Constructing Programs From Metasystem Transition Proofs

G.W. Hamilton and M.H. Kabir

School of Computing
Dublin City University
Dublin 9
IRELAND

Abstract. It has previously been shown by Turchin in the context of supercompilation how metasystem transitions can be used in the proof of universally and existentially quantified conjectures. Positive supercompilation is a variant of Turchin's supercompilation which was introduced in an attempt to study and explain the essentials of Turchin's supercompiler. In our own previous work, we have proposed a program transformation algorithm called *distillation*, which is more powerful than positive supercompilation, and have shown how this can be used to prove a wider range of universally and existentially quantified conjectures in our theorem prover *Poitín*. In this paper we show how a wide range of programs can be constructed fully automatically from first-order specifications through the use of metasystem transitions, and we prove that the constructed programs are totally correct with respect to their specifications. To our knowledge, this is the first technique which has been developed for the automatic construction of programs from their specifications using metasystem transitions.

1 Introduction

The construction of programs from their specifications is a difficult process which cannot always be performed fully automatically. In this paper, we show that if specifications are encoded in a specialised form, which we call *distilled form*, then we can construct programs meeting these specifications fully automatically. Distilled form is distinguished by creating no intermediate data structures; this means that existence proofs for specifications in distilled form will require no intermediate lemmas, and can therefore be performed fully automatically. Distilled form corresponds to the 'read-only' primitive recursive programs as defined by Jones [11], within which data can be higher-order. As shown in [11], this class of programs can be used to encode all problems which belong to the fairly wide class of *elementary* sets.

In previous work [8], we proposed the *distillation* program transformation algorithm, which was originally devised with the goal of eliminating intermediate data structures from functional programs. The distillation algorithm was largely influenced by positive supercompilation [7] (a variant of Turchin's supercompiler

[25]), but improves greatly upon it. For example, positive supercompilation can only produce a linear speedup in programs with a call-by-name semantics, while distillation can produce a superlinear speedup. The form of programs generated by the distillation algorithm is precisely our distilled form. This suggests a two-step process for the construction of programs from their specifications: firstly, the specifications are transformed using distillation, then the distilled specifications are provided as input to a theorem prover which can automatically construct programs meeting these specifications.

The step from a program to the application of a metaprogram to this program is a kind of *metasystem transition* [26,7]. Turchin has shown how metasystem transitions can be used in conjunction with supercompilation to prove explicitly quantified conjectures [24] by defining metaprograms for proving both universally and existentially quantified conjectures. However, he has not shown how programs can be constructed from these proofs.

We have previously shown how techniques similar to those of Turchin can be used in conjunction with the distillation algorithm to prove a wider range of universally and existentially quantified conjectures using metasystem transitions in our theorem prover Poitín [12,14]. In this paper we show how programs can be constructed fully automatically from first-order specifications which are in distilled form through the use of metasystem transitions, and prove that the constructed programs are totally correct with respect to their specifications. To our knowledge, this is the first technique which has been developed for the automatic construction of programs from their specifications using metasystem transitions.

The remainder of this paper is organised as follows. In Section 2, we describe the language which will be used throughout this paper. We give a brief overview of the distillation algorithm, and define the distilled form of expressions which is generated by the algorithm. In Section 3, we show how the Poitín theorem prover proves universally and existentially quantified conjectures through the use of metasystem transitions. We give an example proof and we show that the theorem prover is sound and complete for programs which are in distilled form. In Section 4, we show how programs can be constructed fully automatically from specifications which are in distilled form through the use of metasystem transitions. We give an example of this program construction and prove that the constructed programs are totally correct with respect to their specifications. Section 5 considers related work and concludes.

2 Distillation

In this section, we define the language used throughout this paper and we give a brief overview of the distillation algorithm.

2.1 Language

Definition 1 (Language). *The language used throughout this paper is a simple higher-order functional language as shown in Fig. 1.* □

$$
\begin{array}{rll}
\mathit{prog} ::= & e_0 \ \textbf{where} \ f_1 = e_1; \ \ldots; \ f_n = e_n; & \text{program} \\
e ::= & v & \text{variable} \\
& \mid c \ e_1 \ \ldots \ e_n & \text{constructor application} \\
& \mid \lambda v.e & \text{lambda abstraction} \\
& \mid f & \text{function variable} \\
& \mid e_0 \ e_1 & \text{application} \\
& \mid \textbf{case} \ e_0 \ \textbf{of} \ p_1 \Rightarrow \ e_1 \mid \ldots \mid p_k \Rightarrow \ e_k & \text{case expression} \\
& \mid \textbf{let} \ v = e_0 \ \textbf{in} \ e_1 & \text{let expression} \\
& \mid \textbf{letrec} \ f = e_0 \ \textbf{in} \ e_1 & \text{letrec expression} \\
\\
p ::= & c \ v_1 \ldots v_n & \text{pattern}
\end{array}
$$

Fig. 1. Language

Programs in the language consist of an expression to evaluate and a set of function definitions. It is assumed that the language is typed using the Hindley-Milner polymorphic typing system (so erroneous terms such as $(c \ e_1 \ldots e_n) \ e$ and **case** $(\lambda v.e)$ **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$ cannot occur). The variables in the patterns of **case** expressions and the arguments of λ-abstractions are *bound*; all other variables are *free*. We use $FV(e)$ to denote the free variables in the expression e. We require that each function has exactly one definition and that all variables within a definition are bound. The propositional operators (*and*, *or*, *implies*, etc.) are implemented as functions in this language.

Each constructor has a fixed arity; for example *Nil* has arity 0 and *Cons* has arity 2. Within the expression **case** e_0 **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$, e_0 is called the *selector*, and $e_1 \ldots e_k$ are called the *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 [1,27]. No variables may appear more than once within a pattern. We assume that the patterns in a **case** expression are non-overlapping and exhaustive. An example program in the language is shown in Fig. 2.

