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

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

[1] Independent Scholar
[2] CSE Department, Chalmers University of Technology
[3] Computer Laboratory, University of Cambridge

Abstract. Pattern matching is ubiquitous in functional programming and also very useful for definitions in higher-order logic. However, it is not directly supported by higher-order logic. Therefore, the parsers of theorem provers like HOL4 and Isabelle/HOL contain a pattern-compilation algorithm. Internally, decision trees based on case constants are used. For non-trivial case expressions, there is a big discrepancy between the user's view and the internal representation.
This paper presents a new general-purpose representation for case expressions that mirrors the input syntax in the internal representation closely. Because of this close connection, the new representation is more intuitive and often much more compact. Complicated parsers and pretty printers are no longer required. Proofs can more closely follow the user's intentions, and code generators can produce better code. Moreover, the new representation is more general than the currently used representation, supporting guards, patterns with multiple occurrences of the same bound variable, unbound variables, arithmetic expressions in patterns, and more. This work has been implemented in the HOL4 theorem prover and integrated into CakeML's proof-producing code generator.

1 Introduction

Pattern matching is ubiquitous in functional programming and in definitions within interactive theorem provers. Through the use of case expressions (a. k. a. match expressions), pattern matching allows for concise and easy to read definitions. For provers based on higher-order logic (HOL), case expressions are not natively part of the logic. To use them, they are processed outside the logic.

The term parsers of all major HOL systems, in particular HOL4 [11], Isabelle/HOL [13], HOL Light [2], and ProofPower, contain an implementation of pattern compilation. This pattern compilation turns case expressions into decision trees consisting of nested applications of *case constants* [10, 1], which are defined for each algebraic datatype. A complicated pretty printer prints the resulting decision trees as case expressions. The decision trees can be evaluated efficiently using basic rewriting techniques. They represent complete case splits; no cases overlap and no case is missing. However, the pattern-compilation implementation in the parser and the complicated pretty printer are a cause for

concern in an LCF-style prover. In addition to the inference kernel, these components need to be trusted to some degree. Another disadvantage is that pattern compilation often leads to a huge blow-up in term size. Code extracted from the internal representation is often both hard to read and slow. Moreover, the structure intended by the user is obfuscated by pattern compilation and proofs have to follow the artificial, often very complicated structure of the internal representation.

Our contribution is a new representation for case expressions that avoids these problems. Our representation is able to mirror the user's input syntax faithfully. Therefore parsing and pretty printing are straightforward and the blow-up in term size is avoided. Compared to the decision tree representation, extracted code is of better quality and proofs need to consider fewer cases. Moreover, the new representation supports more advanced pattern-matching features. For example, it supports patterns that include guards, binding a variable multiple times, arithmetic expressions as well as a concept similar to simple view patterns [12].

Related to our work are function definitions packages like the ones implemented in Isabelle/HOL [3], HOL4 [10] and HOL Light [2]. At the top level, they are able to avoid case expressions in the logic completely by using a set of (conditional) equations for function specifications. For example, the length function on lists (`len l := case l of [] => 0 | x :: xs => (len xs + 1)`) is described by the equations `len [] = 0` and $\forall$ `x xs. len (x :: xs) = length xs + 1`. Since arbitrary equations are used instead of case constants, these packages provide all the features offered by our approach like guards, arithmetic expressions or binding a variable multiple times. However, unlike our approach, these packages do not represent case expressions in the logic at all. Instead a set of (conditional) equations is returned. The most striking difference in semantics, compared to case expressions, is that there is no precedence on these equations. This means that overlapping patterns are problematic. One option, implemented by e. g. HOL4 and Isabelle/HOL, is to use pattern compilation to transform the input patterns to a set of non-overlapping patterns. This leads however to the same issues and restrictions described for pattern compilation above. An alternative implemented in e. g. Isabelle/HOL [3] and HOL Light is to prove that overlapping input patterns result in the same value. Enforcing non-overlapping patterns often leads to either very complicated guards or similar blow-ups in the number of cases as compilation to a decision tree.

