Poitín: Distilling Theorems From Conjectures

G.W. Hamilton\textsuperscript{1,2}

\textit{School of Computing}
\textit{Dublin City University}
\textit{Dublin, IRELAND}

Abstract
In this paper, we describe a new fully automatic theorem prover called Poitín which makes use of a novel transformation algorithm called distillation to prove input conjectures. The input conjectures are defined in a functional language and are transformed using the distillation algorithm. The result of this transformation can be easily inspected to see whether the original conjecture is true. Possible divergence of the transformation algorithm is detected, and this information is used to perform generalizations to ensure termination. We give several examples of the application of the theorem prover, and compare it to related work.

Key words: transformation, proving, divergence, generalization

1 Introduction

Two of the biggest problems in inductive theorem proving are the generation of appropriate intermediate lemmas and the introduction of generalizations. It has previously been shown in [21] how the possible divergence of a theorem prover can be used to suggest appropriate lemmas and generalizations. In this paper, we describe a fully automatic theorem prover called Poitín, which also uses possible divergence to suggest appropriate generalizations, but which does not require the construction of intermediate lemmas.

The approach which we take to theorem proving is more of a computational approach, similar to that proposed in [18] in conjunction with the supercompiler [19]. The input conjecture is regarded as a program which returns a boolean result, and is transformed into a more efficient equivalent program using a transformation technique which we call \textit{distillation}. The resulting program can then be easily checked to see whether the initial conjecture is true or false.

\textsuperscript{1} Thanks to Alan Bundy, James Brotherston and Lucas Dixon for helpful discussions regarding this work.

\textsuperscript{2} Email: hamilton@computing.dcu.ie

This is a preliminary version. The final version will be published in \textit{Electronic Notes in Theoretical Computer Science}

URL: www.elsevier.nl/locate/entcs
During this transformation, possible divergence is detected through the use of an embedding relation. If such a possible divergence is detected, generalization is performed. Care must be taken at this point not to over-generalize. Over-generalization is considered to have occurred if the value which is extracted is still required in the computation of the remaining program. As an example, consider trying to prove the following conjecture:

$$\forall x. \text{even} \ (\text{double} \ x \ \text{Zero})$$

Where the function \text{even} returns a boolean indicating whether or not its argument is even, and the function \text{double} accumulates within its second argument the result of doubling its first argument. As the function \text{double} has an accumulating parameter, we will encounter the following increasingly large terms during transformation:

$$\text{even} \ (\text{double} \ x \ \text{Zero})$$

$$\vdots$$

$$\text{even} \ (\text{double} \ x' \ (\text{Succ} \ (\text{Succ} \ \text{Zero})))$$

$$\vdots$$

A possible divergence will therefore be detected, and generalization performed. This generalization would usually involve replacing the accumulating parameter with a universally quantified variable (see [10]) to give the following:

$$\forall x. \forall y. \text{even} \ (\text{double} \ x \ y)$$

However, in this case, we can see that the value of this accumulating parameter is required to compute the result of the overall expression. In this paper, we show how such over-generalization can be avoided, thus allowing the proof of this conjecture to go through. The remainder of this paper is structured as follows. In Section 2, we define the language which is used to describe input conjectures. In Section 3, we describe the distillation algorithm which is used to transform the input conjectures. In Section 4, we describe the theorem prover Poitín, which uses the distillation algorithm to help prove input conjectures fully automatically. In Section 5 we consider related work, and Section 6 concludes.

2 Language

In this section, we describe the language which will be used throughout this paper.

Definition 2.1 (Language) The language for which the described transformations are to be performed is a simple higher-order functional language as shown in Fig. 1.

Programs in the language consist of an expression to evaluate and a set of function definitions. The intended operational semantics of the language is
normal order reduction. It is assumed that the language is typed using the
Hindley-Milner polymorphic typing system \cite{15,6}. Each constructor has a
fixed arity (for example, \texttt{Nil} has arity 0, and \texttt{Cons} has arity 2) and each
constructor application must be saturated. Within case expressions of the
form \texttt{case \ e\_0 of \ p\_1 : \ e\_1 | \cdots | p\_k : \ e\_k}, \ e\_0 is called the \textit{selector}, and \ e\_1 \ldots \ e\_k
are called the \textit{branches}. The patterns in case expressions may not be nested.
Methods to transform case expressions with nested patterns to ones without
nested patterns are described in \cite{1,20}.

