Pattern Minimization Problems over Recursive Data Types

Alexander Krauss

Technische Universität München, Institut für Informatik
Boltzmannstr. 3, 85748 Garching, Germany
http://www.in.tum.de/~krauss

Abstract

In the context of program verification in an interactive theorem prover, we study the problem of transforming function definitions with ML-style (possibly overlapping) pattern matching into minimal sets of independent equations. Since independent equations are valid unconditionally, they are better suited for the equational proof style using induction and rewriting, which is often found in proofs in theorem provers or on paper.

We relate the problem to the well-known minimization problem for propositional DNF formulas and show that it is Σ_2^P-complete. We then develop a concrete algorithm to compute minimal patterns, which naturally generalizes the standard Quine-McCluskey procedure to the domain of term patterns.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]

General Terms Theory, Verification

Keywords Complexity, Pattern Matching, Theorem Proving

1. Introduction

Pattern matching plays an important role in functional languages, where it is used as a structured and concise way of expressing complex branching behaviour. Its power is best demonstrated by the Haskell fragment given in Fig. 1 (due to Okasaki (1999)), which implements the rebalancing operation for red-black trees.

Here, a complex series of primitive tests is expressed in just four seemingly symmetric equations, plus a default case. A large part of the elegance attributed to functional programming is due to this powerful construct.

However, when looking closely, the elegance is misleading: Since patterns have a top-to-bottom semantics and the above patterns overlap, the four cases in the definition are not at all symmetric: If we reorder the first four equations, we get a different function. Moreover, nontrivial computation is necessary to determine the set of inputs which will be handled by the default case.

This is all fine if we just want to run the program, since all the technical details are handeled by the compiler. However, when reasoning statically about the code (e.g. to verify its correctness in a theorem prover like Isabelle/HOL (Nipkow et al. 2002)), the complexity is back: the equations as written above are unsuitable for equational reasoning, since they are not valid independently: obviously, $balance\ t = t$ is not universally true.

To reason equationally, we must first disambiguate the specification by removing overlap between the different clauses, thus recovering the set of equations that was actually "meant". Figure 2 demonstrates this step on a small example function. The equation $sep\ a\ xs\ =\ xs$ in Fig. 2(a) is not a theorem (otherwise sep would be just a projection), and needs to be instantiated to the cases that do not overlap with the first equation. The resulting equations, given in Fig. 2(b) can then be used independently as rewrite rules.

In other words, we are considering the transformation of a functional program with overlapping patterns into a *term rewrite system*, where rules can be applied in any order.

This disambiguation step reveals the real complexity of the balance operation. If we split up naively, we end up with a total of 91 (!) equations. Although most of the patterns that arise are not actually red-black trees, since the usual invariant does not hold, they must be generated at that point, since they belong to the specification of the balance function, which is defined on the free datatype *Tree* in general. In general, this disambiguation can lead to an exponential blowup.

In this paper, we study how to do this transformation in a way that the number of resulting equations is *minimal*. This is particularly important, as the size of the specification criticallye influences the size and complexity of subsequent proofs: e.g. induction proofs

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

$$
\begin{aligned}
&\textbf{data}\ Color = R \mid B\\
&\textbf{data}\ Tree\ a = E \mid T\ Color\ (Tree\ a)\ a\ (Tree\ a)\\
&balance :: Tree\ a \to Tree\ a\\
&balance\ (T\ B\ (T\ R\ (T\ R\ a\ x\ b)\ y\ c)\ z\ d) = (T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d))\\
&balance\ (T\ B\ (T\ R\ a\ x\ (T\ R\ b\ y\ c))\ z\ d) = (T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d))\\
&balance\ (T\ B\ a\ x\ (T\ R\ (T\ R\ b\ y\ c)\ z\ d)) = (T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d))\\
&balance\ (T\ B\ a\ x\ (T\ R\ b\ y\ (T\ R\ c\ z\ d))) = (T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d))\\
&balance\ t = t
\end{aligned}
$$

Figure 1. Pattern matching in the *balance* function

$$
\begin{aligned}
sep\ a\ (x : y : ys) &= x : a : sep\ a\ (y : ys) \\
sep\ a\ xs &= xs
\end{aligned}
$$

(a) Original equations

$$
\begin{aligned}
sep\ a\ (x : y : ys) &= x : a : sep\ a\ (y : ys) \\
sep\ a\ [\,] &= [\,] \\
sep\ a\ [x] &= [x]
\end{aligned}
$$

(b) Modified equations

Figure 2. Disambiguation of a function

about a recursive function usually split up into as many cases as there are defining equations.

After pinpointing the underlying problem, we exhibit a nice correspondence to the well-known problem of minimizing boolean formulas. This link allows us to prove that we are indeed dealing with a hard optimization problem (it is Σ_2^P-complete) (§3). Then we describe a method for finding minimal patterns, which is inspired by known results on boolean minimization and generalizes the classical Quine-McCluskey algorithm (McCluskey 1956) (§4). We implemented a prototype of the algorithm in Haskell. Despite the discouraging complexity results, it performs reasonably well on the problem sizes we encounter in theorem proving practice (§5).

2. Notation and Problem definition

We now introduce the necessary notation to state the pattern minimization problem formally. Since function types and polymorphism are not relevant for pattern matching, we can pretend that we live in a monomorphic first-order language, and work with many-sorted first order terms for a fixed set of sorts $\mathcal{S}$ and a finite sorted signature Σ.