The work[4] presented here has been implemented in HOL4 [11]. Our representation is very similar to concepts used internally by the HOL Light function definition package[5]. However, we expose our definitions to the user, whereas the function definition package uses it only internally. Since our representation uses Hilbert's choice operator and existential quantification, naive usage is likely to cause problems. In contrast to decision trees, simple rewrite techniques

[4] The code can be found under: `https://github.com/HOL-Theorem-Prover/HOL/examples/pattern_matches`

[5] compare function `CASEWISE` in `define.ml`

are not sufficient. Therefore, a significant part of this work consists in providing specialised tools for dealing with our case expressions. We provide parsers and pretty printers[6], and evaluation and simplification tools[7]. There is support for turning function definitions using our case expression into equations similar to the ones produced by function definition packages[8]. Furthermore, there are tools for converting between our case expressions and ones represented by decision trees. These tools range from untrustworthy parsers and pretty printers to a pattern-compilation algorithm implemented inside the logic. This allows the user to choose the right representation for the task at hand. Moreover, our implementation of a pattern-compilation algorithm[9] lets us leverage some of the nice properties of decision trees. It is used to check the exhaustiveness of our case expressions[10] as well as pruning patterns that are made redundant by the combination of multiple other patterns[11].

Applying code generation to the new representation, we can produce higher-quality code. We demonstrate the quality improvement with a few examples, and describe how proof-producing code generation (for CakeML [5]) can be extended to generate code that mirrors the exact structure of the HOL term and the concrete syntax provided by the user (Section 6).

2 Shortcomings of Decision Tree Representation

All major HOL systems, in particular HOL4 [11], Isabelle/HOL [13], ProofPower and HOL Light [2], use decision trees based on case constants for representing case expressions. In this section, we try to illustrate this method and its shortcomings using a number of examples. If not explicitly stated otherwise, we use HOL4 as the example prover. The implementation in the other systems is very similar, though.

Basic Example The classical representation of case expressions is based on case constants [10, 1]. HOL4's datatype definition package produces for each algebraic datatype definition a case constant that can perform a top-level pattern match on the constructors of that datatype. HOL4's list datatype (with constructors `Nil` and `Cons`) for example has an associated case constant `list_case`, which is characterised by the following equations

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

[6] see `patternMatchesSyntax`
[7] see e. g. `PMATCH_SIMP_ss` or `PMATCH_REMOVE_GUARDS_ss`
[8] see `PMATCH_LIFT_BOOL_ss`
[9] see e. g. `PMATCH_CASE_SPLIT_ss`
[10] see `COMPUTE_REDUNDANT_ROWS_INFO_OF_PMATCH` and `PMATCH_IS_EXHAUSTIVE_CONSEQ_CONV`
[11] see `PMATCH_REMOVE_REDUNDANT_ss`

The datatype definition package also informs the parser and the pretty printer about such case constants. The term `list_case x 5 (λy ys. y + 3)` is pretty printed as

```
case x of Nil => 5 | Cons y ys => y + 3
```

This representation is also accepted by the parser and parsed to the internal `list_case` representation.

Pattern Match Heuristics For such simple case expressions, the classical approach works perfectly. However, problems start to appear if the case expressions become even slightly more complex. As an example, let's define `b ∨ (a ∧ c)` via a case expression:

```
case (a, b, c) of
  (_, T, _) => T
| (T, _, T) => T
| (_, _, _) => F
```

For this example, several applications of the case constant for type `bool`, better known as if-then-else, need to be nested. Classically, HOL4 performs the splits from left to right, i. e. in the order `a`, `b`, `c`. This leads to (slightly simplified) the following internal representation and corresponding pretty printed output:

```
if a then (                          case (a,b,c) of
  if b then (                          (T,T,T) => T
    if c then T else T               | (T,T,F) => T
  ) else (                           | (T,F,T) => T
    if c then T else F               | (T,F,F) => F
  )                                  | (F,T,_) => T
) else (                             | (F,F,_) => F
  if b then T else F
)
```

Even for this simple example, one can observe a severe blow-up. The number of rows has doubled. One might also notice that the clear structure of the input is lost. Other systems using the classical approach might behave slightly differently in detail but in principle suffer from the same issues. Isabelle/HOL for example also performs pattern compilation always from left to right, but is slightly better at avoiding unnecessary splits.

To combat some of these issues, we extended HOL4's pattern compilation algorithm in early 2013 with state-of-the-art pattern match heuristics[12] presented by Luc Maranget [6]. These heuristics often choose a decent ordering of case splits. Moreover, we also implemented – similar to Isabelle/HOL – the avoidance of some unnecessary splits. With these extensions, the example is compiled to:

[12] see `/src/1/PmatchHeuristics.sig` in the HOL4 sources

```
if b then T else (
  if c then
    (if a then T else F)
  else F
)
```

```
case (a,b,c) of
  (v,T,v3) => T
| (T,F,T) => T
| (F,F,T) => F
| (v,F,F) => F)
```

However, this improvement has a price. The pattern compilation algorithm in the parser became slightly more complicated and the results are even harder to predict.