## 3 Distillation

In this section, we define the distillation algorithm using a set of transformation
rules which attempt to convert a given expression into a more efficient
equivalent expression. We define these rules by identifying the next reducible
expression (\textit{redex}) within some context. An expression which cannot be broken
down into a redex and a context is called an \textit{observable}. These are defined as
follows.

**Definition 3.1 (Redexes, Contexts and Observables)** Redexes, contexts
and observables are defined by the grammar shown in Fig. 2, where \textit{red} ranges
over redexes, \textit{con} ranges over contexts and \textit{obs} ranges over observables.

The expression \( e\langle r \rangle \) denotes the result of replacing the ‘hole’ \( \langle \rangle \) in \( e \) by \( r \).

**Lemma 3.2 (Unique Decomposition Property)** For every term \( t \), either
\( t \) is an observable or there is a unique context \( c \) and redex \( r \) s.t.
\( t = c\langle r \rangle \).
red ::= f
    | (λv.e₀) e₁
    | case (v e₁...eₙ) of p₁ : e₁' | ... | pₖ : eₖ'
    | case (c e₁...eₙ) of p₁ : e₁' | ... | pₖ : eₖ'

con ::= ⟨⟩
    | con e
    | case con of p₁ : e₁ | ... | pₖ : eₖ

obs ::= v e₁...eₙ
    | c e₁...eₙ
    | λv.e

Fig. 2. Grammar of Redexes, Contexts and Observables

3.1 Transformation Rules

The transformation rules for the distillation algorithm are shown in Fig. 3. Within these rules, the parameter ρ gives the expressions which have been previously encountered during transformation, and the heads of the functions which were defined when they were first encountered. This environment will be empty when initially supplied to the distillation algorithm. The unique decomposition property shows that this algorithm is deterministic. The rules cover all possible kinds of expression (variable, constructor, lambda abstraction, function, application and case). In rule (1) for a variable application, the arguments in the application are further transformed. In rule (2), the arguments of a constructor application are also further transformed. In rule (3), the body of a lambda abstraction is further transformed. In rule (4), where the innermost redex is a function call, the overall expression is compared to previously encountered expressions within the environment ρ. If the current expression is an instance of a previously encountered one (i.e. there is an assignment of the variables within the previously encountered expression such that it is identical to the current expression up to α-conversion), then the current expression is replaced with an appropriate call to the function which was introduced the first time the expression was encountered. Otherwise, a new function is defined, the body of which is the result of transforming the original expression with its function call unfolded. The function which is defined is a local function definition of the form letrec f = e₀ in e₁, which may
\[
\begin{align*}
\mathcal{T}[v \ e_1 \ldots e_n] \rho &= v (\mathcal{T}[e_1] \rho) \ldots (\mathcal{T}[e_n] \rho) \\
\mathcal{T}[c \ e_1 \ldots e_n] \rho &= c (\mathcal{T}[e_1] \rho) \ldots (\mathcal{T}[e_n] \rho) \\
\mathcal{T}[\lambda v. e] \rho &= \lambda v. (\mathcal{T}[e] \rho) \\
\mathcal{T}[e(f)] \rho &= f' e_1 \ldots e_k, \quad \text{if } \exists (f' v'_1 \ldots v'_k, e') \in \rho. e(f) = e'[e_1/v'_1, \ldots, e_k/v'_k] \\
&= \text{letrec } f' = \lambda v_1 \ldots v_n. \mathcal{T}[e(e')] (\rho \cup \{(f' v_1 \ldots v_n, e(f))\}) \\
&\quad \text{in } f' v_1 \ldots v_n, \quad \text{otherwise}
\end{align*}
\]

where \( f \) is defined by \( f = e' \) and \( v_1 \ldots v_n \) are the free variables in \( e(f) \)