2.2 Distillation

Distillation [8] is a powerful program transformation technique to remove intermediate data structures from higher-order functional programs and represents a significant advance over the positive supercompilation algorithm [7]. Using the positive supercompilation algorithm, it is only possible to obtain a linear improvement in the run-time performance of programs with a call-by-name semantics; with distillation it is possible to produce a superlinear improvement. This extra power is obtained by performing more than one transformation pass over terms, as opposed to the single pass performed by positive supercompilation. We do not give a full description of the distillation algorithm here; details can be found in [8]. These details are not required to understand the remainder

```
implies (and (member y xs) (member z xs)) (leq y z)
where
implies = λx.λy. case x of
                   True  ⇒ y
                 | False ⇒ True
and     = λx.λy. case x of
                   True  ⇒ y
                 | False ⇒ False
member  = λy.λxs. case xs of
                    Nil       ⇒ False
                  | Cons x xs ⇒ case (eq x y) of
                                  True  ⇒ True
                                | False ⇒ member y xs
leq     = λx.λy. case x of
                   Zero   ⇒ True
                 | Succ x ⇒ case y of
                              Zero   ⇒ False
                            | Succ y ⇒ leq x y
eq      = λx.λy. case x of
                   Zero   ⇒ case y of
                              Zero   ⇒ True
                            | Succ y ⇒ False
                 | Succ x ⇒ case y of
                              Zero   ⇒ False
                            | Succ y ⇒ eq x y
```

Fig. 2. Example Program

of this paper; it is sufficient to know that the expressions resulting from distillation are in a specialised form which we call *distilled form*. The logical rules presented in this paper are defined over this distilled form without the need to know how expressions are converted into this form.

The transformation rules in distillation essentially perform normal-order reduction. *Folding* is performed when an expression is encountered which is an *instance* of a previously encountered expression, and *generalization* is performed to ensure termination of the transformation process. The terms which are compared before folding or generalizing in distillation are terms which have already been transformed; in positive supercompilation these will be untransformed terms. Generalization is performed when an expression is encountered within which a previously encountered expression is embedded. The form of embedding which we use to guide this generalization is the *homeomorphic* embedding relation which was derived from results by Higman [9] and Kruskal [18] and was defined within term rewriting systems [6] for detecting the possible divergence of the term rewriting process.

2.3 Distilled Form

Definition 2 (Distilled Form). *The expressions resulting from distillation are in* distilled form $dt^{\{\}}$*, where within an expression of the form* dt^{ρ}*,* ρ *denotes the set of all variables which have been introduced using* **let** *expressions, and cannot therefore appear in the selectors of* **case** *expressions. Distilled form* dt^{ρ} *is defined as shown in Fig. 3.* □

$$
\begin{aligned}
dt^{\rho} ::= {} & v_0\ v_1 \ldots v_n \\
| {} & c\ dt_1^{\rho} \ldots dt_n^{\rho} \\
| {} & \lambda v.dt^{\rho} \\
| {} & \textbf{case}\ v\ \textbf{of}\ p_1 \Rightarrow dt_1^{\rho} \mid \cdots \mid p_k \Rightarrow dt_k^{\rho}, \text{ where } v \notin \rho \\
| {} & \textbf{let}\ v\ =\ dt_0^{\rho}\ \textbf{in}\ dt_1^{(\rho \cup \{v\})} \\
| {} & \textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.dt^{\rho}\ \textbf{in}\ f\ v_1 \ldots v_n \\
| {} & f\ v_1 \ldots v_n
\end{aligned}
$$

Fig. 3. Distilled Form

In addition, at least one of the parameters in every function definition must be decreasing; all functions which do not have this property are replaced by $\bot$ during distillation, where $\bot$ is treated as a constructor in our language. Variables which are introduced using **let** expressions cannot appear in the selectors of **case** expressions. Expressions in distilled form therefore create no intermediate structures. This means that proofs over expressions which are in distilled form will require no intermediate lemmas, and can therefore be performed fully automatically. Programs in distilled form correspond to the 'read-only' primitive recursive programs as defined in [11]. The program defined in Fig. 2 is transformed by distillation into the program shown in Fig. 4. Note that the variables xs, y and z are also free within this program. We can see that all the intermediate structures have been eliminated from this program.

3 Theorem Proving Using Metasystem Transitions

In this section, we show how the theorem prover Poitín handles explicit quantification using metasystem transitions. To facilitate this, we add quantifiers of the form ALL $v.e$ and EX $v.e$ to our language, where the quantified variable v must be first-order. These quantifiers are defined over the three-valued logic with values $True$, $False$ and $\bot$. The universally quantified expression ALL $v.e$ has the value $True$ if the expression e has the value $True$ for all possible values of the quantified variable v, $False$ if e has the value $False$ for at least one value of v, and $\bot$ if e has the value $\bot$. The existentially quantified expression EX $v.e$ has the value $True$ if the expression e has the value $True$ for at least one value of the

```
case xs of
    Nil          ⇒ True
    Cons x xs ⇒
        letrec f =
        λx.λxs. case xs of
                    Nil          ⇒
                        letrec g =
                        λx.λy.λz. case x of
                                      Zero    ⇒ case y of
                                                    Zero    ⇒ True
                                                  | Succ y' ⇒ case z of
                                                                  Zero    ⇒ False
                                                                | Succ z' ⇒ True
                                      Succ x' ⇒
                                          case y of
                                              Zero    ⇒ False
                                            | Succ y' ⇒ case z of
                                                            Zero    ⇒ True
                                                          | Succ z' ⇒ g x' y' z'
                        in g x y z
                    Cons x' xs' ⇒ letrec h =
                                  λy.λz. case y of
                                             Zero    ⇒ f x xs'
                                           | Succ y ⇒ case z of
                                                          Zero    ⇒ f x' xs'
                                                        | Succ z ⇒ h y z
                                  in h x x'
        in f x xs
```

Fig. 4. Example Program Distilled

quantified variable v, $False$ if e has the value $False$ for all values of v, and $\perp$ if e has the value $\perp$. These quantifiers can be arbitrarily nested within an expression, provided that the expression is well-typed. The addition of these quantifiers into our language means that programs are no longer executable; however, the rules defined in this section show how these quantifiers can be eliminated to produce an executable program. We give the definitions of the sets of rules $\mathcal{A}$ for eliminating universal quantifiers and $\mathcal{E}$ for eliminating existential quantifiers which have been implemented in Poitín. More details of these rules, including examples of their application and a proof of their soundness, can be found in [15,12].

