Supporting ArcAngel in ProofPower

Frank Zeyda[a,1], Marcel Oliveira[b] and Ana Cavalcanti[a]

[a] *Department of Computer Science, University of York, York, YO10 5DD, UK*

[b] *Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte, Brazil*

Abstract

ArcAngel is a specialised tactic language devised to facilitate and automate program developments using Morgan's refinement calculus. It is especially well-suited for the specification of high-level strategies to derive programs by construction, and equipped with a formal semantics that enables reasoning about tactics. In this paper, we present an implementation of ArcAngel for the ProofPower theorem prover. We discuss the underlying design, explain how it implements the semantics of ArcAngel, and examine differences in expressiveness and flexibility in comparison to ProofPower's in-built tactic language. ArcAngel supports backtracking through angelic choice; this is beyond the basic capabilities of ProofPower and many other main-stream theorem provers. The implementation is demonstrated with a non-trivial tactic example.

Keywords: tactic language; proof automation; refinement; Z

1 Introduction

Morgan's refinement calculus [10] supports the derivation of programs from specifications. It incorporates the constructs of Dijkstra's guarded command language and adds a specification statement $w : [pre, post]$ which captures the behaviour of a program that can update the variables in the list w (the frame), and has precondition pre and postcondition $post$. Specifications are transformed into executable programs by a series of correctness-preserving refinement steps, and each step is justified by the application of a law within the calculus. This guarantees by construction that the concrete refinement correctly implements its abstract specification.

To automate recurring sequences of derivation steps in the refinement of program specifications, the ArcAngel tactic language was proposed [12]. It has a formal semantics and an extensive set of algebraic laws that support transformation and reasoning about tactics. ArcAngel is itself an extension of Angel [6,8], owing its name to the angelic resolution of nondeterminism in the process of solving proof goals. Whereas Angel is a general-purpose tactic language, ArcAngel specifically targets

[1] Email: zeyda@cs.york.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Law	Definition
$\mathsf{expandFrame}(x)$	$w : [pre, post] \;=\; w, x : [pre, post \wedge x = x_0]$
$\mathsf{followAssign}(x, E)$	$w, x : [pre, post] \;\sqsubseteq\; w, x : [pre, post[x \backslash E]] \,;\, x := E$

Fig. 1. Two examples of elementary refinement laws.

transformation of programs. We can think of ArcAngel tactics as procedures for rewriting program expressions in Morgan's calculus.

ArcAngel tactics are written in a notation that supports the application of primitive refinement laws, as well as various operators to combine tactics; the latter are often called tacticals. Two fundamental tactic combinators are alternation $t_1 \mid t_2$ and sequential composition $t_1 \,;\, t_2$. Alternation first attempts to apply t_1, and if this leads to failure at any point, applies t_2. Sequential composition applies the tactics t_1 and t_2 in sequence. In general, failure might occur for a number of reasons, one of them is a primitive law not being applicable to a program.

An important feature of the alternating choice in ArcAngel is that it is *angelic*, in that it always finds a successful execution, if there is one, by making the right choices. As an example, we consider the following tactic.

$$\mathsf{robustFollowAssign}(x, E) \;\widehat{=}\; (\mathbf{skip} \mid \mathbf{law}\, \mathsf{expandFrame}(x)) \,;\, \mathbf{law}\, \mathsf{followAssign}(x, E)$$

The $\mathbf{law}\, \mathsf{name}(args)$ construct invokes a parametrised refinement law name passing the list of arguments *args*. Here, we use refinement laws from [10]: $\mathsf{expandFrame}(x)$ extends the frame of a specification with a variable x, and $\mathsf{followAssign}(x, E)$ refines a specification statement into a sequence composed of a specification statement followed by the assignment $x := E$. The definition of the laws is in Fig.1. The **skip** tactic always succeeds and does not alter the program.

We observe that $\mathsf{followAssign}(x, E)$ can only be applied if the program is of the form $w, x : [pre, post]$. The tactic $\mathsf{robustFollowAssign}(x, E)$ above, on the other hand, may be successfully applied to specification statements that may not have x in their frame. Operationally, this results in $\mathsf{followAssign}(x, E)$ failing after the first choice **skip** is taken, and the execution backtracking to explore the second choice of the alternation in attempting to find a successful path of continuation.