\[
\begin{align*}
\mathcal{T}[e(\lambda v. e_0) e_1] \rho &= \mathcal{T}[e(e_0[e_1/v])] \rho \\
\mathcal{T}[e(\text{case } (v e_1 \ldots e_n) \text{ of } p_1 : e'_1 | \cdots | p_k : e'_k)] \rho \\
&= \text{case } \mathcal{T}[v e_1 \ldots e_n] \rho \text{ of } \\
&\quad p_1 : (\mathcal{T}[e(e'_1[p_1/(v e_1 \ldots e_n)])]) \rho \\
&\quad \quad \vdots \\
&\quad \quad | p_k : (\mathcal{T}[e(e'_k[p_k/(v e_1 \ldots e_n)])]) \rho \\
\mathcal{T}[e(\text{case } (c e_1 \ldots e_n) \text{ of } p_1 : e'_1 | \cdots | p_k : e'_k)] \rho \\
&= \mathcal{T}[e((\lambda v_1 \ldots v_n. e'_0) e_1 \ldots e_n)] \rho \\
\text{where } p_i &= c v_1 \ldots v_n
\end{align*}
\]

Fig. 3. Transformation Rules for Distillation

contain non-local variables. When transforming the body of the newly defined function, the head of this new function and the original expression prior to unfolding are added to the environment \( \rho \). In rule (5), the parameter \( e_1 \) in a lambda application \( (\lambda v. e_0) e_1 \) is substituted for the argument \( v \) within the body \( e_0 \). In rule (6), if the selector in a **case** expression is not a constructor application, then the context of the case expression is distributed across its branches. Within the branches of the **case**, any occurrences of the selector expression are replaced with the corresponding pattern for that branch. In rule (7), if the selector in a **case** expression is a constructor application, pattern matching is applied and the appropriate branch is selected for further transformation. Rules (5) and (6) are valid only if there is no name clash between the free and bound variables of the expression being transformed. It is always possible to rename the bound variables of the expression so that this condition applies.
3.2 Generalization

If the transformation rules for distillation were left unsupervised, possible non-termination could arise. This non-termination can take one of two possible forms: *accumulating parameters* or *accumulating context*. An example of non-termination due to an accumulating parameter occurs during the transformation of the accumulating reverse function, where the following successively larger terms are encountered:

\[
qrev \; xs \; Nil \\
\vdots \\
qrev \; xs' \; (Cons \; x' \; Nil) \\
\vdots \\
qrev \; xs'' \; (Cons \; x'' \; (Cons \; x' \; Nil)) \\
\vdots
\]

An example of non-termination due to an accumulating context occurs during the transformation of the naive reverse function, where the following successively larger terms are encountered:

\[
rev \; xs \\
\vdots \\
\text{case} \; (rev \; xs) \; \text{of} \; \cdots \\
\vdots \\
\text{case} \; (\text{case} \; (rev \; xs) \; \text{of} \; \cdots) \; \text{of} \; \cdots \\
\vdots
\]

In both cases, a sub-term is becoming more deeply embedded within the overall term. We therefore allow transformation to continue until an embedding of a previous term is encountered within the current one. The form of embedding which is used is known as a *homeomorphic embedding*. The homeomorphic embedding relation was derived from results by Higman [8] and Kruskal [12] and was defined within term rewriting systems [7] for detecting the possible divergence of the term rewriting process. Variants of this relation have been used to ensure termination within supercompilation [17], partial evaluation [14] and partial deduction [3,13]. It can be shown that the homeomorphic embedding relation $\sqsubseteq$ is a *well-quasi-order*, which is defined as follows.

**Definition 3.3 (Well-Quasi Order)** A well-quasi order on a set $S$ is a reflexive, transitive relation $\leq_S$ such that for any infinite sequence $s_1, s_2, \ldots$ of elements from $S$ there are numbers $i, j$ with $i < j$ and $s_i \leq_S s_j$.

This ensures that in any infinite sequence of terms $t_0, t_1, \ldots$ there definitely exists some $i < j$ where $t_i \leq t_j$, so an embedding must eventually be encountered and transformation will not continue indefinitely. If $t_i \leq t_j$ then all of the sub-terms of $t_i$ are present in $t_j$ embedded in extra sub-terms. This is defined more formally as follows.
Definition 3.4 (Homeomorphic Embedding Relation)

Variable Diving Coupling
\[ x \triangleleft y \quad \text{for some } i \quad s \triangleleft t_i \quad \text{for all } i \]
\[ s \triangleleft \sigma(t_1, \ldots, t_n) \quad s_i \triangleleft \sigma(t_1, \ldots, t_n) \]

This embedding relation is extended slightly to be able to handle constructs such as \( \lambda \)-abstraction and case which may contain bound variables. In these instances, the corresponding bound variables within the two expressions must also match up. This is an extension of the embedding relations defined elsewhere in the literature, but we can show that this relation is still a well-quasi order. Some examples of the homeomorphic embedding relation are as follows.