For pattern matching, only constructor terms are relevant, so we just assume that all function symbols in Σ are data type constructors. Then the sort-indexed family of terms $(\mathcal{T}_s)_{s\in\mathcal{S}}$ is defined inductively as usual. A term is called *linear* or a *pattern*, if no variable occurs more than once. $\mathcal{P}_s$ denotes the set of patterns of sort s and $\mathcal{G}_s$ denotes the set of ground terms (i.e. terms without variables). We write Σ_s for the sets of constructors for values of sort s.

Since patterns must be linear and we are only concerned with matching, we can replace all variables by the wildcard symbol $*$. Formally, the wildcard carries a sort annotation ($*_s$), such that the sort of a term is always uniquely defined. However, we will liberally drop sort annotations that are clear from the context.

We write $t \preceq q$ if t is an instance of q (and $t \prec q$ if it is a proper instance). Given a pattern p, we can express its "semantics" as the set of all ground instances:

$$[p] := \{g \in \mathcal{G} \mid g \preceq p\}\ .$$

For finite sets of patterns we set $[P] := \bigcup_{p\in P}[p]$.

Computing the intersection $p \wedge q$ of two patterns is a degenerate case of unification. Note that this is a partial operation: if the patterns are disjoint, we write $p \wedge q = \bot$. We write $\sup(p, q)$ for the supremum with respect to $\preceq$: E.g. $\sup(f(a, *), f(*, b)) = f(*, *)$. More generally, $(\mathcal{P}_s \cup \{\bot\}, \preceq)$ is a complete lattice, a special case of the subsumption lattice for terms described by Huet (Huet 1980).

For the examples and constructions in this paper, we implicitly assume that Σ contains the constructors T and F for booleans, 0 and Suc for naturals and suitable constructors for n-tuples, written $\langle\cdot,\cdot,\cdot\rangle$. We ignore currying in this paper and assume that function arguments are always tupled. For example, the patterns of the function in Fig. 2(a) are written $\{\langle *, \mathsf{Cons}(*, \mathsf{Cons}(*, *))\rangle,\ \langle *, *\rangle\}$.

Observe that we may have $[p] = [q]$ and $p \neq q$: For example, $[C(*, \ldots, *)] = [*]$ if C is the only constructor of the given sort[1]. To resolve this ambiguity, we define $\lceil p \rceil = \sup\{q \in \mathcal{P} \mid [q] = [p]\}$. Note that $\lceil p \rceil$ can easily be computed from p.

2.1 Pattern minimization and complement

We can now state the problems formally and relate them to the informal discussion above. As usual, optimization problems are stated as decision problems:

> DISAMBIGUATION: Given patterns $p_1, \ldots, p_n$ and an integer k, are there sets of patterns $P_1, \ldots, P_n$, such that for each $i \in \{1, \ldots, n\}$, $[P_i] = [p_i] \setminus \bigcup_{j=1}^{i-1}[p_j]$, and the total number of patterns in $P_1, \ldots, P_n$ is less than k?

Note that we do not require that *all* the resulting patterns be non-overlapping. Within one group P_i, the patterns may overlap, since they are stemming from the same equation (with the same right hand side).

To approach this problem, we study some related problems which are slightly simpler to express, like that of building a complement:

> PAT COMPLEMENT: Given a finite set P of patterns and an integer k, is there a set Q of at most k patterns, such that $[Q] = \mathcal{G} \setminus [P]$?

The related problem of just minimizing a set of patterns is given as follows:

> MIN PAT: Given a finite set P of patterns and an integer k, is there a set P' of at most k patterns, such that $[P'] = [P]$?

Technical note: For the complement-like problems, the bound k must be given in unary notation. This avoids that we merely measure the output complexity. A set of patterns can grow exponentially under complementation, so any algorithm computing it must take exponential time. If k is in unary, then the size of the complemented patterns is again polynomial in the size of the input, and we are measuring the complexity of the actual optimization process, not its result. The same technique is used by Umans et al. (2006).

Example 1. *With the following patterns over the datatype* $\mathbf{data}\ T = A \mid B \mid C$*, disambiguation must lead to an exponential blowup:*

$$
\begin{array}{c}
\langle A, *, *, *, * \rangle \\
\langle *, A, *, *, * \rangle \\
\langle *, *, A, *, * \rangle \\
\langle *, *, *, A, * \rangle \\
\langle *, *, *, *, A \rangle \\
\langle *, *, *, *, * \rangle
\end{array}
$$

It is easy to see that the last pattern stands for all value combinations that do not contain A. But this set of values cannot be expressed compactly by some patterns, since any pattern that has a wildcard at some position cannot be a candidate for the last equation because it would match a term having an A at that position. Thus, the patterns for the default case cannot have any wildcards, and therefore we need all combinations of B and C.

While the above example demonstrates a blowup that is unavoidable, the following (artificial) example shows that, in theory, an optimization can even save us from an exponential blowup:

[1] This implies that we model a language where pattern matching cannot trigger any possibly non-terminating computations. This is true for strict languages, and for total languages like Isabelle/HOL.

Example 2. *For a given $n \in \mathbb{N}$, we construct a function with $\frac{n(n-1)}{2}$ boolean arguments, and with n equations. We assign indices $1 \ldots n$ to the equations, and we associate a pair (i, j) with $1 \leqslant i < j \leqslant n$ to each argument position. Now, equation k has at argument position (i, j) the pattern T, if $k = i$ and F if $k = j$. Otherwise there is a wildcard pattern $*$.*

For $n = 3$, this construction yields a variation of the diagonal function (cf. Wadler 1987):

$$diagonal :: Bool \to Bool \to Bool \to Int$$
$$diagonal\ T\ T\ \_ = 1$$
$$diagonal\ F\ \_\ T = 2$$
$$diagonal\ \_\ F\ F = 3$$

Since the equations i and j obviously have different patterns at argument position (i, j), they are all disjoint. Hence the optimal disambiguation is to leave everything as it is. However, the naive disambiguation fails to recognize this and produces exponentially many equations.