When a quantified expression is encountered by Poitín, the expression is first of all transformed by distillation. A metasystem transition is then used to apply inductive proof rules to the resulting distilled expression. If there are a number of nested quantifiers within the conjecture to be proved, then the proof rules are applied to the innermost quantified expression first. These inner

quantified expressions may contain free variables, which will be bound by another quantifier in some outer scope. The expression resulting from the application of the proof rules may therefore also contain free variables if these were present in the original expression. We therefore construct a hierarchy of metasystems in which the construction of each subsequent level is achieved by a metasystem transition. All quantifiers will be eliminated in the final resulting expression.

3.1 Rules for Universal Quantification

The rules for proving a universally quantified conjecture e are of the form $\mathcal{A}[\![e]\!]\ \rho\ \phi\ \sigma$ as shown in Fig. 5, where the parameter ρ is an environment mapping local variables to their values, ϕ is the set of previously encountered function calls and σ is the set of universally quantified variables. Note that these rules will only be applied to expressions which are in distilled form. Using these rules, the local variables contained within the domain of ρ and the universally quantified variables contained within σ are eliminated, and a simplified expression defined over the remaining free variables is obtained. If there are no free variables, then the input conjecture is reduced to a value in our three-valued logic.

In rule ($\mathcal{A}1$), if a local variable is encountered, then the value of this variable in the environment ρ is substituted for it and the resulting expression is further simplified using the proof rules. If a universally quantified variable is encountered, then since it must have a value in our three-valued logic, the value $False$ is returned as the variable cannot always have the value $True$. If a free variable is encountered, then it remains unchanged. In rule ($\mathcal{A}2$), if a constructor is encountered, then the value of this constructor is returned; this must be a value in our three-valued logic, since the input term is also of this type and contains no intermediate structures. In rule ($\mathcal{A}3$), if a λ-abstraction is encountered, then the body of the abstraction is further simplified. In rule ($\mathcal{A}4$), if we encounter a **case** expression then, since this expression must be in distilled form, the redex must be a non-local variable. If this variable is universally quantified, then a *case split* is performed in which we prove the current term separately for each of the possible values of the selector, and then return the conjunction of the resulting values. The different possible values of the selector are simply the patterns within the **case** expression. If the redex variable is not universally quantified, then it must be free, so it remains in the resulting term, and the proof rules are further applied to the branches of the **case** expression. In rule ($\mathcal{A}5$), if we encounter a **let** expression and none of the variables in the extracted expression are free, then the proof rules are applied to the extracted expression and the resulting value for the **let** variable is added to the environment ρ before applying the proof rules to the generalized expression. If the extracted expression contains free variables, then the proof rules are applied to each of the sub-expressions within the **let**. In rule ($\mathcal{A}6$), if we encounter a **letrec** function definition and none of the parameters in the initial application of this function are free, then this function application is a potential inductive hypothesis. Since at least one of these parameters must be decreasing, this parameter can be used as the *induction variable*. If we subsequently encounter a recursive call of this

$$\mathcal{A}[\![v_0\ v_1 \ldots v_n]\!]\ \rho\ \phi\ \sigma \qquad (\mathcal{A}1)$$
$$= \mathcal{A}[\![e[v_1/v'_1 \ldots v_n/v'_n]]\!]\ \rho\ \phi\ \sigma, \text{ if } \rho(v_0) = \lambda v'_1 \ldots v'_n.e$$
$$= False, \qquad \text{if } v_0 \in \sigma$$
$$= v_0\ v_1 \ldots v_n, \qquad \text{otherwise}$$

$$\mathcal{A}[\![c]\!]\ \rho\ \phi\ \sigma = c \qquad (\mathcal{A}2)$$

$$\mathcal{A}[\![\lambda v.e]\!]\ \rho\ \phi\ \sigma = \lambda v.\mathcal{A}[\![e]\!]\ \rho\ \phi\ \sigma \qquad (\mathcal{A}3)$$

$$\mathcal{A}[\![\textbf{case}\ v\ \textbf{of}\ p_1\ :\ e'_1 \mid \ldots \mid p_k\ :\ e'_k]\!]\ \rho\ \phi\ \sigma \qquad (\mathcal{A}4)$$
$$= (\mathcal{A}[\![e'_1]\!]\ \rho\ \phi\ \sigma_1)\ \wedge \ldots \wedge\ (\mathcal{A}[\![e'_k]\!]\ \rho\ \phi\ \sigma_k), \qquad \text{if } v \in \sigma$$
$$= \textbf{case}\ v\ \textbf{of}\ p_1\ :\ (\mathcal{A}[\![e'_1]\!]\ \rho\ \phi\ \sigma) \mid \ldots \mid p_k\ :\ (\mathcal{A}[\![e'_k]\!]\ \rho\ \phi\ \sigma), \text{ otherwise}$$
where
$\sigma_i = \sigma \cup FV(p_i)$

$$\mathcal{A}[\![\textbf{let}\ v\ =\ e_0\ \textbf{in}\ e_1]\!]\ \rho\ \phi\ \sigma \qquad (\mathcal{A}5)$$
$$= \mathcal{A}[\![e_1]\!]\ \rho[(\mathcal{A}[\![e_0]\!]\ \rho\ \phi\ \sigma)/v]\ \phi\ \sigma, \qquad \text{if } FV(e_0) \subseteq (dom(\rho) \cup \sigma)$$
$$= \textbf{let}\ v\ =\ (\mathcal{A}[\![e_0]\!]\ \rho\ \phi\ \sigma)\ \textbf{in}\ (\mathcal{A}[\![e_1]\!]\ \rho\ \phi\ \sigma), \text{ otherwise}$$