The angelic nondeterminism embedded in the choice supports a concise and general description of many robust tactics of refinement and proof. Several examples are presented in [12]. The implicit backtracking manifests itself in the following law which is valid for ArcAngel (and Angel): $(t_1 \mid t_2) \,;\, t_3 = (t_1 \,;\, t_3) \mid (t_2 \,;\, t_3)$.

The Ergo theorem prover [16], which is implemented in Prolog, was extended to use Angel as a tactic language [9]. ArcAngel is incorporated in the refinement editor REFINE [14], which however does not provide facilities for theorem proving. In this paper, we present an implementation of ArcAngel in ProofPower, a flexible and extensible theorem prover based on HOL. It has an open architecture and has been successfully used on industrial-scale projects [1]. ProofPower also provides an embedding and formalisation of the Z language. This is useful in defining a semantic model for Morgan's refinement calculus as part of the ArcAngel implementation.

To encode the ArcAngel tactic above in ProofPower, we could try and use the tactical `ORELSE` to represent alternation, and `THEN` to represent sequencing. Even

so $(t_1$ `ORELSE` $t_2)$ `THEN` t_3 would not have the same operational behaviour as the ArcAngel tactic $(t_1 \mid t_2)\,;\, t_3$. The ProofPower tactic would first apply t_1, and if this fails resolve to applying t_2. The choice of either applying t_1 or t_2, however, is not revised if t_3 subsequently turns out to fail. In general, `ORELSE` acts like a cut on alternation; the choice it makes is not a provisional one unless t_1 on its own fails, in which case t_2 is executed. Similar limitations to backtracking also exist in LCF and HOL. PVS, on the other hand, does provide in-built support for backtracking tactics via its *try* tactical, but its semantics is more difficult to describe (and, consequently, more difficult to use) as failure and backtracking are treated as distinct outcomes of tactic applications; this becomes apparent, for example, in [5].

A second limitation that we overcome in our implementation is that conventionally tactics in ProofPower, like in many other theorem provers, apply to goals (sequents), namely pairs of assumptions and conclusions. The purpose of ArcAngel tactics, on the other hand, is to transform program *expressions*. In this paper we describe how we support the use of ArcAngel tactics, which is independent of the ProofPower tactic language. Our implementation is very close to the formal semantics of ArcAngel, increasing confidence in its correctness.

We also address several issues that led to a generalisation and in parts unification of ArcAngel with its kin Angel, as well as the more specialised derivate ArcAngel$C$, a variant tailored for refinement in the *Circus* language [11]. Whereas Angel deals with proving general theorems, ArcAngel(C) are methods for constructive refinement. A by-product of this unification is a framework that fosters the development of other derivatives of Angel, and we explain how we support their embedding.

The structure of the paper is as follows. In Section 2 we introduce the relevant preliminary material; more specifically, we give a brief account of the syntax and semantics of ArcAngel, and the ProofPower theorem prover. Section 3 discusses the fundamental design of our implementation and its relationship to the ArcAngel semantics. The following section illustrates the use of the tool through an example, and in Section 5 we draw our conclusions, address some aspects of extensions and generalisations, and identify future work.

2 Preliminaries

In this section, we introduce the relevant preliminary material. First, the ArcAngel tactic language is explained in more detail: we illustrate its use, and briefly discuss its semantic model. The last section provides background information on ProofPower.

2.1 ArcAngel

ArcAngel includes basic tactics, like **skip** or the application of laws, tacticals, and structural combinators, which facilitate the application of tactics to arguments of the program operators. The basic tactics and tacticals are inherited from Angel, albeit adapted to deal with refinement laws. Whereas Angel provides one structural combinator for applying tactics to sequences of subgoals, ArcAngel provides a collection of structural combinators corresponding to the program constructors.

A tactic program in ArcAngel is a sequence of tactic declarations. We declare a tactic name with body t and arguments *args* as **Tactic** name(*args*) t **end**. The

optional clause **proof obligations** documents the proof obligations (provisos) produced by application of t. An additional optional clause **generates** records the shape of the generated program. The body of the declaration can be any tactic expression involving the variables introduced through *args*.

The most basic tactic is **law** name(*args*); it assumes name, the law, to be *a priori* defined and to have parameters that are suitably instantiated by *args*. If name with arguments *args* is applicable, the application of the tactic succeeds and returns a new program, possibly generating proof obligations for the provisos of name. If, on the other hand, the law is not applicable, the tactic fails. An analogous construct exists to invoke a declared tactic. Its syntax is **tactic** name(*args*) where name is the name of the tactic, and *args* the list of arguments passed to it.