Example 3.5

\[
\begin{align*}
  f\ x & \trianglelefteq g\ (f\ y) & f\ (g\ x) & \not\triangleright f\ y \\
  f\ x & \trianglelefteq f\ (h\ y) & f\ (g\ x) & \not\triangleright g\ (f\ y) \\
  f\ x & \trianglelefteq g\ (f\ (h\ y)) & f\ (g\ x) & \not\triangleright f\ (h\ y)
\end{align*}
\]

When an expression is encountered in which a previously encountered term is embedded, generalization is performed. This involves replacing with variables those sub-terms (other than variables) which are more deeply embedded within the current expression (thus requiring the application of the diving rule). When applying these rules, diving is given priority over coupling. This ensures that the generalization which is performed corresponds to a \textit{maximal difference match} as defined in [2]. Some examples of applying generalization are shown in Table 1.

<table>
<thead>
<tr>
<th>( s )</th>
<th>( t )</th>
<th>( t^g )</th>
<th>( \theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_1\ x )</td>
<td>( f_1\ x )</td>
<td>( f_2\ v )</td>
<td>( {v := f_1\ y} )</td>
</tr>
<tr>
<td>( f_1\ x )</td>
<td>( f_2\ y )</td>
<td>( f_1\ (f_2\ y) )</td>
<td>( {} )</td>
</tr>
<tr>
<td>( f_1\ (f_2\ x) )</td>
<td>( f_1\ (f_3\ v) )</td>
<td>( f_1\ (f_3\ v) )</td>
<td>( {v := f_2\ y} )</td>
</tr>
<tr>
<td>( f_1\ (f_2\ x) )</td>
<td>( f_3\ v_2 )</td>
<td>( f_3\ v_2 )</td>
<td>( {v_1 := f_1\ (f_4\ v_2), v_2 := f_2\ y} )</td>
</tr>
</tbody>
</table>

Table 1
Examples of Generalization

In these examples, the shaded parts of the current term \( t \) represent the additional term structure which was not present within the previously encountered
term $s$. The sub-terms which are more deeply embedded within $t$ can easily be identified as the holes within these shaded areas. These are the sub-terms which are replaced with variables within the resulting generalized term $t^g$. The values of the extracted sub-terms are given by the substitution $\theta$.

If we were to extract all those sub-terms which were more deeply embedded within the current term, over-generalization might occur. This will be the case when the sub-term which is extracted is intermediate within the remaining generalized term. For example, consider the conjecture shown in Fig. 6. Using our definition of generalization, the accumulating parameter within this conjecture would be extracted, even though it is intermediate within the remaining generalized term. In distillation, we get round this problem by firstly transforming the generalized term, then substituting back in to the resulting term those extracted sub-terms which are intermediate within it, and finally transforming the expression resulting from this substitution. Those sub-terms which are not intermediate are permanently extracted through the introduction of a \texttt{let} construct. As the result of transforming any term will itself be a term which constructs no intermediate structures, it is quite straightforward to determine whether a sub-term which is substituted into it is intermediate. The \textit{un-generalization} of a term is defined as follows:

\[
\text{ungeneralize}(e, \{\}) = e \\
\text{ungeneralize}(e, \{v := e'\} \cup \theta) \\
= \text{ungeneralize}(e[e'/v], \theta), \quad \text{if } v \text{ is intermediate within } e \\
= \text{ungeneralize(\texttt{let } v = e' \texttt{ in } e, \theta), \quad \text{otherwise}}
\]

As an extracted sub-term may itself contain an embedding of a previously encountered term, a further embedding could be encountered immediately if this sub-term were substituted back in. To avoid this situation occurring, the embedded term is removed from the set of previously encountered expressions $\rho$ before continuing with the transformation. This also allows us to be able to obtain the most specific embedding of a term, as there could otherwise be more than one term within $\rho$ which is embedded within the current term.