Real World Example The following case expression is taken from Okasaki's book on Functional Datastructures [8]. It is used in a function for balancing red-black trees, which are represented using the constructors `Empty`, `Red` and `Black`.

```
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
```

Parsing this term with the classical pattern-compilation settings in HOL4 results in a huge term that pretty prints with 121 cases! Even with our state of the art pattern-match heuristics a term with 57 cases is produced. In this term the right-hand sides of the rows are duplicated a lot. The right-hand side of the last row (`Black a n b`) alone appears 36 times in the resulting term.

This blowup is intrinsic to the classical approach. Our pattern match heuristics are pretty good at finding a good order in which to perform case splits. For this example, they find an optimal order. There is no term based on case constants that gets away with fewer than 57 cases. However, clever pretty printers might present a smaller looking case expression (see Sec. 6.2).

Also notice that this example relies heavily on the precedence of earlier rows over later ones in the case expressions. If we use – as required by the equations produced by function definition packages – non-overlapping patterns, we get a similar blow-up as when compiling to a decision tree.

3 New Approach

In the previous section we presented the classical approach used currently by all major HOL systems. We showed that the internal representation for this approach often differs significantly from the input. There is often a huge blow-up in size. This leads to less readable and more importantly less efficient code as well as lengthier and more complicated proofs. In the following we will present our new approach and how it overcomes these issues.

3.1 Definition

A row of a case expression consists of a pattern p, a guard g and a right hand side r. We need to model the variables bound by the pattern. Therefore, p, g and r are functions that get the value of the bound variables as their argument. They are of type $\gamma \to \alpha$, $\gamma \to$ `bool` and $\gamma \to \beta$, respectively. The type of the bound variables γ changes for each row of a case expression. In order to easily fit into HOL's type system, we apply these components to a function `PMATCH_ROW` which maps them to their intended 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
```

For injective patterns, i. e. the ones normally used, this function models perfectly the standard semantics of case expressions (as e. g. defined in [9]).

It remains to extend this definition of the semantics of a single row to case expressions. Our case expressions try to find the first row that matches a given input value. If no row matches, a functional language like ML would raise a match-exception. Here, we decided not to model this error case explicitly and use `ARB` (a. k. a. `Undef`) instead. This constant is used to denote a fixed, but unknown value of an arbitrary type in HOL. Formally, this means that our case-expression constant `PMATCH` is defined recursively by the following equations:

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

3.2 Concrete syntax and bound variables

The definitions above let us write case expressions with guards. The body of the list-membership function `mem x l`, can for example be written as:

```
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)
]
```

This syntax closely mirrors the user's intention inside HOL. However, it is rather lengthy and hard to read and write. Therefore, we implemented a pretty printer and a parser for such expressions, enabling the following syntax:

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

For rows, we write bound variables only once instead of repeating them for pattern, guard and right-hand side. Moreover, there is support for wildcard syntax. Finally, we provide the `CASE . OF .` notation for `PMATCH` and reuse standard list syntax for the list of rows. Thus, in contrast to the classical approach the parser and pretty printer are straightforward.

3.3 Advanced Features

Our representation provides more expressive case expressions than the classical approach. We don't enforce syntactic restrictions like using only datatype constructors or binding variables only once in a pattern. Fine control over the bound variables in a pattern allows inclusion of free variables, which act like constants. Finally, there are guards.

These features can be used to very succinctly and clearly express complicated definitions that could not be handled with the classical approach. Division with remainder can for example be defined by:

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

The new case expressions are not even limited to injective patterns. They can for example be used to perform case splits on sets.

```
CASE n OF [
  ||. {} ~> NONE;
  ||(x, s). x INSERT s when ~(x IN s) ~> SOME (x, s)
]
```

3.4 Congruence rules

Case expressions are frequently used to define recursive functions. In order to prove the well-foundedness of recursive definitions, HOL systems use a termination condition extraction mechanism, which is configured via congruence rules[13]. We provide such congruence rules for HOL4.