$$\mathcal{A}[\![\textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.e_0\ \textbf{in}\ f\ v'_1 \ldots v'_n]\!]\ \rho\ \phi\ \sigma \qquad (\mathcal{A}6)$$
$$= e'_0, \qquad \text{if } \{v'_1 \ldots v'_n\} \subseteq (dom(\rho) \cup \sigma)$$
$$= \textbf{letrec}\ f\ =\ \lambda v''_1 \ldots v''_k.e'_0\ \textbf{in}\ f\ v''_1 \ldots v''_k, \text{ otherwise}$$
where
$e'_0 = \mathcal{A}[\![e_0]\!]\ \rho\ (\phi \cup \{f\ v'_1 \ldots v'_n\})\ \sigma$
$\{v''_1 \ldots v''_k\} = \{v'_1 \ldots v'_n\} \setminus (dom(\rho) \cup \sigma)$

$$\mathcal{A}[\![f\ v_1 \ldots v_n]\!]\ \rho\ \phi\ \sigma \qquad (\mathcal{A}7)$$
$$= True, \qquad \text{if } \{v'_1 \ldots v'_n\} \subseteq (dom(\rho) \cup \sigma)$$
$$= (f\ v''_1 \ldots v''_k)[v_1/v'_1 \ldots v_n/v'_n], \text{ otherwise}$$
where
$(f\ v'_1 \ldots v'_n) \in \phi$
$\{v''_1 \ldots v''_k\} = \{v'_1 \ldots v'_n\} \setminus (dom(\rho) \cup \sigma)$

Fig. 5. Proof Rules for Universal Quantification

function in rule ($\mathcal{A}7$), then we have re-encountered this inductive hypothesis, so the value *True* is returned. If a function definition contains free variables, then the function is re-defined over these free variables.

As an example of the application of these rules, consider the expression ALL $z.e$ where e is the program defined in Fig. 2. We first of all apply distillation to the expression e, yielding the distilled expression e' as shown in Fig. 4. We then apply the universal proof rules $\mathcal{A}[\![e']\!]\ \{\}\ \{\}\ \{z\}$ for the universal variable z giving the program shown in Fig. 6. We can see that the variable z has been eliminated, and the variables xs and u are still free within the resulting program.

```
case xs of
    Nil          ⇒ True
    Cons x xs ⇒
        letrec f =
        λx.λxs. case xs of
                    Nil          ⇒ letrec g =
                                     λx.λy. case x of
                                                Zero   ⇒ case y of
                                                             Zero   ⇒ True
                                                           | Succ y' ⇒ False
                                              | Succ x' ⇒ case y of
                                                             Zero   ⇒ False
                                                           | Succ y' ⇒ g x' y'
                                     in g x y
                  | Cons x' xs' ⇒ letrec h =
                                     λy.λz. case y of
                                                Zero   ⇒ f x xs'
                                              | Succ y ⇒ case z of
                                                           Zero   ⇒ f x' xs'
                                                         | Succ z ⇒ h y z
                                     in h x x'
        in f x xs
```

Fig. 6. Program Resulting From Application of Universal Proof Rules

3.2 Rules for Existential Quantification

The rules for proving an existentially quantified conjecture e are of the form $\mathcal{E}[\![e]\!]\ \rho\ \phi\ \sigma$ as shown in Fig. 7, where the parameter ρ is an environment mapping local variables to their values, ϕ is the set of previously encountered function calls and σ is the set of existentially quantified variables. Using these rules, the local variables contained within the domain of ρ and the existentially quantified variables contained within σ are eliminated, and a simplified expression over the remaining free variables is obtained.

The rules are similar to those for universal quantification, with the only major differences being in rules ($\mathcal{E}1$), ($\mathcal{E}4$), ($\mathcal{E}6$) and ($\mathcal{E}7$). In rule ($\mathcal{E}1$), if an existentially quantified variable is encountered, then since it must have a value in our three-valued logic, the value *True* is returned as the value of the variable can be $True$. In rule ($\mathcal{E}4$), if the redex in a **case** expression is an existentially quantified variable, then we also perform a *case split* and prove the current term separately for each of the possible values of the selector, but in this instance we return the disjunction of the resulting values. In rules ($\mathcal{E}6$) and ($\mathcal{E}7$), function applications are no longer possible inductive hypotheses as they contain existential variables. However, if none of the parameters in a function application are

$$\mathcal{E}[\![v_0\ v_1 \ldots v_n]\!]\ \rho\ \phi\ \sigma \qquad (\mathcal{E}1)$$
$$= \mathcal{E}[\![e[v_1/v'_1 \ldots v_n/v'_n]]\!]\ \rho\ \phi\ \sigma, \text{ if } \rho(v_0) = \lambda v'_1 \ldots v'_n.e$$
$$= True, \qquad \text{if } v_0 \in \sigma$$
$$= v_0\ v_1 \ldots v_n, \qquad \text{otherwise}$$

$$\mathcal{E}[\![c]\!]\ \rho\ \phi\ \sigma = c \qquad (\mathcal{E}2)$$

$$\mathcal{E}[\![\lambda v.e]\!]\ \rho\ \phi\ \sigma = \lambda v.\mathcal{E}[\![e]\!]\ \rho\ \phi\ \sigma \qquad (\mathcal{E}3)$$

$$\mathcal{E}[\![\textbf{case}\ v\ \textbf{of}\ p_1\ :\ e'_1 \mid \ldots \mid p_k\ :\ e'_k]\!]\ \rho\ \phi\ \sigma \qquad (\mathcal{E}4)$$
$$= (\mathcal{E}[\![e'_1]\!]\ \rho\ \phi\ \sigma_1)\ \vee \ldots \vee\ (\mathcal{E}[\![e'_k]\!]\ \rho\ \phi\ \sigma_k), \qquad \text{if } v \in \sigma$$
$$= \textbf{case}\ v\ \textbf{of}\ p_1\ :\ (\mathcal{E}[\![e'_1]\!]\ \rho\ \phi\ \sigma) \mid \ldots \mid p_k\ :\ (\mathcal{E}[\![e'_k]\!]\ \rho\ \phi\ \sigma), \text{ otherwise}$$
where
$\sigma_i = \sigma \cup FV(p_i)$