Now let us look at the relationship between the different problems. Obviously PAT COMPLEMENT cannot be any harder than DISAMBIGUATION, as we can see from the *balance* example: A catch-all pattern in the end will be replaced by the complement of the preceding patterns.

The following lemma shows that the two problems are actually equivalent:

Lemma 3. DISAMBIGUATION *can be reduced (in P-time) to* PAT COMPLEMENT.

Proof. We do a reduction between the optimization problems, showing how to disambiguate optimally if we can compute minimal complements.

We first show how we can use PAT COMPLEMENT to *subtract* a set of patterns Q from a pattern p:

First compute $Q' = \{p \wedge q \mid q \in Q,\ p \wedge q \neq \bot\}$. Now consider all the positions where p has a wildcard and call them $\pi_1, \ldots, \pi_n$. Since the patterns in Q' are instances of p, they can only differ at these positions. We remove the outer structure and replace it by a tuple: $Q'' = \{\langle q|_{\pi_1}, \ldots, q|_{\pi_n}\rangle \mid q \in Q'\}$. We can now solve PAT COMPLEMENT for Q'' to compute the minimal pattern set C with $[C] = \mathcal{G} \setminus Q''$. We obtain the result of the subtraction by adding the term structure of p again: $R = \{p[c_1, \ldots, c_n] \mid \langle c_1, \ldots, c_n\rangle \in C\}$. Since C is minimal, R must also be minimal.

DISAMBIGUATION is now easily reduced to multiple subtractions. □

Let us demonstrate the above reduction of subtraction to complementation by a small example: Consider the datatype

$$\mathbf{data}\ T = A \mid B\ Nat \mid C\ (Nat, Nat)$$

and suppose we want to compute

$$\begin{array}{cc} & \langle C(*,*), \quad * \quad \rangle \\ - & \langle C(0,0), \quad A \quad \rangle \\ - & \langle \quad * \quad , B(Suc(*)) \rangle \end{array}$$

If we want to subtract the first and second from the third pattern, we first compute the intersections, obtaining

$$\begin{array}{cc} & \langle C(*,*), \quad * \quad \rangle \\ - & \langle C(0,0), \quad A \quad \rangle \\ - & \langle C(*,*), B(Suc(*)) \rangle \end{array}$$

We remove the outer term structure and replace it by a tuple. The first pattern is now a universal pattern, hence we have reduced the problem to computing the complement:

$$\begin{array}{cc} & \langle \quad * \quad , \quad * \quad , \quad * \quad \rangle \\ - & \langle \quad 0 \quad , \quad 0 \quad , \quad A \quad \rangle \\ - & \langle \quad * \quad , \quad * \quad , B(Suc(*)) \rangle \\ \hline = & \langle Suc(*) , \quad * \quad , \quad A \quad \rangle \\ & \langle \quad * \quad , Suc(*) , \quad A \quad \rangle \\ & \langle \quad 0 \quad , \quad 0 \quad , \quad B(0) \quad \rangle \\ & \langle \quad * \quad , \quad * \quad , \quad C(*,*) \quad \rangle \end{array}$$

After that, we just add the outer structure $\langle C(\cdot,\cdot), \cdot\rangle$ again, and obtain the result of the subtraction.

3. Complexity Results

In this section, we will show that pattern minimization problems can encode the well known-problem of minimizing boolean formulas in Disjunctive Normal Form (DNF).

This problem has already recieved a lot of attention, as it is crucial for the design of digital circuits. Many exact and heuristic methods have been studied, the most well-known probably being the classical algorithm by Quine & McCluskey (McCluskey 1956), on which we will base our pattern minimization algorithm in §4.

Despite the high practical importance, the exact complexity of the problem has only recently been settled, when Umans proved it Σ_2^P-complete in his PhD thesis (Umans 2000). The complexity class Σ_2^P belongs to the polynomial hierarchy and contains the problems that can be solved by a nondeterministic Turing machine with access to a SAT oracle that it can use to solve NP-complete problems in a single step[2]. Σ_2^P can be seen as "one level up" from NP, and its canonical complete problem is QSAT_2, the satisfiability problem for formulas of the form $\exists\vec{x}\forall\vec{y}.\,\phi(\vec{x},\vec{y})$ where $\vec{x}$ and $\vec{y}$ are vectors of boolean-valued variables. (For more details, see Papadimitriou 1994)

The DNF minimization problems can be stated as follows:

> MINIMUM EQUIVALENT DNF (MIN DNF): Given a DNF formula ϕ and an integer k, is there a DNF formula equivalent to ϕ with at most k terms[3]?
>
> SHORT CNF: Given a formula ϕ in Conjunctive Normal Form (CNF) and an integer k in unary notation, is there a DNF formula equivalent to ϕ with at most k terms?

Both problems are known to be complete for Σ_2^P (Schaefer and Umans 2002).

The central idea in showing that MIN PAT is Σ_2^P-complete is that DNF formulas can be mapped to patterns:

Definition 4. *Let ϕ be a boolean DNF formula with variables v_1 through v_n. It has the form $\phi = t_1 \vee \ldots \vee t_k$, where each t_i is a conjunction of literals, which we view as a set. Then*

$$E(\phi) = \{\ \langle p_1^1, \ldots, p_1^n\rangle\ , \ldots,\ \langle p_k^1, \ldots, p_k^n\rangle\ \}$$

where

$$p_i^j = \begin{cases} \mathsf{T} & \text{if } v_i \in \mathit{Literals}(t_j) \\ \mathsf{F} & \text{if } \neg v_i \in \mathit{Literals}(t_j) \\ * & \text{otherwise} \end{cases}$$

For example $E(v_1\bar{v}_3 \vee \bar{v}_2 v_3) = \{\langle \mathsf{T}, *, \mathsf{F}\rangle, \langle *, \mathsf{F}, \mathsf{T}\rangle\}$.
Obviously, $\phi(b_1, \ldots, b_n)$ is true iff $\langle b_1, \ldots, b_n\rangle \in [E(\phi)]$.