```
∀v v' rows rows'. (
    (v = v') ∧ (r v' = r' v') ∧
    (PMATCH v' rows = PMATCH v' rows')) ⟹
  (PMATCH v (r :: rows) =
   PMATCH v' (r' :: rows'))
```

```
∀p p' g g' r r' v v'. (
    (p = p') ∧ (v = v') ∧
    (∀x. (v = (p x)) ⇒ (g x = g' x)) ∧
    (∀x. (v = (p x) ∧ g x) ⇒
         (r x = r' x))) ⟹
  (PMATCH_ROW p g r v =
   PMATCH_ROW p' g' r' v')
```

These rules lead to very similar termination conditions as produced by the congruence rules for the classical decision trees. Therefore they work well with existing automatic well-foundedness checkers.

[13] see e. g. HOL4's *Description Manual* Section 4.5.2 or Isabelle/HOL's manual *Defining Recursive Functions in Isabelle/HOL* Section 10.1

Remark The observant reader might wonder, why `PMATCH_ROW` uses 3 functions as arguments instead of just one function returning a triple. This would simplify parsing and pretty printing, but cause severe problems for recursive definitions using `PMATCH`. HOL4's machinery would not be able to use the resulting congruence rules, since their application would require higher-order matching. We expect that Isabelle/HOL would be fine with rules that require higher-order matching, but have not tested this.

4 Evaluation and Simplification

If one naively expands the definition of `PMATCH`, one easily ends up with huge terms containing Hilbert's choice operator and existential quantifiers. To avoid this, we developed specialised tools for HOL4 to evaluate and simplify our case expressions. As a running example consider

```
CASE (SOME x, ys) OF [
  || y. (NONE, y::_) ~> y;
  || (x',y). (x', y::_) ~> (THE x')+y;
  || x. (SOME x, _) ~> x;
  ||. (_, _) ~> 0
]
```

Pruning Rows For each row of a `PMATCH` expression, we check, whether its pattern and guard match the input value. If we can show that a row does not match, it can be dropped. If it matches, all following rows can be dropped. We don't need a decision for each row. If it is unknown whether a row matches, the row can just be kept. Finally, if the first remaining row matches, we can evaluate the whole case expression. Applying this method to the running example results in

```
CASE (SOME x, ys) OF [
  || (x',y). (x', y::_) ~> (THE x')+y;
  || x. (SOME x, _) ~> x
]
```

Partial Evaluation In order to partially evaluate `PMATCH` expressions, we try to split the involved patterns into more primitive ones. For this we split tuples and group corresponding tuple elements in multiple rows into so-called *columns*. In the running example the first column contains the input `SOME x` and the patterns `x'` (where `x'` is bound) and `SOME x` (where `x` is bound). The second column contains `ys` as input and `y::_` and `_` as patterns.

If the input value of a column consists of the application of an injective function, e. g. a datatype constructor, and all patterns of this column contain either applications of the same injective function or bound variables, this column can be partially evaluated. We can remove the function application and just keep the arguments of the function in new columns. For rows containing bound variables,

we substitute that variable with the input value and fill the new columns with fresh bound variables. In our running example, we can simplify the first column with this method. This partial evaluation leads to:

```
CASE (x, ys) OF [
  || y. (x'', y::_) ~> x'' + y;
  || x. (x, _) ~> x
]
```

Now, the first column consists of only variables and can be removed:

```
CASE ys OF [
  || y. y::_ ~> x + y;
  ||. _ ~> x
]
```

The semantic justification for this partial evaluation is straightforward. Essentially we are employing the same rules used by classical pattern compilation algorithms (compare e. g. Chap. 5.2 in [9]). However, implementing it smoothly in HOL4 is a bit fiddly. It involves searching for a suitable column to simplify and instantiating general theorems in non-straightforward ways.

Integration with Simplifier Pruning rows and partial evaluation are the most important conversions for `PMATCH`-based case expressions. Other useful conversions use simple syntactic checks to remove redundant or subsumed rows. Additionally, we implemented conversions that do low-level maintenance work on the datastructure. For example, there are conversions to ensure that no unused bound variables are present, that the variable names in the pattern, guard and right-hand side of each row coincide and that each row has the same number of columns. All these conversions are combined in a single conversion called `PMATCH_SIMP_CONV`. We also provide integration with the simplifier in form of a simpset-fragment called `PMATCH_SIMP_ss`.

The presented conversions might look straightforward. However, the implementation is surprisingly fiddly. For example, one needs to be careful about not accidentally destroying the names of bound variables. The implementation of `PMATCH_SIMP_CONV` consists of about 1100 lines of ML.

5 Pattern Compilation

We provide several methods based on existing pretty printing and parsing techniques to translate between case expressions represented as decision trees and via `PMATCH`. The equivalence of their results can be proved automatically via repeated case splits and evaluation.

More interestingly, we implemented a highly flexible pattern-compilation algorithm for our new representation. As stated above, our simplification tools for `PMATCH` are inspired by pattern compilation. Thus, the remainder is simple: we provide some heuristics to compute a case-split theorem. Pattern compilation consists of choosing a case split, simplifying the result and iterating.

5.1 Constructor Families

We implemented the heuristics for finding case splits in form of a library called `constrFamiliesLib`. This library maintains lists of ML functions that construct case splits. An example of such a function is a literal-case function that performs a case distinction based on nested applications of if-then-else when a column consists only of bound variables and constants. However, the main source of case splits found by the library are *constructor families*.

A constructor family is a list of functions (constructors) together with a case constant and a flag indicating whether the list of constructors is exhaustive. Moreover, it contains theorems stating that these constructors have the desired properties, i. e. they state that all the constructors are injective and pairwise distinct and that the case constant performs a case split for the given list of constructors. If a column of the input `PMATCH` expression contains only bound variables and applications of constructor functions of a certain constructor family, a case-split theorem based on the case constant of that constructor family is returned.

`constrFamiliesLib` accesses HOL4's TypeBase database and therefore automatically contains a constructor family for each algebraic datatype. This default constructor family uses the classical datatype constructors and case constant. One can easily define additional constructor families and use them for pattern compilation as well as simplifying `PMATCH` expressions. These families can use constructor functions that are not classical datatype constructors. Such constructor families provide a different view on the datatype. They lead to a feature similar to the original views in Haskell [12]. This is perhaps best illustrated by a few examples.

List Example One can for example declare `[]` and `SNOC` (appending an element at the end of a list) together with `list_REVCASE` as a constructor family for lists, where `list_REVCASE` is defined by

```
list_REVCASE l c_nil c_snoc =
  if l = [] then c_nil else (c_snoc (LAST l) (BUTLAST l))
```

With this declaration in place, we get `list_REVCASE l 0 (λ x xs. x)` automatically from compiling

```
CASE l OF [
  ||. [] ~> 0;
  || (x, xs). SNOC x xs ~> x
]
```

5.2 Exhaustiveness Check / Redundancy Elimination

The case-split heuristics of our pattern compilation algorithm can be used to compute for a given `PMATCH` expression an exhaustive list of patterns with the following property: a pattern in the list is either subsumed by a pattern of the

original PMATCH expression or does not overlap with it. There are no partial overlaps. Moreover, subsumption can be checked easily via first-order matching.

We can use such an exhaustive list of patterns to implement an exhaustiveness check[14]. We prune all patterns from the list that are subsumed by a pattern and guard in the original PMATCH expression. Pruning the list with respect to a pattern p and guard g, consists of adding the negation of g to all patterns in the list that are matched by p. Then we remove patterns whose guard became false. If after pruning with all original patterns and guards, the resulting list is empty, the original pattern match is exhaustive. Otherwise, we computed a list of patterns and guards that don't overlap with the original patterns and when added to the original case expression make it exhaustive.

The same algorithm also leads to a powerful redundancy-detection algorithm[15]. To check whether a row is redundant, we prune the exhaustive list of patterns with the patterns and guards of the rows above it. Then we check whether any pattern in the remaining list overlaps with the pattern of the row in question.

6 Improved code generation

In this section we turn our attention to code generation. It has become increasingly common to generate code from function definitions in theorem provers. Code generators for HOL operate by traversing the internal structure of HOL terms and producing corresponding code in a suitable functional programming language, e. g. SML, OCaml or Scala.

As discussed in Section 2, the classical per-datatype case constants can produce harmful duplication and a significant blow-up in the size of the HOL terms. Code generators that walk the internal structure of the terms are likely not to realise that there is duplication in various subterms and therefore to produce code that is unnecessarily verbose and somewhat unexpected considering what the user gave as input in their definition.

Our PMATCH-based case expressions avoid accidental duplication and instead very carefully represent the user's desired format of case expressions in the logic. As a result, even naive term-traversing code generators can produce high-quality code from HOL when PMATCH-based case expressions are used.

In what follows, we first illustrate the quality difference between code generated from the classical approach versus from PMATCH-based case expressions. Then, we will explain how our proof-producing code generator for CakeML has been extended to handle PMATCH. Typically, code generators do not provide any formal guarantee that the semantics of the generated code matches the semantics of the HOL definitions given as input, but code generation for CakeML is exceptional in that it does produce proofs. We have found that PMATCH is beneficial not only to the generated code itself but also when producing proofs about the semantics of the generated code.

[14] see PMATCH_IS_EXHAUSTIVE_CONSEQ_CONV
[15] see PMATCH_REMOVE_REDUNDANT_ss

6.1 The quality of generated code

Simple code generators, which traverse the syntax of the HOL terms, produce code that is very similar to the internal representation. Take for instance a variation, using a catch-all pattern, of the basic example in Section 2. When compiled classically, this case expression results in a term that repeats the `5` on the right-hand side. The generated code also repeats the `5`. We show the user input on the left and the output of code generation[16] on the right.