The other basic tactics are **skip**, **fail**, and **abort**. The tactic **skip** always succeeds leaving the program unchanged, **fail** always fails, and **abort** neither succeeds nor fails, but may produce any (list of) outcome(s) or even run indefinitely. Nontermination is not equated with failure since we cannot compute it. With regards to implementability of angelic nondeterminism, failure must always be inferable from tactic execution. Thence comes the need to distinguish **fail** and **abort**.

Tactics can be composed using tacticals. We already met the binary tacticals $t_1 \,;\, t_2$ for sequential composition, and $t_1 \mid t_2$ for alternation. Alternation is strict with respect to **abort** in its first operand, but not the second one, because, whenever t_1 succeeds or leads to success, application of t_2 is not carried out.

The cut operator $!\, t$ is a unary tactical. Its effect is to apply t, but only considers the first result of the application when there is more than one possible outcome due to nondeterminism. It acts like a 'cut' in Prolog with regards to the backtracking search of finding a feasible path of tactic execution.

Two further unary tactics are the assertions **succs** t and **fails** t. The first terminates without changing the program (that is, behaving like **skip**) if t succeeds, and otherwise fails. The second terminates without changing the program if t fails, and otherwise fails. Both are strict with respect to **abort** too.

ArcAngel also permits the specification of recursive tactics; they are generally useful to define tactics that carry out repetitive actions. The fixed-point construction $\mu\, X \bullet f(X)$ is used for this purpose; here f is some function on tactics.

The tactic **applies to** p **do** t guards the application of t by checking whether the program the tactic is applied to is of the form p, which acts as a pattern. If the pattern matching succeeds, the free variables in p are instantiated as meta-variables, and can be referenced in the definition of t. Otherwise, the tactic fails. To illustrate this, consider, for example, **applies to** $w : [pre, post_1 \land post_2]$ **do** t being applied to the program $x, y : [true, x = 1 \land y = 2]$. The matching in this case associates w with $\langle x, y \rangle$, pre with $true$, $post_1$ with $x = 1$, and $post_2$ with $y = 2$. The body of t can refer to w, pre, $post_1$ and $post_2$ in its definition.

Finally, structural combinators allow us to apply tactics to subprograms of some program operator. For example, $t_1 \boxed{;} t_2$ transforms programs of the form $p_1 \,;\, p_2$ by applying t_1 to p_1 and t_2 to p_2. The proof obligations generated are those arising from both tactic applications, and the piecewise application of the tactics is sound because of monotonicity, namely here of sequential composition in both operands. In ArcAngel, we have a structural combinator for each syntactic construct of the

Tactic takeConjAsInv($invBound, lstVar, lstVal, variantExp$)
applies to $w : [pre, inv \wedge \neg\ guard]$ **do**
law strPost($inv \wedge invBound \wedge \neg\ guard$) ;
law seqComp($inv \wedge invBound$) ;
(**law** assign($lstVar, lstVal$) $\boxed{;}$ **law** iter($\langle guard \rangle, variantExp$))
proof obligations
1. $inv \wedge \neg\ guard \wedge invBound \Rightarrow inv \wedge \neg\ guard$ (from strPost)
2. $pre \Rightarrow (inv \wedge invBound)[lstVar \backslash lstVal]$ (from assign)
generates
$lstVar := lstVal$;
do $guard \rightarrow w : \left[\begin{array}{c} inv \wedge invBound \wedge guard, \\ inv \wedge 0 \leq variantExp < variantExp[w \backslash w_0] \end{array}\right]$ **od**
end

Fig. 2. Definition of the takeConjAsInv tactic.

refinement calculus. They are easily identifiable as boxed versions of the program operators. In the sequel, we give an example of a non-trivial ArcAngel tactic.

2.1.1 Tactic Example

Fig. 2 presents a tactic to derive an initialised iteration. It uses the iter law in Fig. 3 to refine a specification of the form $w : [inv, inv \wedge \neg\ GG]$ into an iteration **do** $[]\ i \bullet G_i \rightarrow w : [inv \wedge G_i, inv \wedge 0 \leq V < V_0]$ **od**. Notice that 0-subscripted variables in the postcondition refer to initial values. Here, inv is the invariant of the loop, and GG the disjunction of the guards G_i. To apply this law, we have to provide a list of guards $\langle G_1, G_2, \ldots, G_n \rangle$ as well as a variant expression V.