$$\mathcal{E}[\![\textbf{let}\ v\ =\ e_0\ \textbf{in}\ e_1]\!]\ \rho\ \phi\ \sigma \qquad (\mathcal{E}5)$$
$$= \mathcal{E}[\![e_1]\!]\ \rho[(\mathcal{E}[\![e_0]\!]\ \rho\ \phi\ \sigma)/v]\ \phi\ \sigma, \qquad \text{if } FV(e_0) \subseteq (dom(\rho) \cup \sigma)$$
$$= \textbf{let}\ v\ =\ (\mathcal{E}[\![e_0]\!]\ \rho\ \phi\ \sigma)\ \textbf{in}\ (\mathcal{E}[\![e_1]\!]\ \rho\ \phi\ \sigma), \text{ otherwise}$$

$$\mathcal{E}[\![\textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.e_0\ \textbf{in}\ f\ v'_1 \ldots v'_n]\!]\ \rho\ \phi\ \sigma \qquad (\mathcal{E}6)$$
$$= e'_0, \qquad \text{if } \{v'_1 \ldots v'_n\} \subseteq (dom(\rho) \cup \sigma)$$
$$= \textbf{letrec}\ f\ =\ \lambda v''_1 \ldots v''_k.e'_0\ \textbf{in}\ f\ v''_1 \ldots v''_k, \text{ otherwise}$$
where
$e'_0 = \mathcal{E}[\![e_0]\!]\ \rho\ (\phi \cup \{f\ v'_1 \ldots v'_n\})\ \sigma$
$\{v''_1 \ldots v''_k\} = \{v'_1 \ldots v'_n\} \setminus (dom(\rho) \cup \sigma)$

$$\mathcal{E}[\![f\ v_1 \ldots v_n]\!]\ \rho\ \phi\ \sigma \qquad (\mathcal{E}7)$$
$$= False, \qquad \text{if } \{v'_1 \ldots v'_n\} \subseteq (dom(\rho) \cup \sigma)$$
$$= (f\ v''_1 \ldots v''_k)[v_1/v'_1 \ldots v_n/v'_n], \text{ otherwise}$$
where
$(f\ v'_1 \ldots v'_n) \in \phi$
$\{v''_1 \ldots v''_k\} = \{v'_1 \ldots v'_n\} \setminus (dom(\rho) \cup \sigma)$

Fig. 7. Proof Rules for Existential Quantification

free then the value *False* is returned as we know that the search space of the existential variables has been exhausted.

As an example of the application of these rules, consider the proof of the conjecture ALL xs.EX y.ALL $z.e$ where e is the program defined in Fig. 2. We first of all apply distillation to the expression e, yielding the distilled expression e' as shown in Fig. 4. We then apply the universal proof rules as shown above to the expression ALL $z.e'$, giving the expression e'' shown in Fig. 6. The existential proof rules $\mathcal{E}[\![e'']\!]\ \{\}\ \{\}\ \{y\}$ are then applied for the existential variable y giving the program shown in Fig. 8.

```
case xs of
    Nil          ⇒ True
    Cons x xs ⇒
        letrec f =
        λx.λxs. case xs of
                    Nil          ⇒ letrec g =
                                   λv. case v of
                                           Zero   ⇒ True
                                         | Succ v ⇒ g v
                                   in g x
                  | Cons x' xs' ⇒ letrec h =
                                   λy.λz. case y of
                                              Zero   ⇒ f x xs'
                                            | Succ y ⇒ case z of
                                                          Zero   ⇒ f x' xs'
                                                        | Succ z ⇒ h y z
                                   in h x x'
        in f x xs
```

Fig. 8. Program Resulting From Application of Existential Proof Rules

We can see that the variable y has been eliminated and that the variable xs is still free. We then apply the universal proof rules $\mathcal{A}[\![e'''']\!]\ \{\}\ \{\}\ \{xs\}$ where e''' is the expression shown in Fig. 8 and xs is the universally quantified variable, giving the value *True* as required.

3.3 Soundness and Relative Completeness

In this section, we consider the soundness and relative completeness of our theorem prover. Full details of the proofs of these properties can be found in [15,12]; we do not include these here. To facilitate these proofs, sequent calculus rules are defined for the distilled form of conjecture which is input to our theorem prover. Note that there is no need for a cut rule as all the intermediate structures in the input conjecture will have been eliminated.

Our proof rules are proved to be sound by showing that all conjectures in distilled form which are found to have the value $True$ using our proof rules can also be proved using the sequent calculus rules.

Theorem 1 (Soundness of Universal Proof Rules).
$\mathcal{A}[\![e]\!]\ \{\}\ \phi\ \{v_1 \ldots v_n\}\ =\ True\ \wedge\ e \in dt^{\{\}}\ \Rightarrow\ \phi \vdash ALL\ v_1 \ldots v_n.e$ □

Theorem 2 (Soundness of Existential Proof Rules).
$\mathcal{E}[\![e]\!]\ \{\}\ \phi\ \{v_1 \ldots v_n\}\ =\ True\ \wedge\ e \in dt^{\{\}}\ \Rightarrow\ \phi \vdash EX\ v_1 \ldots v_n.e$ □

Our proof rules are proved to be complete for all conjectures which are in distilled form by showing that all conjectures in distilled form which can be proved using the sequent calculus rules also have the value $True$ using our proof rules.

Theorem 3 (Relative Completeness of Universal Proof Rules).
$\phi \vdash ALL\ v_1 \ldots v_n.e\ \wedge\ e \in dt^{\{\}}\ \Rightarrow\ \mathcal{A}[\![e]\!]\ \{\}\ \phi\ \{v_1 \ldots v_n\}\ =\ True$ □

Theorem 4 (Relative Completeness of Existential Proof Rules).
$\phi \vdash EX\ v_1 \ldots v_n.e\ \wedge\ e \in dt^{\{\}}\ \Rightarrow\ \mathcal{E}[\![e]\!]\ \{\}\ \phi\ \{v_1 \ldots v_n\}\ =\ True$ □

The proofs of each of these theorems are by recursion induction on the rules $\mathcal{A}$ and $\mathcal{E}$. Details of the proofs can be found in [15,12].