```
case x of
  Cons y Nil => y + 3
| _ => 5
```

```
case v5 of
  Nil => 5
| Cons v4 v3 => case v3 of
                  [] => v4 + 3
                | Cons v2 v1 => 5
```

If we instead input the example above as a `PMATCH`-based case expression, we retain the intended structure: the result does not repeat the `5` and the `Cons y Nil` row stays on top. It is easy for a code generator to follow the structure of a `PMATCH` term, so the generated code reflects the user's input.

```
CASE x OF [
  || y. Cons y Nil ~> y + 3 ;
  ||. _ ~> 5
]
```

```
case v2 of
  Cons v1 Nil => v1 + 3
| _ => 5
```

In this simple example, the duplication of the `5` is not too bad. However, for more serious examples like the red-black tree balancing function (in Section 2) the difference in code quality is significant. The code generated from the classical version is 90 lines long and unreadable, while the code generated from the `PMATCH`-based case expression is almost identical to the input expression, i. e. readable, unsurprising and only 8 lines long.

The red-black tree example is not the only source of motivation for producing better code. Our formalisation of the HOL Light kernel, which we have proved sound [4], contains several functions with tricky case expressions. For the HOL Light kernel, we want to carry the soundness proof over to the generated implementation, so it is important that the code generator can also produce proofs about the semantics of its output.

The helper function `raconv` (used in deciding alpha-convertibility) has the most complex case expression in the HOL Light kernel:

```
raconv env tm1 tm2 =
  case (tm1,tm2) of
    (Var _ _, Var _ _) => alphavars env tm1 tm2
  | (Const _ _, Const _ _) => (tm1 = tm2)
  | (Comb s1 t1, Comb s2 t2) => raconv env s1 s2 ∧ raconv env t1 t2
  | (Abs v1 t1, Abs v2 t2) =>
     (case (v1,v2) of
        (Var n1 ty1, Var n2 ty2) => (ty1 = ty2) ∧
```

[16] Our code generator renames variables. Here `x` has become `v5`, for example.

```
                                    raconv ((v1,v2)::env) t1 t2
      | _ => F)
  | _ => F
```

For this and other examples, a significant case explosion happens when parsed using the classical approach. The generated code is verbose, and the performance of our verified CakeML implementation [4] suffers as a result. By using `PMATCH`-based case expressions as input to the code generator, we retain the original structure and avoid the explosion.

6.2 Why good input case expressions matter

For generating high-quality code, there is an alternative to rephrasing the input: post-processing the generated code. Such post-processing is (in a simple form) implemented as part of the case-expression pretty printers and the code-generation facilities of all major HOL systems. For translating decision trees into `PMATCH` expressions, (see Sec. 5) we implemented a similar, but more powerful post-processor, which combines rows by reordering them and introducing wildcards.

As discussed in Section 2, the 5 input cases of the red-black tree example produce 57 cases when printed naively in HOL4. After re-sorting and collapsing rows, our post-processor reduces this to 8 cases. Isabelle/HOL's pretty printer and code generator produce 41 cases. What's worse, these figures depend on the exact form of the internal decision tree. For another valid decision tree our post-processor produced e. g. 25 cases. So, post-processing can improve the result, but the results are still significantly worse than good user input.

For CakeML, there is the added difficulty that we need to *verify* the post-processing optimisation phase. Formally verifying optimisations over a language which includes closure values is very time consuming, as we have found when working on the CakeML compiler [5]. The reason is that optimisations alter the code, and, through closure values, code can appear in the values of the language. As a result, every optimisation requires a longwinded value-code relation for its correctness theorem.

Reasoning about and optimising `PMATCH`-based case expressions is much simpler. Moreover, the `PMATCH`-based approach allows manual fine-tuning of the exact form of the case expression in the logic, before the automation for code generation takes over. In general this leads to better results.

6.3 Proof-producing code generation for CakeML

It is straightforward to write a code generator that walks a `PMATCH` term and produces a corresponding case expression in a functional programming language like CakeML. For CakeML, we additionally need to derive a (certificate) theorem which shows that the semantics of the generated CakeML code matches the semantics of the input `PMATCH` term. In this section, we explain how the proof-producing code generator of Myreen and Owens [7] has been extended to handle `PMATCH`-based case expressions.

The proof-producing code generator has been described previously [7]; due to space restrictions, we will not present a detailed description here. Since the approach is compositional, it is sufficient for the purposes of this paper to focus on the form of the HOL theorems that are produced during code generation. These theorems relate generated code (deep embeddings) to input terms (shallow embeddings) via the CakeML semantics. They are of the the following form.