The tactic *takeConjAsInv* performs a more sophisticated refinement of such specifications by additionally strengthening the invariant with a user-supplied (boundary) constraint, and performing an initialisation of the variables modified by the iteration. It was originally presented in [12]. In its definition and hereafter, we assume that $\wedge$ associates to the *left*; this reduces the number of parentheses.

The tactic *takeConjAsInv* is parametrised in terms of the additional invariant constraint $invBound$, the left-hand side $lstVar$ and right-hand side $lstVal$ of the initialising assignment, and the variant expression $variantExp$. The **applies to** _ **do** _ construct requires the program to be of the form $w : [pre, inv \wedge \neg\ guard]$ for the tactic to be applicable. Its body executes the primitive laws strPost, seqComp, assign and iter; their definitions are also given, in Fig. 3.

First, the application of the strPost law strengthens the postcondition of the specification statement to $inv \wedge invBound \wedge \neg\ guard$, including the additional conjunct $invBound$. The corresponding proviso (1) is always true. The second law seqComp decomposes the specification statement using $inv \wedge invBound$ as an intermediate condition. The program resulting from this step is of the general form

$$w : [pre, inv \wedge invBound] ; w : [inv \wedge invBound, inv \wedge invBound \wedge \neg\ guard]$$

The law seqComp is applicable providing that neither the intermediate condition nor

Law Name	Definition	Provisos
strPost($post'$)	$w : [pre, post] \sqsubseteq w : [pre, post']$	$post' \Rightarrow post$
seqComp(mid)	$w : [pre, post] \sqsubseteq w : [pre, mid] \mathbin{;} w : [mid, post]$	mid and $post$ have no free initial variables
assign(w, E)	$w : [pre, post] \sqsubseteq w := E$	$pre \Rightarrow post[w \backslash E]$
iter($\langle G_1, G_2, \ldots, G_n \rangle, V$)	$w : [inv, inv \wedge \neg\, GG] \sqsubseteq$ **do** $[\!]\ i \bullet G_i \rightarrow w : \left[\begin{array}{c} inv \wedge G_i, \\ inv \wedge 0 \leq V < V_0 \end{array}\right]$ **od**	neither inv nor any of the G_i contain initial variables

Fig. 3. Laws used in the definition of the takeConjAsInv tactic.

the postcondition $inv \wedge invBound \wedge \neg\, guard$ contains initial variables. The final step uses the structural combinator $\boxed{;}$ to apply the assign law to the first operand of the sequential composition, and the iter law to the second operand. The assign law refines a specification statement into an assignment, and gives rise to proof obligation (2). We thus obtain the program reported in the **generates** clause.

In Section 4 we encode *takeConjAsInv* in ProofPower.

2.1.2 Semantics of ArcAngel

An important feature of ArcAngel is that, like Angel, it is equipped with a formal semantics. Tactics in ArcAngel are characterised by functions that map refinement cells to (possibly infinite) lists of refinement cells. A refinement cell captures a program expression and includes a set of proof obligations to derive that program.

$$\underline{RCell} == \underline{Program} \times \mathbb{P}\, \underline{Predicate} \quad \text{and} \quad \underline{Tactic} == \underline{RCell} \nrightarrow \text{pfiseq}\ \underline{RCell}$$

$\underline{Program}$ is the semantic domain for program expressions, and $\underline{Predicate}$ represents proof obligations. The list generated by a tactic application can be infinite, namely if there is an infinite succession of possible outcomes, and also has to admit the possibility of being only *partially* defined. For example, the tactic **skip** | **abort** generates a list for which evaluation of only the first element is guaranteed to succeed. Any further outcome is undefined and could even lead to evaluation failing to terminate. The standard representation of lists as (finite) sequences is not expressive enough. In [7] Martin presents a model for partial, finite and infinite lists (pfi lists). The function pfiseq above is the type constructor for such lists.

In this model lists can be either partial or finite. Whereas finite lists end in concatenation with the empty list as in $1 : 2 : [\,]$, partial lists end in concatenation with the undefined list $\bot$. The interpretation of, for example, $1 : 2 : \bot$, is that evaluation of only the first two elements is guaranteed to succeed; attempting to evaluate the remainder of the list could give any result, and even fail to terminate.