[2] This is strictly more than the "guessing" facility of nondeterminism, since it can also detect when no solution exists.

[3] In the terminology of boolean minimization, the word *term* specifically means a disjunct in a DNF. They should not be confused with the first-order terms that we use as patterns.

Theorem 5. *(1) Deciding whether a given set of patterns P is incomplete (i.e. $[P] \neq [*]$) is NP-hard.*
(2) MIN PAT *is Σ_2^P-hard.*
(3) PAT COMPLEMENT *is Σ_2^P-hard.*

Proof. We reduce from the related boolean problems using the embedding from Definition 4.

(1) Reduction from SAT: Let ϕ be in CNF. Using deMorgan's laws we produce the DNF formula ψ equivalent to $\neg\phi$. Then

$$[E(\psi)] \neq [*] \Leftrightarrow \psi \text{ is not a tautology}$$
$$\Leftrightarrow \phi \text{ is satisfiable.}$$

(2) Reduction from MIN DNF.
(3) Reduction from SHORT CNF, again interpreting the CNF formula as a negated DNF formula.

□

The incompleteness problem is interesting, since it is actually solved by most compilers of functional languages, which can issue a warning when the patterns of a function definition do not cover all cases. However, the exponential behaviour of the implementations does not seem to pose any problems, since the problem instances are usually small.

By a simple guess-and-check argument, we can show that MIN PAT and PAT COMPLEMENT are also contained in Σ_2^P:

Lemma 6. *The equivalence problem of two pattern sets is in co-NP.*

Proof. For given sets of patterns P and P', we can nondeterministically choose a ground term, and check if it is either covered by both P and P' or by none of them. □

Theorem 7. MIN PAT *and* PAT COMPLEMENT *are Σ_2^P-complete.*

Proof. To show that MIN PAT $\in \Sigma_2^P$, note that a nondeterministic Turing machine with access to a SAT oracle can solve our problem as follows: For a given input P and integer k, it nondeterministically guesses a pattern set P' of size k. It remains to check if $[P] = [P']$. Due to Lemma 6, this can be done by the SAT oracle.

For PAT COMPLEMENT $\in \Sigma_2^P$, a similar argument works. Then, with Thm. 5 we have completeness for both problems. □

4. A Minimization Algorithm

In this section, we will describe an algorithm which computes minimal patterns. We will focus on the MIN PAT problem first, but with simple modifications (sketched later), we can also use the procedure to solve the other problems.

Again, our algorithm arises from the analogy to boolean mimization. It is in fact a generalization of the well-known Quine-McCluskey method (McCluskey 1956).

In short, the Quine-McCluskey procedure proceeds as follows to minimize a formula ϕ:

(1) Write ϕ in canonical disjunctive normal form, i.e. as a disjunction of "minterms". These are products (i.e. conjunctions) of literals where each variable occurs either positively or negatively. Minterms correspond to the entries in the truth table where ϕ becomes true.
(2) From the minterms, constuct the "most general terms that imply ϕ", that is, conjunctions of literals that imply ϕ, but when one literal is removed, the result does not imply ϕ. These terms are called *prime implicants*.
(3) Find a minimal subset of prime implicants that covers all minterms of ϕ. Then the minimized formula is the sum (disjunction) of these prime implicants.

Example 8.

(1) Consider the following formula in canonical disjunctive normal form:

$$\phi = \bar{x}\bar{y}\bar{z}\bar{w} \vee \bar{x}y\bar{z}\bar{w} \vee x\bar{y}\bar{z}\bar{w} \vee \bar{x}y\bar{z}w \vee \bar{x}yz\bar{w} \vee$$
$$x\bar{y}\bar{z}w \vee x\bar{y}z\bar{w} \vee \bar{x}yzw \vee \bar{x}\bar{y}z\bar{w} \vee xyzw$$

The terms of the disjunction are the (positive) minterms. They correspond to entries in the truth table for ϕ.
(2) The prime implicants are those terms that cannot be generalized further without leaving ϕ.

$$\{\bar{x}\bar{z}\bar{w}, \bar{y}\bar{z}\bar{w}, x\bar{y}\bar{z}, x\bar{y}\bar{w}, x\bar{z}w, \bar{x}y, yw\}$$

(3) By choosing a minimal set of prime implicants that cover all the minterms, we obtain a minimized formula.

$$\phi = x\bar{y}\bar{w} \vee x\bar{z}w \vee \bar{x}y$$

Step (1), which is often implicit in textbook descriptions (Katz and Borriello 2005; McCluskey 1986), means that we basically start from the full truth table of the function. Note that the number of minterms is generally exponential in the size of ϕ.

The exact method of combining the minterms to prime implicants in Step (2) is often only vaguely described in textbooks, and if it is described, the algorithm often takes exponential time. However, Strzemecki (1992) showed that this step can be done in polynomial time.

Finally, it remains to solve a covering problem in Step (3), which is known to be NP-hard (even in the particular instances arising here (Umans et al. 2006)).

In the following, we will see that this algorithm can be extended to the more general problem on patterns.