4 Program Construction Using Metasystem Transitions

In this section, we present our novel technique for the construction of programs from specifications. The constructed programs essentially compute the existential witness of the proof of their corresponding specification. To facilitate this, we add specifications of the form ANY $v.e$ to our language, where the quantified variable v must be first-order. The specification ANY $v.e$ can have any value of the quantified variable v for which the expression e has the value $True$; if no such value exists, then it has the undefined value $\perp$. The variable v is therefore implicitly existentially quantified within e, but the ANY quantifier differs from the existential quantifier EX in that it has the same type as the variable v, rather than being a vlaue in our three-valued logic. When a specification ANY $v.e$ is encountered by Poitín, the quantified expression e is first of all transformed using distillation. A metasystem transition is then used to apply the rules for program construction to the resulting distilled expression, thus constructing an executable program form a non-executable specification.

4.1 Rules for Program Construction

The program construction rules are defined by $\mathcal{C}[\![e]\!]\ [\![e']\!]\ \phi\ \sigma$ as shown in Fig. 9, where the expression e is the distilled specification, e' is the existential witness, ϕ is the set of the previously encountered function calls and σ is the set of universally quantified variables. For a specification ANY $v.e$, the quantified variable v is passed as the initial existential witness and the free variables in the specification are passed as the initial set of universally quantified variables.

In rule ($\mathcal{C}1$), if a universally quantified variable is encountered, then since it must be a value in our three-valued logic, the value $\perp$ is returned as the variable cannot have the value $True$. Otherwise, the value of the variable is returned unchanged. In rule ($\mathcal{C}2$), if we encounter a constructor, then if this constructor is *True* and the existential witness is fully instantiated, the value of the existential witness is returned. Otherwise, the value $\perp$ is returned. In rule ($\mathcal{C}3$), if a λ-abstraction is encountered, then the program construction rules are applied to the body of the abstraction. In rule ($\mathcal{C}4$), if we encounter a **case** expression then, since this expression must be in distilled form, the redex must be a non-local variable. If this variable is universally quantified, then it remains within the expression and the program construction rules are further applied to the branches

$$\mathcal{C}[\![v_0\ v_1 \ldots v_n]\!][\![e]\!]\ \phi\ \sigma\ = \bot, \qquad \text{if } v_0 \in \sigma \tag{$\mathcal{C}1$}$$
$$= v_0\ v_1 \ldots v_n, \text{ otherwise}$$

$$\mathcal{C}[\![c]\!][\![e]\!]\ \phi\ \sigma\ = e, \text{ if } c = True \wedge FV(e) = \{\} \tag{$\mathcal{C}2$}$$
$$= \bot, \text{otherwise}$$

$$\mathcal{C}[\![\lambda v.e]\!][\![e']\!]\ \phi\ \sigma = \lambda v.\mathcal{C}[\![e]\!][\![e']\!]\ \phi\ \sigma \tag{$\mathcal{C}3$}$$

$$\mathcal{C}[\![\textbf{case } v \textbf{ of } p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n]\!][\![e]\!]\ \phi\ \sigma \tag{$\mathcal{C}4$}$$
$$= \textbf{case } v \textbf{ of } p_1 \Rightarrow (\mathcal{C}[\![e_1]\!][\![e]\!]\ \phi\ \sigma_1) \mid \cdots \mid p_n \Rightarrow (\mathcal{C}[\![e_n]\!][\![e]\!]\ \phi\ \sigma_n), \text{ if } v \in \sigma$$
$$= (\mathcal{C}[\![e_1]\!][\![e[p_1/v]]\!]\ \phi\ \sigma)\ \sqcup \ldots \sqcup\ (\mathcal{C}[\![e_n]\!][\![e[p_n/v]]\!]\ \phi\ \sigma), \qquad \text{otherwise}$$
where
$\sigma_i = \sigma \cup FV(p_i)$

$$\mathcal{C}[\![\textbf{let } v = e_0 \textbf{ in } e_1]\!][\![e]\!]\ \phi\ \sigma = \textbf{let } v = (\mathcal{C}[\![e_0]\!][\![e]\!]\ \phi\ \sigma) \textbf{ in } (\mathcal{C}[\![e_1]\!][\![e]\!]\ \phi\ \sigma) \tag{$\mathcal{C}5$}$$

$$\mathcal{C}[\![\textbf{letrec } f\ =\ \lambda v_1 \ldots v_n.e_0 \textbf{ in } f\ v'_1 \ldots v'_n]\!][\![v]\!]\ \phi\ \sigma \tag{$\mathcal{C}6$}$$
$$= e'_0, \qquad \text{if } \{v'_1 \ldots v'_n\} \cap \sigma = \{\}$$
$$= \textbf{letrec } f\ =\ \lambda v''_1 \ldots v''_k.e'_0 \textbf{ in } f\ v''_1 \ldots v''_k, \text{ otherwise}$$
where
$e'_0 = \mathcal{C}[\![e_0]\!][\![v]\!]\ (\phi \cup \{f\ v'_1 \ldots v'_n\})\ \sigma$
$\{v''_1 \ldots v''_k\} = \{v'_1 \ldots v'_n\} \cap \sigma$

$$\mathcal{C}[\![\textbf{letrec } f\ =\ \lambda v_1 \ldots v_n.e_0 \textbf{ in } f\ v'_1 \ldots v'_n]\!][\![c\ e_1 \ldots e_k]\!]\ \phi\ \sigma \tag{$\mathcal{C}7$}$$
$$= \mathcal{C}[\![e_0]\!][\![c\ e_1 \ldots e_k]\!]\ (\phi \cup \{f\ v'_1 \ldots v'_n\})\ \sigma, \qquad \text{if } \{v'_1 \ldots v'_n\} \cap \sigma = \{\}$$
$$= c\ (\mathcal{C}[\![\textbf{letrec } f\ =\ \lambda v_1 \ldots v_n.e_0 \textbf{ in } f\ v'_1 \ldots v'_n]\!][\![e_1]\!]\ \phi\ \sigma) \qquad \text{otherwise}$$
$$\ldots (\mathcal{C}[\![\textbf{letrec } f\ =\ \lambda v_1 \ldots v_n.e_0 \textbf{ in } f\ v'_1 \ldots v'_n]\!][\![e_k]\!]\ \phi\ \sigma),$$