To illustrate how infinite lists represent tactic outcomes, we consider the application of **skip** | **abort** to a refinement cell r. Applying **skip** only yields one result, the finite list $r : [\,]$. When moving to infinite lists, we however have to consider all approximations, here that is $\{\bot, r : \bot, r : [\,]\}$. The result of applying **abort**, on the other hand, is $\bot$, and the chain of approximations is the singleton set $\{\bot\}$.

The semantic function $[\![\ldots]\!]$ for tactics is further parametrised in terms of law and tactic environments which record the declared laws and tactics. Its type is $\underline{TacExpr} \rightarrow \underline{LEnv} \rightarrow \underline{TEnv} \rightarrow \underline{Tactic}$, where $\underline{TacExpr}$ is the set of tactic expressions.

The semantics of basic tactics is then given as follows.

$$[\![\mathbf{skip}]\!]\ \Gamma_L\ \Gamma_T\ r\ =\ [r] \qquad [\![\mathbf{fail}]\!]\ \Gamma_L\ \Gamma_T\ r\ =\ [\,] \qquad [\![\mathbf{abort}]\!]\ \Gamma_L\ \Gamma_T\ r\ =\ \bot$$

Here, Γ_L and Γ_T denote the law and tactic environments under which application is considered, and r the refinement cell to which the tactic is applied.

The semantics of **law** $\mathsf{name}(args)$ is a singleton list with the refinement cell containing the transformed program and possibly additional proof obligations, or otherwise an empty list if application fails. The law definition is inferred from the law environment. Similarly, **tactic** $\mathsf{name}(args)$ executes the tactic name by inferring its definition from the tactic environment.

For sequential composition, we have the following definition.

$$[\![t_1\,;\,t_2]\!]\ \Gamma_L\ \Gamma_T\ r\ =\ {}^{\infty}\!/\ ([\![t_2]\!]\ \Gamma_L\ \Gamma_T)*\ ([\![t_1]\!]\ \Gamma_L\ \Gamma_T\ r)$$

Here, ${}^{\infty}\!/$ is the distributed concatenation of pfi lists. The operator $*$ is a mapping function: $(f*)\ s$ applies f to all elements of a pfi list s. Informally, we apply t_2 to all cells obtained by applying t_1, and flatten the resulting list of lists.

We omit a discussion of the semantics of the remaining tacticals and structural combinators [12]. Recursive tactics are defined using Kleene's fixed-point theorem.

$$\mu\, X \bullet f(X)\ =\ \bigsqcup \{i : \mathbb{N} \bullet f^i(\mathbf{abort})\}$$

This requires a complete partial ordering on tactics with respect to which the tactic operators must be continuous. It is defined by $t_1 \sqsubseteq_T t_2 \equiv \forall\, r : RCell \bullet t_1\ r \sqsubseteq_\infty t_2\ r$ where $\sqsubseteq_\infty$ is the generalised prefix ordering on infinite lists. Intuitively, if t_1 is refined by t_2 then t_1 can at least produce as many outcomes as t_2 if applied to some arbitrary program. Moreover, whenever t_1 guarantees to terminate, t_2 must also be guaranteed to terminate under the same conditions. The notion of equivalence and refinement of tactics provides opportunities for specifying and proving algebraic laws about the tactic language. In the context of our work, they additionally allow us to test the correctness of the implementation to be developed.

2.2 ProofPower and Standard ML

ProofPower is a mechanical theorem prover that resulted from a re-engineering of the Cambridge HOL proof system. The latter is itself a descendant of LCF, and hence ProofPower shares various commonalities with the LCF prover; for example, it uses Standard ML (SML) as its implementation language and takes advantage of ML's type system to ensure that theorems can be constructed only by means of logical inference, and hence must be valid. This level of assurance is achieved by introducing an abstract data type `THM` for proved theorems whose exposed constructor functions invariably correspond to valid inferences in the logic.

A design objective of ProofPower was to facilitate the semantic embedding of other languages. In particular, the Z language has been embedded and formalised, producing the ProofPower-Z package and dialect. It is in essence an extension of ProofPower that provides additional syntactic constructs, parsing facilities, rules,

theorems and tactics specific to transforming and proving theorems about Z expressions. The open architecture and flexibility of ProofPower encouraged the development of several tools that promoted its use on industrial-scale projects [4,2].

Our implementation of ArcAngel integrates with ProofPower by supplying a database of additional SML constants and function definitions. Although much of it needs to use lower-level functions of ProofPower for dissecting syntactic expressions, manipulating type information, and so on, none of this poses a risk in terms of soundness, neither can potential bugs lead to unsound inferences.