We adapt some terminology from boolean minimization: For a fixed pattern-set P and a pattern p, we say that p is an *implicant* iff $[p] \subseteq [P]$. An implicant is called *prime*, iff none of its proper generalizations is an implicant.

Obviously, a minimal covering can be constructed from the prime implicants: Any other patterns in a minimal covering could simply be generalized to some prime implicant.

4.1 Minterms

Our more general setting is different in one important aspect:

In the boolean case, the base set we are considering is just the finite product space $\{0,1\}^n$, whereas the set underlying our patterns is a possibly infinite set of terms. So it is not immediately clear what corresponds to the "truth table" of a boolean function.

However, the nature of pattern matching is still finitary in a certain sense, which allows us to generalize the boolean methods. The idea is to define inductively a set of terms, depending on the patterns we want to minimize, which behaves similarly to the product space.

These terms, called *minterms*, are mutually non-overlapping and cover all of $\mathcal{G}$. Furthermore, they respect the structure of P, in the sense that for a minterm m and a $p \in P$ either $m \preceq p$ or $m \wedge p = \bot$.

Definition 9 (Projection)**.** *For $P \subseteq \mathcal{P}_s$, $C \in \Sigma_s$ and $i \leqslant arity(C)$, we define the projection*

$$\Pi_{C,i}(P) = \begin{cases} \{p_i \mid C(p_1,\ldots,p_n) \in P\} \cup \{*\} & \text{if } * \in P \\ \{p_i \mid C(p_1,\ldots,p_n) \in P\} & \text{otherwise} \end{cases}$$

For example, $\Pi_{\langle\rangle,2}(\{\langle *, \mathsf{Suc}(*)\rangle, \langle 0, *\rangle\}) = \{\mathsf{Suc}(*), *\}$, and $\Pi_{\mathsf{Suc},1}(\{0, \mathsf{Suc}(0), \mathsf{Suc}(*)\}) = \{0, *\}$.

Definition 10 (Minterms). *A pattern set $P \subseteq \mathcal{P}_s$ is called* trivial, *iff $P = \{\}$ or $P = \{*\}$. We compute the set of minterms $MT(P)$ recursively as follows:*

$$MT(P) = \begin{cases} \{*\} & \text{if } P = \emptyset \text{ or } P = \{*\} \\ \bigcup_{C \in \Sigma_s} C(MT(\Pi_{C,1}(P)), \ldots, MT(\Pi_{C,n}(P))) & \text{otherwise} \end{cases}$$

Note that above the constructor C is lifted to sets:

$$C(A_1, \ldots, A_n) = \{C(a_1, \ldots, a_n) \mid a_1 \in A_1 \ldots a_n \in A_n\}$$

For example, $MT(0) = \{0\} \cup \mathsf{Suc}(MT(\Pi_{\mathsf{Suc},1}(\{0\})))$
$= \{0\} \cup \mathsf{Suc}(MT(\emptyset)) = \{0, \mathsf{Suc}(*)\}$.

We divide the set of minterms into *positive* ones that lie within $[P]$ and *negative* ones that are outside:

$$M_P^+ = \{m \in MT(P) \mid \exists p \in P.\, m \preceq p\}$$
$$M_P^- = MT(P) \setminus M_P^+$$

Example 11. *For $P = \{\langle \mathsf{Suc}(0), * \rangle, \langle *, 0 \rangle\}$, we have*

$$M_P^+ = \{\langle \mathsf{Suc}(0), 0 \rangle, \quad \langle \mathsf{Suc}(0), \mathsf{Suc}(*) \rangle, \quad \langle 0, 0 \rangle, \quad \langle \mathsf{Suc}(\mathsf{Suc}(*)), 0 \rangle\}$$

$$M_P^- = \{\langle 0, \mathsf{Suc}(*) \rangle, \quad \langle \mathsf{Suc}(\mathsf{Suc}(*)), \mathsf{Suc}(*) \rangle\}\ .$$

Defined like this, minterms satisfy the following properties:

Lemma 12 (Properties of minterms). *Let $p \in P$ and $m \in MT(P)$.*

(1) The elements of $MT(P)$ are pairwise disjoint, and $[MT(P)] = \mathcal{G}$.
(2) If $m \wedge p \neq \bot$ then $m \preceq p$
(3) $[m] \subseteq [P]$ iff $\exists p \in P.\, m \preceq p$
(4) For any pattern set $A \subseteq \mathcal{P}_s$, we have

$$[A] \subseteq [P] \iff \forall m \in M_P^-. \forall a \in A.\, m \wedge a = \bot$$

Proof. (1) Simple induction

(2) By induction on p. For $p = *$ the statement is trivial. If $p = C(p_1, \ldots, p_n)$, then $* \notin MT(P)$ and thus m must also start with a constructor. Since $m \wedge p \neq \bot$, we know that $m = C(m_1, \ldots, m_n)$ with $m_i \in MT(\Pi_{C,i}(P))$ and $m_i \wedge p_i \neq \bot$. By induction hypothesis we get $m_i \preceq p_i$ for each i, and thus $m \preceq p$.

(3) For the forward implication, assume $[m] \subseteq [P]$. Since $[m]$ cannot be empty, m must overlap with at least one element of P, and we can apply (2). The reverse implication is immediate.

(4) Obvious, since $[M_P^-] = \mathcal{G} \setminus [P]$.

□

Hence we can see M_P as a partitioning of $\mathcal{G}$, where each partition is represented by a minterm. Moreover, the partitioning is fine enough to be compatible with the shape of $[P]$.

Note that part (4) of the above lemma enables us to check algorithmically, if a given pattern (or set of patterns) is an implicant. This check will be required in the next sections.

4.2 Constructing Prime Implicants

The next step in the Quine-McCluskey procedure requires finding the prime implicants of the boolean formula.