$$\mathcal{C}[\![f\ v_1 \ldots v_n]\!][\![v]\!]\ \phi\ \sigma \tag{$\mathcal{C}8$}$$
$$= \bot, \qquad \text{if } \{v'_1 \ldots v'_n\} \cap \sigma = \{\}$$
$$= f\ v''_1 \ldots v''_k[v_1/v'_1 \ldots v_n/v'_n], \text{ otherwise}$$
where
$(f\ v'_1 \ldots v'_n) \in \phi$
$\{v''_1 \ldots v''_k\} = \{v'_1 \ldots v'_n\} \cap \sigma$

$$\mathcal{C}[\![f\ v_1 \ldots v_n]\!][\![c\ e_1 \ldots e_k]\!]\ \phi\ \sigma \tag{$\mathcal{C}9$}$$
$$= \bot, \qquad \text{if } \{v'_1 \ldots v'_n\} \cap \sigma = \{\}$$
$$= c\ (\mathcal{C}[\![f\ v_1 \ldots v_n]\!][\![e_1]\!]\ \phi\ \sigma) \ldots (\mathcal{C}[\![f\ v_1 \ldots v_n]\!][\![e_k]\!]\ \phi\ \sigma), \text{ otherwise}$$

Fig. 9. Rules for Program Construction

of the **case** expression. Before transforming each branch, the corresponding pattern variables are added to σ as they are also implicitly universally quantified. If the selector is existentially quantified, existential witnesses are constructed for each of the branches separately. These witnesses will be constructed using the corresponding patterns which give the value of the selector within the branch.

The existential witness for the overall expression is then given by the least upper bound of these existential witnesses for each branch (the least upper bound operator $\sqcup$ is defined separately for each of the data types in our language). In rule ($\mathcal{C}5$), if we encounter a **let** expression, then the program construction rules are applied to each of the sub-expressions contained within it. In rules ($\mathcal{C}6$)-($\mathcal{C}9$), if we encounter a recursive function call, then this is simplified to be defined over only the universal variables of this call. If the recursive call does not contain any universally quantified variables, then the value $\perp$ is returned as the search space of the existential variables has been exhausted.

4.2 Example

In this section, we give an example of the application of our program construction rules. Consider the construction of a program from the specification ANY y. ALL $z.e$ where e is the program defined in Fig. 2. We first of all apply distillation to the expression e, yielding the distilled expression e' as shown in Figure 4. We then apply the universal proof rules as shown previously to the expression ALL $z.e'$, giving the expression e'' as shown in Fig. 6. We then apply the program construction rules $\mathcal{C}[\![e'']\!][\![y]\!]\ \{\}\ \{xs\}$ where y is the existential witness and xs is the only free variable in e'', giving the program shown in Fig. 10.

```
case xs of
   Nil          ⇒ ⊥
   Cons x xs ⇒
      letrec f =
      λx.λxs. case xs of
                  Nil          ⇒ letrec g =
                                   λv. case v of
                                          Zero   ⇒ Zero
                                        | Succ v ⇒ Succ (g v)
                                   in g x
                | Cons x' xs' ⇒ letrec h =
                                   λy.λz. case y of
                                             Zero   ⇒ f x xs'
                                           | Succ y ⇒ case z of
                                                        Zero   ⇒ f x' xs'
                                                      | Succ z ⇒ h y z
                                   in h x x'
      in f x xs
```

Fig. 10. Program Resulting From Application of Program Construction Rules

From this, we can see that we have generated a program for finding the smallest element in the list xs fully automatically from the specification, and that this program creates no intermediate data structures.

4.3 Correctness of Constructed Programs

In order to prove that the programs constructed by our program construction rules are correct with respect to the original specification ANY $v.e$ we need to show that when the constructed existential witnesses are substituted for the existential variables in the specification, then the specification statement has the value $True$.

Theorem 5 (Correctness of Constructed Programs).
$\mathcal{C}[\![e]\!][\![e']\!]\ \phi\ \sigma = e'[e_1/v_1 \ldots e_n/v_n] \Rightarrow \mathcal{A}[\![e[e_1/v_1 \ldots e_n/v_n]]\!]\ \{\}\ \phi\ \sigma = True$
where $\{v_1 \ldots v_n\} = FV(e')$ □

The proof is by recursion induction on the rules $\mathcal{C}$. Full details of the proof can be found in [13,12].

5 Conclusion and Related Work

In this paper, we have shown how metasystem transitions can be used in conjunction with the distillation transformation algorithm to prove a wide range of theorems in first-order logic fully automatically. The programs generated by distillation are in distilled form; they create no intermediate structures, and correspond to the 'read-only' primitive recursive programs [11], within which data can be higher-order. As shown in [11], this form can be used to encode all problems which belong to the fairly wide class of *elementary* sets. We have shown that our theorem prover is sound and complete for theorems which are in this form. We then showed how programs can be constructed fully automatically from specifications which are in this form through the use of metasystem transitions. We have given an example of the application of our approach, and proved that the constructed programs are totally correct with respect to their specifications.

The most closely related work to that presented here is Turchin's work on supercompilation [25]. Turchin has shown how metasystem transitions can be used in conjunction with supercompilation to prove explicitly quantified conjectures [26], but has not shown how programs can be constructed from their specifications using metasystem transitions. To our knowledge, the techniques described in this paper are the first which have been developed for the automatic construction of programs from their specifications using metasystem transitions. Distillation is more powerful than positive supercompilation [7], removing more intermediate structures. The presence of more intermediate structures implies the need for more intermediate lemmas when theorem proving. The set of theorems which can be proved fully automatically using positive supercompilation is therefore a subset of those which can be proved fully automatically using distillation.