Standard ML is a strongly-typed, strict and impure functional language. It is a modern descendant of the ML programming language that was used in implementing the Edinburgh LCF proof system. Being impure it permits the use of global mutable data structures by means of reference types. A comprehensive account of the ML language and its facilities can be found in [15] and `http://www.smlnj.org`.

3 Fundamental Design

In this section we discuss some of the core features of the design integrating ArcAngel into ProofPower. We first explain how we encode tactics, secondly address some implementation issues of operator encodings, and lastly show how ArcAngel tactics are used together with the standard backward proof facilities of ProofPower.

3.1 Encoding of ArcAngel Tactics

As already hinted, encoding ArcAngel tactics directly by virtue of ProofPower tactics is problematic. First, ProofPower tactics do not exhibit backtracking behaviour, and secondly ProofPower tactics solve (or reduce) *proof goals* whereas ArcAngel tactics transform program expressions. To bridge this gap, we first introduce the notion of a *refinement theorem*. It is a theorem of the form $\Gamma \vdash A \sqsubseteq B$, where Γ is a list of assumptions (provisos), and both A and B are program expressions. The operator $\sqsubseteq$ is assumed to represent refinement in Morgan's calculus. We later highlight that it may indeed be *any* reflexive and transitive relation.

Because refinement theorems are of central importance, we introduce a type abbreviation `REF_THM` for them. In fact, `REF_THM` is equated with `THM`, but it allows us to indicate when functions expect or return refinement theorems, and we implicitly assume that such theorems will always be of the correct shape.

ArcAngel tactics in ProofPower apply to refinement theorems rather than program expressions. Their application results in transformation of the *second* program of a refinement theorem. For example, if we have a tactic t which refines $x := x + y$ into $x := y + x$, applying it to the theorem $\vdash x : [x = x_0 + y] \sqsubseteq x := x + y$ yields the refinement theorem $\vdash x : [x = x_0 + y] \sqsubseteq x := y + x$.

In general, the successful application of a single law to a refinement theorem $\Gamma_1 \vdash A \sqsubseteq B$ delivers a theorem $\Gamma_1, \Gamma_2 \vdash A \sqsubseteq B'$ where B' is a valid refinement of B under the additional provisos Γ_2, which contribute to those of the resulting theorem. It is obtained by first matching the left-hand program of the refinement law against B. This gives an instantiation $\Gamma_2 \vdash B \sqsubseteq B'$ of the law which, by transitivity of refinement, permits the prover to conclude $\Gamma_1, \Gamma_2 \vdash A \sqsubseteq B'$.

The above design unifies ArcAngel's approach to program transformation with the design of ProofPower which is centred on theorem-generating functions. The application of an ArcAngel tactic to a program X can be simulated by first creating an initial refinement theorem $\vdash X \sqsubseteq X$ that is trivially proved by reflexivity of refinement. To it, we apply the encoding of the ArcAngel tactic. If successful, it returns a theorem $\Gamma \vdash X \sqsubseteq Y$ encapsulating the transformation of X to Y. The validity of the refinement is established by the soundness of primitive inferences of ProofPower's core logic; for that reason it is independent of our actual implementation of ArcAngel which merely drives the prover. This protection we do not get in an implementation of ArcAngel based on rewrite systems such as Gabriel [14], since in those the validity of rules and laws are not independently verified.

Nondeterminism and Infinite Behaviours

To accommodate nondeterminism, which surfaces when the application of a tactic can produce more than one possible result, we have to keep track of all possible outcomes of tactic behaviours. For example, $t_1 \mid t_2$ can have two possible outcomes if both tactics are applicable to the program. In order to determine which execution path leads to success and realise backtracking, if necessary, we have to keep track of both outcomes. We therefore characterise tactics as functions mapping refinement theorems to *lists* of refinement theorems. This characterisation closely resembles the semantic model of ArcAngel presented in Section 2.1 that modelled tactics by functions mapping refinement cells to (infinite) lists of refinement cells.