In textbooks this is often done by repeatedly joining minterms to larger terms, until a fixpoint is reached. However, this procedure can have exponential runtime, which is unnecessary.

Strzemecki (1992) describes how to obtain prime implicants in polynomial time. Unfortunately, the paper is a little hard to read due to a lot of nonstandard and redundant notation. However, it can be reduced to a few simple ideas, which generalize nicely to our pattern world.

Lemma 13. *Let p be an implicant for P and $m \in M_P^+$ a minterm such that $m \preceq p$. Then there exists a minterm $m' \in M_P^+$, such that $[p] = [\sup(m, m')]$.*

Proof. We proceed by induction on m.

If $m = *$, choose $m' = *$.

Assume $m = C(m_1, \ldots, m_n)$. If $p = *$, we simply choose any $m' \in M_P^+$ whose topmost constructor is different from C. If this doesn't exist, then C must be the only constructor of the respective sort and we know that $[p] = [C(*, \ldots, *)]$ and we proceed as if p had that form.

If $p = C(p_1, \ldots, p_n)$, we have $m_i \prec p_i$, and the m_i are minterms. Applying the induction hypothesis we obtain minterms $m_i' \in M_{\Pi_{C,i}(P)}$, such that $p_i = \sup(m_i, m_i')$. We get $m' = C(m_1', \ldots, m_n') \in M_P$. And obviously $\sup(m, m') = p$. □

Corollary 14. *Every prime implicant can be written as $\lceil \sup(m, m') \rceil$ with some $m, m' \in M_P^+$*

This implies a simple polynomial-time algorithm for finding all prime implicants: Build the suprema of all possible pairs of positive minterms and remove those patterns that are not implicants (using Lemma 12(4)) or instances of others.

4.3 Essential Prime Implicants

A prime implicant is called *essential*, if it covers a minterm not covered by any other prime implicant. Since essential prime implicants must necessarily appear in the minimal covering, it is a useful optimization to generate them first. After this, only the remaining minterms must be covered by other prime implicants.

Also here, we can generalize Strzemecki's work, redefining the relevant notions for our framework:

Definition 15. *Informally, the set $G(t)$ of* simple generalizations *of t is the set of terms obtained from t by replacing exactly one non-wildcard subterm by $*$.*

Formally, $G(t)$ is defined recursively as follows:

$$G(*) = \{\}$$
$$G(C(t_1, \ldots, t_n)) = \{*\} \cup \bigcup_{i=1}^{n} C(t_1, \ldots, G(t_i), \ldots, t_n)$$

Lemma 16. *If $s \prec t$ then t is the supremum of some subset of $G(s)$.*

Proof. Informally, for each missing constructor in t compared to s, we include the corresponding element of $G(s)$ in the subset. Formally, use induction. □

Definition 17 (Neighborhood terms). *For given P and $m \in M_P^+$, we define the* neighborhood term *of m by $R_P(m) = \sup \{g \in G(m) \mid [g] \subseteq [P]\}$.*

Lemma 18. *Let $m \in M_P^+$ and $r = R_P(m)$. If $[r] \subseteq [P]$, then it is an essential prime implicant.*

Proof. Assume that $r = R_P(m)$ is an implicant. In order to see that r is prime, we show that every implicant i which contains m must be subsumed by r. So fix i with $m \preceq i$ and $[i] \subseteq [P]$. If $i = m$, then obviously $i \preceq r$. Otherwise we have $m \prec i$, and Lemma 16 yields $i = \sup G$ for some $G \subseteq G(m)$. Since i is an implicant, each element of G must also be, and thus $i \preceq R_P(m) = r$. Since

i was arbitrary, we know that any prime implicant containing m must be equal to r, and thus r is essential (to cover m). □

In fact, all essential prime implicants have the above form:

Lemma 19. *Every essential prime implicant equals $R_P(m)$ for some $m \in M_P^+$.*

Proof. Let e be an essential prime implicant. Then there exists an $m \preceq e, m \in M_P^+$ covered by no other prime implicant. Then e is an upper bound for $\{g \in G(m) \mid [g] \subseteq [P]\}$, which implies $R_P(m) \preceq e$. Since e is an implicant, we have $[R_P(m)] \subseteq [P]$. Applying the above lemma, we get the reverse inequality $e \preceq R_P(m)$, and thus $e = R_P(m)$. □

So, to compute all essential prime implicants, we have to compute $R_P(m)$ for every $m \in M_P^+$, and filter out those that are not implicants.

4.4 Overall Algorithm

Given a pattern set P to minimize, we proceed as follows:

(1) Compute M_P^+ and M_P^-.

(2) Compute the essential prime implicants E, as described in §4.3, and determine the set $\bar{M} = \{m \in M_P^+ \mid [m] \not\subseteq [E]\}$.

(3) Compute the set N of nonessential prime implicants (containing minterms from $\bar{M}$) as described in §4.2.

(4) Find a covering of $\bar{M}$ by a subset N' of N.

(5) Return $E \cup N'$.

Note that if we want to minimize the complement instead, we can just swap the roles of M_P^+ and M_P^-. For DISAMBIGUATION, we just partition the minterms into n classes, one for each of the original equations, and perform the following steps for each of the classes accordingly.

5. Implementation and Experiments

We implemented a prototype of the above procedure in Haskell and tested it on a small suite of examples stemming from user-contributed theories to the library of the Isabelle/HOL prover. From 232 definitions, we filtered out those that just use trivial pattern matching on one of the arguments, without any nesting. The 97 remaining examples were minimized by our prototype in less than 3 seconds on a 1.2 GHz laptop, and no individual example took more than half a second to process. In 16% of the cases, an improvement over the naive disambiguation method could be made.