```
assumptions ⟹ Eval deep_embedding env (inv shallow_embedding)
```

Here `Eval` is an interface to the CakeML semantics, and the `env` argument is the semantic environment. The assumptions are used to collect constraints on the environment. The refinement invariant `inv` describes how a HOL4 value is implemented by a CakeML value. For example, for lists of Booleans, the appropriate refinement invariant would relate the HOL value `Cons F Nil` to the value `Conv "Cons" [Litv (Bool F); Conv "Nil" []]` in the semantics of CakeML.

The code-generation algorithm traverses a given shallow embedding bottom-up. To each subterm `se`, it applies a theorem of the form `... ⟹ Eval ... env (inv se)`, where `inv` is the refinement invariant appropriate for the type of `se`. Assumptions that relate shallow and deep embeddings are discharged via recursive calls. Other assumptions are either collected or discharged directly. The by-product of this forward proof is a deep embedding constructed in the first argument of `Eval`.

In order to support `PMATCH`, we need to provide theorems of the following form to this algorithm:

```
... ⟹ Eval (...) env (inv (PMATCH xv rows))
```

For an empty set of rows, the CakeML semantics of case expressions raises a `Bind` exception, whereas `PMATCH` results in `PMATCH_INCOMPLETE`. There is no connection between these two outcomes. Therefore, the following theorem intentionally uses the assumption false (`F`) to mark that one should never end up in this case.

```
F ⟹ Eval env (Mat x []) (b (PMATCH xv []))
```

The case of non-empty pattern lists is more interesting. The theorem is long and complicated, so we explain its parts in turn. First, let us look at the conclusion, i. e. lines 11 and 12 below. The conclusion allows us to add a pattern row, `PMATCH_ROW`, to the shallowly embedded `PMATCH` term and, at the same time, a row is added to the deep embedding: `Mat x ((p,e)::ys)`.

```
1   ALL_DISTINCT (pat_bindings p []) ∧
2   (∀v1 v2. (pat v1 = pat v2) ⟹ v1 = v2) ∧
3   Eval env x (a xv) ∧
4   (p1 xv ⟹ Eval env (Mat x ys) (b (PMATCH xv yrs))) ∧
5   EvalPatRel env a p pat ∧
6   (∀env2 vars.
7       EvalPatBind env a p pat vars env2 ∧ p2 vars ⟹
8       Eval env2 e (b (res vars))) ∧
9   (∀vars. (pat vars = xv) ⟹ p2 vars) ∧
```