This design is obviously suitable for tactics with finite behaviours, but extra care is required to cater for tactics that potentially generate an infinite number of outcomes, or otherwise fail to produce any result due to nontermination. To illustrate this case, consider the following recursive tactical.

$$\mathbf{EXHAUST}(t) \mathrel{\widehat=} \mu\, X \bullet (t\,;\, X) \mid \mathbf{skip}$$

It entails the possibility of applying t once, twice, or in fact an arbitrary number of times. If t repetitively shows to be applicable, the tactic has an infinite number of potential outcomes. Operationally, this results in an infinite list $p \frown t(p) \frown t^2(p) \frown t^3(p) \ldots$ to be computed when $\mathbf{EXHAUST}(t)$ is applied to some program p. From a computational point of view this evaluation cannot succeed, however $\mathbf{EXHAUST}(t)$ is distinct from **abort** and its behaviour is perfectly well-defined as long as we are not attempting to utilise (evaluate) all outcomes.

A similar situation arises with $!\,(t \mid \mu\, X \bullet X)$. The tactic $\mu\, X \bullet X$ is equivalent to **abort**. On the other hand, if t does not fail, the behaviour of $!\,(t \mid \mu\, X \bullet X)$ is determined by the first result delivered by t. The application of $(t \mid \mu\, X \bullet X)$ then relies on the result of applying only t eluding the abortive recursion.

We, therefore, adopt lazy evaluation when computing the outcomes of tactic applications. Specifically, we introduced a datatype `lazylists` that allows us to defer evaluation of tactics until we actually require their results. Since evaluation in SML is strict, lazy evaluation must be simulated by means of additional layers of functions with a spurious argument. For example, evaluation of `t` is deferred in `(fn () => t p)` until we apply the function to an empty tuple. (The construction `(fn` *args* `=>` *body*`)` is generally used in Standard ML for anonymous functions.)

In particular, `lazylists`, defined below, gives us explicit control over which elements have their evaluation deferred. As an extreme case, it also permits deferred evaluation of the entire list. This is important to represent $\bot$.

```
datatype 'a lazylist = LazyNil |
   LazyAtom of 'a |
   LazyJoin of ('a lazylist) * ('a lazylist) |
   LazyDefer of (unit -> 'a lazylist);
```

This datatype provides four constructor functions. `LazyNil` is used to construct empty lazy lists, `LazyAtom` to construct atomic (non-lazy) elements, `LazyJoin` to concatenate two lazy lists, and `LazyDefer` to explicitly defer evaluation. This list model, being a lazy variant of the *join* list model, was tailored to provide the flexibility and expressiveness to implement ArcAngel operators in a correct, concise, and efficient way, in particular, the tactic combinators for alternation and recursion.

To support parametrised tactics and the **applies to** p **do** t operator, it is necessary to incorporate a special notion of environment that binds (meta)variables to expressions. They are represented by a list of pairs of ProofPower terms, where the first component gives the variable, and the second component the bound expression. We introduce the type abbreviation `ENV` to represent the set of such lists.

To conclude, ArcAngel tactics are encoded by functions that map environments and refinement theorems to lazy lists of refinement theorems.

```
type AA_TACTIC = ENV -> (REF_THM -> REF_THM lazylist);
```

Environments are in most cases just propagated to the operands in tactic combinators; the exception is **applies to** p **do** t and law and tactic applications which need to process them. In the next section we will look at some issues related to the implementation of the ArcAngel operators.

3.2 Operator Implementation

Each operator of ArcAngel is implemented by a designated SML function. They are listed in Fig. 4. The structural combinators are omitted in this table as they can be added dynamically by virtue of a set of constructor functions.

The implementation of the literal tactics is very simple. `TSkip` returns a singleton lazy list containing the program the tactic is applied to, `TFail` returns an empty lazy list, and `TAbort` raises an exception `Abort` that indicates abortion.

For law applications via `TLaw` the implementation essentially carries out the steps discussed in the previous section. It has to do a bit more work, however, to substitute meta-variables occurring free in the arguments, and move implications in the conclusion of the law theorem into the assumptions to make them provisos.

For laws to be applicable, they first have to be declared using the `TLawDecl` function. It expects the name of the law, its formal arguments as a list of typed terms, and the corresponding ProofPower theorem. Similarly, we declare a tactic using the `TTacDecl` function and apply it using `TTactic`. The function `TTacDecl` corresponds to the **Tactic** name(*args*) *tbody* **end** construct of ArcAngel.

The implementation of tacticals mirrors in most cases the respective semantic definitions. For example, SML implementation of `TSeq` is as follows.