Generalization need only be applied when the innermost redex is a function call, as any possible non-terminating transformation must involve the unfolding of a recursive function call. In this case, the current expression is checked to see if it is a renaming of a previously encountered expression. If this is the case, then folding is performed. Otherwise, the current expression is checked to see if a previously encountered expression is embedded within it. If this is the case, then generalization is performed as described above. Otherwise, the function call is unfolded and the expression prior to unfolding is added to $\rho$. Rule (4) for distillation must therefore be changed to the following to define this more formally.
\[ T[e(f)] \rho \]
\[ = f' e_1 \ldots e_k, \quad \text{if } \exists (f' v'_1 \ldots v'_k, e'') \in \rho. e(f) = e''[e_1/v'_1, \ldots, e_k/v'_k] \]
\[ = T[\text{ungeneralize}(T[e^g] \rho, \theta)] (\rho \setminus \{(e_1, e_2)\}), \text{if } \exists (e_1, e_2) \in \rho. e_2 \leq e(f) \]
\[ \text{where } (e^g, \theta) = \text{generalize}(e_2, e(f)) \]
\[ = \text{letrec } f' = \lambda v_1 \ldots v_n. T[e(e')] (\rho \cup \{(f' v_1 \ldots v_n, e(f))\}) \]
\[ \text{in } f' v_1 \ldots v_n, \quad \text{otherwise} \]
where \(f\) is defined by \(f = e'\) and \(v_1 \ldots v_n\) are the free variables in \(e(f)\).

As the term currently being transformed may now contain \texttt{letrec} and \texttt{let} constructs, the homeomorphic embedding relation is extended in the obvious way to be able handle these, and new rules are defined for their transformation as follows:

\[ T[e(\text{letrec } f = e_0 \text{ in } e_1)] \rho \]
\[ = f' e_1 \ldots e_k, \quad \text{if } \exists (f' v'_1 \ldots v'_k, e'') \in \rho. e' = e''[e_1/v'_1, \ldots, e_k/v'_k] \]
\[ = T[\text{ungeneralize}(T[e^g] \rho, \theta)] (\rho \setminus \{(e_1, e_2)\}), \text{if } \exists (e_1, e_2) \in \rho. e_2 \leq e' \]
\[ \text{where } (e^g, \theta) = \text{generalize}(e_2, e') \]
\[ = \text{letrec } f' = \lambda v_1 \ldots v_n. \]
\[ \quad T[e(e_1[(\text{letrec } f = e_0 \text{ in } e_1)/f])] (\rho \cup \{(f' v_1 \ldots v_n, e')\}) \]
\[ \text{in } f' v_1 \ldots v_n, \quad \text{otherwise} \]
where \(e' = e(\text{letrec } f = e_0 \text{ in } e_1)\) and \(v_1 \ldots v_n\) are the free variables in \(e'\)

\[ T[e(\text{let } v = e_0 \text{ in } e_1)] \rho = \text{let } v = (T[e_0] \rho) \text{ in } (T[e(e_1)] \rho) \quad (9) \]

### 4 Poitín

In this section, we describe the Poitín theorem prover, and how it makes use of the distillation algorithm. At present, Poitín can only prove conjectures in which all the variables are universally quantified. The way in which the distillation algorithm is used is similar to the way in which Turchin has shown how the supercompilation algorithm can be applied in theorem proving [18]. In both cases, the boolean expression which is the conjecture to be proved is transformed, and the result of this transformation is then inspected to check whether the conjecture is true. This checking involves looking at all possible exit points from the resulting expression, and seeing whether these are all true. In addition, all of the loops in the resulting expression must terminate.
In [18], this is done by requiring that all functions are total, so the onus is on the user to show that this really is the case. Here, we note that the term resulting from distillation constitutes a *cyclic pre-proof* as defined in [4]. In order to prove that this terminates, it is therefore sufficient to show that it is *infinitely progressing*, thus making it a *cyclic proof*. As shown in [4] this will be the case if in every recursive function definition a sub-component of one of the arguments is a parameter in each recursive call of the function. If this is not the case, the proof attempt is abandoned as it would otherwise diverge. Note that, in addition to the distillation algorithm, the cyclic proof checking performed by Poitín is fully automatic.

