_def = Define `
  tree_red_CASE tr f_red f_else =
  tree_CASE tr (f_else Empty) f_red
    (1̑ n t2. f_else (Black t1 n t2))`

val red_cf = mk_constructorFamily (
  make_constructorList false [(``Red``, ["t1", "n", "t2"])],
  ``tree_red_CASE``,
  (* proof of basic properties *) ...);

val _ = pmatch_compile_db_register_constrFam red_cf;
```

Pattern Compilation II

```
|- ∀a n b.
  balance_black a n b = case (a,b) of (v,v_1) =>
    tree_red_CASE v
      (λt1 n_1 t2. tree_red_CASE t1
        (λt1_1 n_2 t2_1. Red (Black t1_1 n_2 t2_1) n_1 (Black t2 n v_1))
        (λx. tree_red_CASE t2
          (λt1_2 n_2_1 t2_1_1.
            Red (Black x n_1 t1_2) n_2_1 (Black t2_1_1 n v_1))
          (λx_1. tree_red_CASE v_1
            (λt1_3 n_2_2 t2_2.
              tree_red_CASE t1_3
                (λt1_1_1 n_3 t2_1_2.
                  Red (Black (Red x n_1 x_1) n t1_1_1) n_3
                    (Black t2_1_2 n_2_2 t2_2))
                (λx_2. tree_red_CASE t2_2
                  (λt1_4 n_3_1 t2_1_3.
                    Red (Black (Red x n_1 x_1) n x_2) n_2_2
                      (Black t1_4 n_3_1 t2_1_3))
                  (λx_3. Black a n b)))
            (λx_2_1. Black a n b))))
        (...)
```

Removing Redundant Rows

```
|- CASE ([],l) OF [
     ||.          ([], [])     ~> 0;
     || (x,xs). ([], x::xs) ~> 1;
     ||.          (_,  _)      ~> 2
   ] =
   CASE l OF [
     ||. []    ~> 0;
     ||. _::_ ~> 1;
     ||. _     ~> 2
   ]
```

```
|- CASE l OF [
     ||. []    ~> 0;
     ||. _::_ ~> 1;
     ||. _    ~> 2
   ] =
   CASE l OF [
     ||. []    ~> 0;
     ||. _::_ ~> 1
   ]
```

Conclusion

Summary

- we present a new representation for pattern matching in classical higher order logic
- implemented in HOL 4 (`http://hol-theorem-prover.org`)
- integration with CakeML (`http://cakeml.org`)
- tool support

Future Work

- more automation
- improved support for arithmetic expressions
- implement flattening
- ...

FireEye in Dresden

FireEye

- real-time threat protection against next generation cyber attacks
- customers: enterprises and governments worldwide

Formal Methods Team in Dresden, Germany

- hardening of trusted computing base
- verification of microhypervisor
- tools: Coq, static program analysis tools, ...
- currently 8 formal methods engineers
- expected to grow