This indicates that minimization is feasible and occasionally useful for definitions occurring in theorem proving practice. However, such figures always require a good amount of scepticism, as the sample is influenced by the restrictions of previous Isabelle versions. It reflects what people *could* already do, not what they *would like to do*. In particular, some functions in the developments of arithmetic decision procedures (Chaieb 2006, 2008) were developed in a way that tries to avoid excessive blowup.

In the following, we present three interesting examples in more detail:

Balance Our initial example, the balancing function for red-black trees, has five equations, which split up to 91, when done naively. Our minimization algorithm computes a minimum number of 59 patterns.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Nat} = \mathit{Z} \mid \mathit{S}\ \mathit{Nat} \quad \text{-- Written as numerals below} \\
\textbf{data}\ \mathit{Val} = \mathit{Nv}\ \mathit{Int} \mid \mathit{Bv}\ \mathit{Bool} \mid \mathit{Undef} \\
\mathit{interp} :: \mathit{Nat} \to [\,\mathit{Val}\,] \to \mathit{Val} \\
\begin{array}{lll}
\mathit{interp}\ 0\ [\,] & = \mathit{Nv}\ 0 & \text{-- Zero} \\
\mathit{interp}\ 1\ [\mathit{Nv}\ n] & = \mathit{Nv}\ (n+1) & \text{-- Succ} \\
\mathit{interp}\ 2\ [\mathit{Nv}\ m, \mathit{Nv}\ n] & = \mathit{Nv}\ (m+n) & \text{-- Plus} \\
\mathit{interp}\ 3\ [\,] & = \mathit{Bv}\ \mathit{True} & \text{-- True} \\
\mathit{interp}\ 4\ [\mathit{Bv}\ b] & = \mathit{Bv}\ (\neg\ b) & \text{-- Not} \\
\mathit{interp}\ 5\ [\mathit{Bv}\ b1, \mathit{Bv}\ b2] & = \mathit{Bv}\ (b1 \vee b2) & \text{-- Or} \\
\mathit{interp}\ 6\ [\mathit{Nv}\ n1, \mathit{Nv}\ n2] & = \mathit{Bv}\ (n1 < n2) & \text{-- Less} \\
\mathit{interp}\ \_\ \_ & = \mathit{Undef} &
\end{array}
\end{array}
$$

Figure 3. Interpreter function

Interpreter Function Consider the function *interp* given in Fig. 3. This is a somewhat crude specification of an interpreter for a simplistic expression language. Operator names are modeled as natural nunbers. Values are either booleans, numerical values or undefined

This specific example was given to the author by Tobias Nipkow, complaining that Isabelle produced too many equations, when disambiguating the definition. Slind's implementation (Slind 1999) produced either 36 or 39 cases, depending on the order of arguments of the function. By manual inspection, we were able to express the function in just 31 equations, which we were sure was the optimal solution. Only a few months later, when the algorithm presented here was implemented, the computer proved us wrong when it produced just 25 equations.

Numadd Our third example is a function that operates on a representation of arithmetic expressions and is used as part of a decision procedure for Presburger arithmetic. Figure 4 shows the pattern matching used here. Unfortunately, in this example the minimization brings no improvement over naive disambiguation: The set of 256(!) resulting equations resulting from the naive disambiguation is already minimal.

6. Discussion

6.1 Related work

Although the problem is simple to state and natural, it has, up to our knowledge, never been studied systematically.

Previously, Isabelle/HOL, as well as the HOL4 prover (Gordon and Melham 1993), removed overlap from pattern matching in a slightly ad-hoc manner (a procedure implemented by Slind (1999)), with no attempt to minimize the result. The wish to improve on this is what led us to this research.

To our knowledge, the only complexity result related to ML-like pattern matching is given in an unpublished extendend abstract by Baudinet and MacQueen (1985). It states that the problem of compiling a sequence of patterns in to a decision tree (i.e. a case expression) of minimal size, is NP-complete. Our transformation is different, since it produces a set of equations again, and not a decision tree. It is somewhat remarkable that this puts the problem into a different complexity class.

There has been a fair amount of research on pattern match compilation (Augustsson 1985; Wadler 1987; Fessant and Maranget 2001, and others), but it is hard to compare this work with ours, since it has the goal of producing code that can be implemented efficiently, either in the form of a case tree or of some kind of backtracking automaton. Our optimization problem is different, due to the focus on the equational view.

6.2 Other possible applications

While this work was motivated by a theorem proving application, the problem of pattern minimization is a general one, and there could be other applications. Disambiguating an equational specification into independent equations could be of use in a form of parallel pattern matching, where one tries to match with different patterns simultaneously (say, in different threads). Note however, that such a form of parallel matching differs from the usual meaning of the term: Usually (e.g. in the work of While and Field (2005)), different components of the *same* pattern are matched concurrently, in order to let the match fail when any of the components fails to match.

In contrast, unambigous pattern sets in our sense would allow to compute the matches of *different* equations in parallel. In a lazy language, prime implicants, being the most general patterns describing a certain set of values, would probably play an important role.

These are however just vague ideas and it is not clear if they could be exploited to make some improvement in the area of implementation of functional languages.

6.3 Alternative encodings

In the theorem proving context, one could also try to find alternative logical representations for overlapping patterns, instead of instantiating patterns until they no longer overlap. For example, the second equation of the sep function in Fig. 2(a) could also expressed as a conditional equation with quantifiers:

$$(\forall x\ y\ ys.\ xs \neq (x : y : ys)) \implies sep\ a\ xs = xs$$

One could also imagine a special matching construct that serves as a "native" representation of clausal function definitions in the logic.