To give some examples of the application of Poitín, we first of all consider an example which does not require generalization, but which can create difficulties for some theorem provers.

**Example 4.1** Consider the conjecture shown in Fig. 4.

```plaintext

\[
\text{eqnum } (\text{add } x \ y) \ (\text{add } y \ x)
\]

where

\[
\text{add } = \lambda x. \lambda y. \text{case } x \ of
\]

 Zero  :  y
 | \ Succ x :  \ Succ (\text{add } x \ y)

\[
\text{eqnum } = \lambda x. \lambda y. \text{case } x \ of
\]

 Zero  :  \ case y \ of
 | Zero :  True
 | \ Succ y :  False
 | \ Succ x :  \ case y \ of
 | Zero :  False
 | | Succ y :  eqnum \ x \ y

Fig. 4. First Example Conjecture
```

This states that addition is commutative, which is obviously true. This conjecture is transformed by distillation to give the term shown in Fig. 5. By inspecting this term, we can see that the initial conjecture must be true, as the only exit points from the term are *True*, and each of the functions \( f_1 \), \( f_2 \) and \( f_3 \) progress and will therefore terminate, as their recursive calls are applied to a sub-component of one of their arguments.

**Example 4.2** Consider the conjecture shown in Fig. 6.
This conjecture states that every doubled number is even, which is obviously true. Many existing theorem provers have problems proving this con-
jecture in its given form. The initial term to be transformed is as follows:

\[ \text{even (double } x \text{ Zero)} \]  

(1)

A little bit further in the transformation, the following term is obtained:

\[
\text{case (double } x \text{ Zero) of}
\]

\[
\text{Zero} : \text{True}
\]

| Succ \( x' \): case \( x' \) of

\[
\text{Zero} : \text{False}
\]

| Succ \( x'' \): even \( x'' \)

(2)

After another couple of steps, we obtain the following term:

\[
\text{case (double } x' \text{ ) of}
\]

\[
\text{Zero} : \text{True}
\]

| Succ \( x' \): case \( x' \) of

\[
\text{Zero} : \text{False}
\]

| Succ \( x'' \): even \( x'' \)

(3)

We can see that term (3) is a homeomorphic embedding of term (2), as indicated by the additional shaded term structure. The more deeply embedded term \( \text{Zero} \) is therefore replaced with the variable \( v \), and the resulting generalized term transformed to give the following:

\[
\text{letrec}
\]

\[
f = \lambda x. \lambda y. \text{case } x \text{ of}
\]

\[
\text{Zero} : \text{letrec } g = \lambda x. \text{case } x \text{ of}
\]

\[
\text{Zero} : \text{True}
\]

| Succ \( x' \): case \( x' \) of

\[
\text{Zero} : \text{False}
\]

| Succ \( x'' \): \( g \) \( x'' \)

\[
in g y
\]

| Succ \( x' \): \( f \) \( x' \) (Succ (Succ \( y \)))

\[
in f x v
\]

(4)
The term *Zero* is then substituted back in for \( v \), and the resulting term further transformed. Terms equivalent to the following are then encountered:

\[
f \ x \ Zero
\]  
(5)

\[
f \ x' \ (\text{Succ} \ (\text{Succ} \ Zero))
\]  
(6)

Again we can see that term (6) is a homeomorphic embedding of term (5), as indicated by the additional shaded term structure. The more deeply embedded term *Zero* is therefore replaced with the variable \( v' \) to give the following generalized term:

\[
f \ x' \ (\text{Succ} \ (\text{Succ} \ v'))
\]  
(7)

This expression is then further transformed, consuming the second argument to produce an expression equivalent to the following:

\[
f \ x' \ v'
\]  
(8)

The extracted term *Zero* is then substituted back in for \( v' \) to give the following:

\[
f \ x' \ Zero
\]  
(9)

We can now see that term (9) is a renaming of term (5). Folding can therefore be applied to obtain the final transformed term shown in Fig. 7. Again, by

```
  case x of
    Zero    : True
    | Succ x' : letrec f = \x. case x of
    Zero    : True
    | Succ x' : f x'
    in f x'
```

Fig. 7. Second Example Conjecture Transformed

inspecting this term, we can see that the initial conjecture must be true, as the only exit points from the term are *True*, and the function \( f \) progresses and therefore terminates as the recursive call to \( f \) is applied to a sub-component of the argument.