The transformation of logic programs can be regarded as the construction of programs from specifications which contain implicit existential quantification. There has been a considerable amount of work on the use of logic program transformation for inductive theorem proving (for example, [22,23,19]). Many of these techniques are not fully automatic, and their fully automated components are of similar power to supercompilation, so they will not be able to prove as many theorems fully automatically as the technique described in this paper.

A wide range of inductive theorem proving systems have been developed (for example, NQTHM [4], CLAM [5], INKA [3], RRL [16]), but these tend to concentrate mainly on universal quantification, and therefore cannot be used for program synthesis. The inclusion of existential quantification is very problematic and greatly complicates the theorem proving process. Some techniques which have been developed for program synthesis from non-executable specifications include *constructive synthesis*, *deductive synthesis* and *middle-out reasoning*.

Constructive synthesis (e.g. [2]) is based on the Curry-Howard isomorphism [10] and uses the proof-as-programs principle. In this approach, a proof is constructed in a constructive type theory such as that of Martin-Löf [21]. There is a one-to-one relationship between this constructive proof and the corresponding program, which can be easily extracted from the proof. Deductive synthesis (e.g. [20]) attempts to derive an executable program from a high level specification by applying rules of inference. For example, the approach of Manna and Waldinger [20] incorporates ideas from resolution and inductive theorem proving as rules of inference. Middle-out reasoning (e.g. [17]) represents undefined functions in the synthesis conjecture as meta-variables. These meta-variables are instantiated gradually as the subsequent proof takes place. When the proof is complete, the meta-variables should be instantiated to the correct corresponding program.

The programs constructed using the above techniques can often be quite inefficient. The programs constructed using our technique will construct no intermediate structures and should therefore be more efficient. Also, none of the above techniques for program construction are fully automatic and may therefore require user guidance. Although it may be argued that the additional lemmas which are required using these techniques can themselves be automated, the techniques can never be fully automatic as it will never be possible to encode all possible lemmas within them. However, it will of course be possible to construct some programs using these techniques which cannot be constructed using the technique described in this paper. Research is still continuing on determining the class of specifications which can be transformed by distillation into distilled form to allow programs to be constructed automatically from them using our technique.

References

1. L. Augustsson. Compiling Pattern Matching. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 368–381. Springer-Verlag, 1985.

2. J.L. Bates and R.L. Constable. Proofs as Programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.
3. S. Biundo, B. Hummel, D. Hutter, and C. Walther. The Karlrsruhe Induction Theorem Proving System. *Lecture Notes in Computer Science*, 230:672–674, 1987.
4. R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979.
5. Alan Bundy, Frank Van Harmelen, Christian Horn, and Alan Smaill. The Oyster-CLAM System. In *Proceedings of the 10th International Conference on Automated Deduction*, pages 647–648, 1990.
6. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier, MIT Press, 1990.
7. Robert Glück and Morten Heine Sørensen. A Roadmap to Metacomputation by Supercompilation. In *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 137–160. Springer-Verlag, 1996.
8. G.W. Hamilton. Distillation: Extracting the Essence of Programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 61–70, 2007.
9. G. Higman. Ordering by Divisibility in Abstract Algebras. *Proceedings of the London Mathemtical Society*, 2:326–336, 1952.
10. W.A. Howard. The Formulae-as-Types Notion of Construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.
11. N.D. Jones. The Expressive Power of Higher-Order Types or, Life Without CONS. *Journal of Functional Programming*, 11(1):55–94, January 2001.
12. M.H. Kabir. *Automatic Inductive Theorem Proving and Program Construction Methods Using Program Transformation*. PhD thesis, School of Computing, Dublin City University, October 2007.
13. M.H. Kabir and G.W. Hamilton. Constructing Programs From Metasystem Transition Proofs. Working Paper CA07-02, School of Computing, Dublin City University, 2007.
14. M.H. Kabir and G.W. Hamilton. Extending Poitín to Handle Explicit Quantification. In *Proceedings of the Sixth International Workshop on First-Order Theorem Proving*, pages 20–34, 2007.
15. M.H. Kabir and G.W. Hamilton. Extending Poitín to Handle Explicit Quantification. Working Paper CA07-01, School of Computing, Dublin City University, 2007.
16. Deepak Kapur, G. Sivakumar, and Hantao Zhang. RRL: A Rewrite Rule Laboratory. *Lecture Notes in Computer Science*, 230:691–692, 1986.
17. I. Kraan, D. Basin, and A. Bundy. Middle-Out Reasoning For Synthesis and Induction. *Journal of Automated Reasoning*, 16(1–2):113–145, 1996.
18. J.B. Kruskal. Well-Quasi Ordering, the Tree Theorem, and Vazsonyi's Conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.
19. Helko Lehmann and Michael Leuschel. Inductive Theorem Proving by Program Specialisation: Generating Proofs for Isabelle Using Ecce. In *13th International Symposium on Logic Based Program Synthesis and Transformation*, pages 1–19, 2003.
20. Z. Manna and R. Waldinger. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, January 1980.
21. P. Martin-Löf. Constructive Mathematics and Computer Programming. *Logic, Methodology and Philosophy of Science*, VI:153–175, 1980.

22. Alberto Pettorossi and Maurizio Proietti. Synthesis and Transformation of Logic Programs Using Unfold/Fold Proofs. *Journal of Logic Programming*, 41(2–3):197–230, 1999.
23. Abhik Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. Proofs by Program Transformation. In *International Symposium on Logic Based Program Synthesis and Transformation*, 1999.
24. V.F. Turchin. The Use of Metasystem Transition in Theorem Proving and Program Optimization. *Lecture Notes in Computer Science*, 85:645 – 657, 1980.
25. V.F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):90–121, July 1986.
26. V.F. Turchin. Metacomputation: Metasystem Transitions plus Supercompilation. *Lecture Notes in Computer Science*, 1110:481–509, 1996.
27. P. Wadler. Efficient Compilation of Pattern Matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pages 78–103. Prentice Hall, 1987.