```
10    ((∀vars. ¬(pat vars = xv)) ⟹ p1 xv) ⟹
11     Eval env (Mat x ((p,e)::ys))
12       (b (PMATCH xv ((PMATCH_ROW pat (K T) res)::yrs)))
```

Now let us look at the assumptions on the theorem and how they are discharged by the code generator when the theorem is used. The subterm evaluations are on lines 4 and 6-8. The code generator derives theorems of these forms by recursively calling its syntax-traversing function. As mentioned above, each such translation comes with assumptions and these assumptions are captured by variables `p1` and `p2`. When the theorem above is used lines 9 and 10 will be left as assumptions, but the internal assumptions, `p1` and `p2`, are passed in these lines to higher levels (see [7] for details).

The other lines 1, 2 and 5 are simple assumptions that are discharged by evaluation and an automatic tactic. Line 1 states that all the variables in the pattern have distinct names. `PMATCH` allows multiple binds to the same variable, but CakeML's pattern matching semantics does not allow this. Line 2 states that the pattern function in HOL is injective; and line 5 states that the CakeML pattern `p` corresponds to the pattern function `pat` in the current CakeML environment `env` and based on refinement invariant `a` for the input type.

The CakeML code generator can only generate code for `PMATCH`-based case expressions when there is an equivalent pattern expression in CakeML. This means, for instance, that one cannot generate code for case expressions with multiple binds to a variable, those that use non-constructor based patterns, or those that use guards. `PMATCH`-based case expressions that do not fall into this subset can usually be translated by removing these features first. We provide automated tools which work for most situations[17], although using this feature-removing automation can, in the worst case, lead to significant changes in structure of the terms, even replacing them with bulky decision trees similar to those of the classical approach.

We have used this `PMATCH`-based translation to produce high-quality CakeML code for all of the case expressions in the HOL Light kernel.

7 Summary

This paper presents a new representation, `PMATCH`, for case expressions in higher-order logic which faithfully captures the structure of the user's input. Because pattern-matching structure is retained, proofs over `PMATCH` expressions are simpler, and code generated from `PMATCH` expressions is better. Moreover, `PMATCH` is more general than currently-used representations: it supports guards, views, unbound variables, arithmetic expressions in patterns and even non-injective functions in patterns.

In addition to the new representation itself, we provide tools for working with `PMATCH` expressions in HOL4. Our tools include a parser and pretty printer, conversions for simplification and evaluation, and a pattern-compilation algorithm

[17] The exceptions are non-constructor patterns that are not part of a constructor family.

inside the logic. This pattern compilation is used to check the exhaustiveness of lists of patterns as well as for implementing powerful techniques for eliminating redundant rows. Furthermore, we have extended CakeML's proof-producing code generator to translate `PMATCH` expressions into high-quality CakeML code.

At present, our tools are already more powerful and convenient than the existing support for case expressions in the major HOL systems. In the future we plan to extend them further. In particular, we plan to improve the support for advanced patterns like arithmetic expressions.

Acknowledgements. The second author was partially supported by the Royal Society UK and the Swedish Research Council.

References

1. Augustsson, L.: Compiling pattern matching. In: FPCA. pp. 368–381 (1985), http://dx.doi.org/10.1007/3-540-15975-4_48
2. Harrison, J.: HOL Light: An overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs, Proceedings. LNCS, vol. 5674, pp. 60–66. Springer (2009), http://dx.doi.org/10.1007/978-3-642-03359-9_4
3. Krauss, A.: Partial recursive functions in higher-order logic. In: Int. Joint Conference on Automated Reasoning (IJCAR 2006), LNCS. pp. 589–603. Springer-Verlag (2006)
4. Kumar, R., Arthan, R., Myreen, M.O., Owens, S.: HOL with definitions: Semantics, soundness, and a verified implementation. In: Klein, G., Gamboa, R. (eds.) ITP, Proceedings. LNCS, vol. 8558, pp. 308–324. Springer (2014), http://dx.doi.org/10.1007/978-3-319-08970-6_20
5. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Jagannathan, S., Sewell, P. (eds.) POPL. pp. 179–192. ACM (2014)
6. Maranget, L.: Compiling pattern matching to good decision trees (September 2008)
7. Myreen, M.O., Owens, S.: Proof-producing translation of higher-order logic into pure and stateful ML. J. Funct. Program. 24(2-3), 284–315 (2014)
8. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)
9. Peyton Jones, S.L.: The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1987)
10. Slind, K.: Function definition in higher-order logic. In: In Theorem Proving in Higher-Order Logics. 9th International Conference, TPHOLs '96. Springer-Verlag LNCS. pp. 381–397. Springer (1996)
11. Slind, K., Norrish, M.: A brief overview of HOL4. In: TPHOLs. pp. 28–32 (2008)
12. Wadler, P.: Views: A way for pattern matching to cohabit with data abstraction. In: POPL. pp. 307–313. ACM (1987)
13. Wenzel, M., Paulson, L.C., Nipkow, T.: The isabelle framework. In: Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings. pp. 33–38 (2008), http://dx.doi.org/10.1007/978-3-540-71067-7_7