However, the extra complexity introduced by such constructions makes proofs more technical and destroys the purely equational view, which is very common in the informal proofs from the literature on functional programming (see e.g. Hughes 1995 or Thompson 1999). Also, the automated reasoning tools would need to be adapted to deal with such constructions.

For this reason, this paper focused on the optimization opportunities (and their limits) on purely equational specifications, for which appropriate reasoning tools are already in place.

7. Conclusion

We identified several minimization problems on term patterns as they are used in functional programming languages, and proved that they are complete for Σ_2^P. We showed how an algorithm similar to the well-known Quine-McCluskey method can be used to solve these problems.

This work was motivated by the need to disambiguate specifications of recursive functions, transforming them to a form suitable for equational reasoning (i.e. term rewrite systems) in the theorem prover Isabelle/HOL. In this context, keeping the size of the specification small can be important, as it influences the size and complexity of proofs.

Our implementation performs reasonably well on the problems we currently encounter in theorem proving practice, but two issues remain: First, there exist examples (such as *Numadd* above) where the blowup is unavoidable, and the mininimization does not save us from getting overly large equational specifications. Second, for larger examples, the computational complexity of the problem might make exact minimization practically intractable. Then, one could try to apply heuristic methods. Here, we expect the methods developed for boolean minimization to be applicable in our context.

Although our main motivation is to reason about functional programs in a theorem prover, our results might be relevant for other areas, since they naturally generalize results on boolean minimization.

Another interesting question is, whether one can also do the encoding of §3 in the reverse direction, i.e. encode a given pattern minimization problem into a boolean formula and read off the solution from the minimized formula. Then we could readily use existing high-performance boolean minimizers (e.g. ESPRESSO (Brayton et al. 1984)) to solve our pattern problems. From Σ_2^P-completeness, we know that there must be such an encoding, but we could not find a natural one. The main problem here is that the pattern problems contain more structure, which may be destroyed during boolean minimization.

Acknowledgments

I want to thank the anonymous reviewers for their patience and the very detailed comments. Clemens Ballarin, Amine Chaieb and Tobias Nipkow also commented on previous drafts of this paper.

References

Lennart Augustsson. Compiling pattern matching. In *FPCA'85*, pages 368–381, 1985.

Marianne Baudinet and David MacQueen. Tree pattern matching for ML. URL `http://www.smlnj.org/compiler-notes/85-note-baudinet.ps`. Unpublished, 1985.

R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Boston, MA, 1984.

Amine Chaieb. Verifying mixed real-integer quantifier elimination. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference*, LNAI 4130, pages 528–540. Springer, 2006. ISBN 3-540-37187-7.

Amine Chaieb. *Automated methods for formal proofs in simple arithmetics and algebra*. PhD thesis, Technische Universität München, Germany, April 2008.

Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ICFP*, pages 26–37, 2001.

Michael Gordon and Tom Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

$$
\begin{array}{l}
\textbf{data}\ Nat = Z \mid S\ Nat \\
\textbf{data}\ T = C\ Nat \mid Bound\ Nat \mid Neg\ T \mid Add\ T\ T \mid Sub\ T\ T \mid Mul\ Nat\ T \\
numadd :: T \to T \to T \\
\begin{array}{lll}
numadd\ (Add\ (Mul\ c1\ (Bound\ n1))\ r1)\ (Add\ (Mul\ c2\ (Bound\ n2))\ r2) & = \ldots & \text{-- right hand sides omitted} \\
numadd\ (Add\ (Mul\ c1\ (Bound\ n1))\ r1)\ t & = \ldots & \\
numadd\ t\qquad\qquad\qquad\qquad & (Add\ (Mul\ c2\ (Bound\ n2))\ r2) = \ldots & \\
numadd\ (C\ b1) & (C\ b2) & = \ldots \\
numadd\ a & b & = \ldots
\end{array}
\end{array}
$$

Figure 4. Numadd

Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980.

John Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

Randy H. Katz and Gaetano Borriello. *Contemporary Logic Design*. Prentice Hall, 2005.

E. J. McCluskey. *Logic Design Principles*. Prentice Hall, 1986.

E. J. McCluskey. Minimization of boolean formulas. *Bell Lab. Tech. J.*, 35 (6):1417–1444, Nov 1956.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

Chris Okasaki. Red-black trees in a functional setting. *J. Funct. Program.*, 9(4):471–477, 1999.

Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, New York, 1994.

Markus Schaefer and Christopher Umans. Completeness in the polynomial-time hierarchy: Part I: A compendium. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 33, 2002.

Konrad Slind. *Reasoning About Terminating Functional Programs*. PhD thesis, Institut für Informatik, TU München, 1999.

Tadeusz Strzemecki. Polynomial-time algorithms for generation of prime implicants. *J. Complexity*, 8(1):37–63, 1992.

Simon Thompson. *Haskell: The Craft of Functional Programming (2nd Edition)*. Addison-Wesley, 1999.

Christopher Umans. *Approximability and completeness in the polynomial hierarchy*. PhD thesis, University of California, Berkeley, 2000.

Christopher Umans, Tiziano Villa, and Alberto L. Sangiovanni-Vincentelli. Complexity of two-level logic minimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(7):1230–1246, 2006.

Philip Wadler. Efficient compilation of pattern-matching. In S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, chapter 5. Prentice-Hall International, 1987.

Lyndon While and Tony Field. Optimising parallel pattern-matching by source-level program transformation. In Vladimir Estivill-Castro, editor, *ACSC*, volume 38 of *CRPIT*, pages 239–248. Australian Computer Society, 2005. ISBN 1-920682-20-1.