## 5 Related Work

The distillation algorithm and the Poitín theorem prover were largely inspired by Turchin’s work on supercompilation [19], and its use in theorem proving...
The distillation algorithm would be of equivalent power to the supercompilation algorithm if the terms which are extracted on performing generalization were not substituted back in to the generalized term. This means that over-generalization would occur quite frequently when using supercompilation, thus greatly limiting its power. Also, in order to show that the term resulting from supercompilation terminates, Turchin requires that all functions are total, so the onus is on the user to show that this really is the case. In Poitín, the termination of the term resulting from distillation is determined automatically.

A number of different approaches have been developed to identify potentially failing proof attempts, and to apply appropriate techniques to allow the proof to go through. Rippling is a powerful technique developed at Edinburgh for proving theorems involving explicit induction [5]. In the step case of an inductive proof, the induction conclusion typically differs from the induction hypothesis. Rippling uses annotations to mark these differences and applies annotated rewrite rules to remove them. In the case where no rewrite rules can be applied, the proof becomes blocked. In this case, proof critics [9] can be applied. Various critics for explicit induction have been developed that speculate missing lemmas, perform generalizations, etc. There are significant differences between the rippling approach, and the approach described here. Firstly, rippling works in an explicit induction setting, as opposed to the implicit approach described here. Secondly, in rippling, the difference matching is performed statically on the rewrite rules (although a dynamic version of rippling has also been developed [16]). In distillation, this difference matching is dynamically performed on each term as it is encountered during rewriting. Thirdly, rippling usually requires that an instance of the inductive hypothesis can be obtained within the induction conclusion. This involves moving the identified differences within the induction conclusion so that this will be the case. In distillation, we try to find a re-occurrence of any previously encountered term, whether or not this term is the inductive hypothesis. This is done by identifying the sub-terms which match with the previously encountered term, removing these, and transforming the remaining difference term so that the transformed difference terms will eventually match. Rippling often requires the use of additional lemmas to allow the proof to go through. This may therefore require a reasonable amount of search, and possible user guidance. In Poitín, no additional lemmas are required, thus reducing the amount of search required, and allowing proofs to be performed fully automatically. The notion of rippling has however been extended to be able to deal with existentially quantified variables and synthesis in the work on middle-out reasoning [11]. At present, Poitín can only prove theorems in which all variables are universally quantified.

Perhaps the most closely related work to that described here is the divergence critic [21]. The divergence critic also performs difference matching dynamically, and also performs a maximal difference match. The outcome of
this difference matching is then used to either speculate a lemma or suggest a generalization. In Poitín, no additional lemmas are required, so the maximal difference match is used to suggest generalizations only. One problem with the generalization performed by the divergence critic is that over-generalization might be performed if the proof term is not really diverging. This possibility is avoided in our approach by checking to see whether the extracted term is intermediate within the remaining generalized term. In [21], this problem is reduced by waiting until a sequence of three embedded expressions is encountered before performing generalization, and it is argued that as a result little over-generalization occurs.

6 Conclusions

In this paper, we have presented a novel transformation algorithm and theorem proving technique. We argue that the Poitín theorem prover greatly extends the range of theorems which can be proved fully automatically without the need for intermediate lemmas. Poitín is also fully deterministic and only needs to search through a subset of previously encountered expressions, rather than through a large collection of rules and axioms. We therefore argue that Poitín is likely to be more efficient than other theorem provers which have a relatively large search space and require backtracking.

There are a number of possible directions for further work. Firstly, the implementation of Poitín must be completed, and run on a wider range of test cases. This would allow a more thorough examination of the range of theorems which can be proved by Poitín, and a more detailed comparison with other theorem provers. Secondly, Poitín can currently only prove theorems in which all variables are universally quantified. Further work is continuing on extending Poitín to handle existentially quantified variables, and on extracting programs from the resulting proofs.

References