Operator	Syntax	Signature of corresponding SML function
Basic Law	**law** N (*args*)	`fun TLaw (name : string) (args : TERM list);`
Tactic	**tactic** N (*args*)	`fun TTactic (name : string) (args : TERM list);`
Skip	**skip**	`val TSkip;`
Fail	**fail**	`val TFail;`
Abort	**abort**	`val TAbort;`
Sequence	$t_1 \,;\, t_2$	`fun (t1 : AA_TACTIC) TSeq (t2 : AA_TACTIC);`
Alternation	$t_1 \mid t_2$	`fun (t1 : AA_TACTIC) TAlt (t2 : AA_TACTIC);`
Cut	$!\, t$	`fun TCut (t : AA_TACTIC);`
Recursion	$\mu\, X \bullet t(X)$	`fun TRec (tfun : AA_TACTIC -> AA_TACTIC);`
Assertion	**succs** t	`fun TSuccs (t : AA_TACTIC);`
Assertion	**fails** t	`fun TFails (t : AA_TACTIC);`

Fig. 4. SML functions that encode ArcAngel operators.

```
fun (t1 : AA_TACTIC) TSeq (t2 : AA_TACTIC) : AA_TACTIC =
   (fn env : ENV => (fn p : REF_THM =>
      (lazyflat (lazymap (t2 env) (t1 env p)))));
```

It is a direct literal translation of the semantics where $\overset{\infty}{/}$ is encoded by `lazyflat`, and $*$ is encoded by `lazymap`. These two SML functions perform operations on lazy lists similar to the semantic functions on infinite lists. Both functions are implemented in a way that defers evaluation until an element is requested.

A further interesting function is `TRec`, which implements the ArcAngel recursion construct, and whose implementation is given below.

```
fun TRec (tfun : AA_TACTIC -> AA_TACTIC) : AA_TACTIC =
   (let val rec (trec : AA_TACTIC) =
      (fn env => (fn p =>
         (defer_tac_eval (tfun trec) env p))) in
      trec
   end);
```

The `tfun` argument provides the body of the recursion: a function on ArcAngel tactics. The local constant `trec` is introduced as a recursively-defined value which is used to determine the result of `TRec`. In defining `trec` it is vital that the recursive unfolding takes place incrementally and application of the tactic to the goal is deferred in each step. This is achieved by the function `defer_tac_eval` which defers the application of one unfolding `(tfun trec)` to the program `p`.

```
fun defer_tac_eval (t : AA_TACTIC) (env : ENV) (p : REF_THM) =
   LazyDefer (fn () => t env p);
```

This function takes advantage of the lazy list constructor `LazyDefer` to create a deferred list, suppressing the application of `t` to the environment and program.

In the next section we will clarify the integration of ArcAngel tactics with ProofPower's subgoal package providing the facilities for backward proofs.

3.3 Backward Proofs

The embedding of ArcAngel was developed outside the subgoal package of ProofPower. We can, however, support the use of ArcAngel tactics to facilitate the proof of refinement conjectures in a backward manner. This makes ArcAngel available to support development of programs correct by construction and verification of proposed refinements. For example, a proof goal of the form $A \sqsubseteq B$ can be discharged by an ArcAngel tactic that is able to transform A into B while possibly generating some provisos which contribute to the subgoals of the proof. Alternatively, if the tactic cannot discharge the goal in one step, it may still be able to reduce it to some intermediate refinement which may be discharged with less effort.

In order to invoke ArcAngel tactics within backward proofs, we provide a function (`aa_tac` *atac*) that lifts an ArcAngel tactic *atac* into a corresponding ProofPower tactic. The behaviour of the ProofPower tactic for a proof goal of the form $A \sqsubseteq B$ is as follows. First, the ArcAngel tactic is applied to the first program of the refinement conjecture. If the application of the ArcAngel tactics fails, this also results in failure of the wrapping ProofPower tactic. Otherwise we take the first element of the list of generated refinement theorems; it will always be of the form $\Gamma \vdash A \sqsubseteq A'$.

By adding the provisos Γ as subgoals to the current proof tree, we can justify the addition of $A \sqsubseteq A'$ to the goal hypotheses. ProofPower's default `asm_tac` achieves this for assumptionless theorems, and we have a more general version that also handles assumptions. The additional hypothesis either immediately discharges the goal if $A' = B$, or can be used to reduce the goal to $A' \sqsubseteq B$. This is justified by transitivity of refinement since $A \sqsubseteq A'$ and $A' \sqsubseteq B$ imply the initial goal $A \sqsubseteq B$. The low-level steps of this reduction are automatically carried out by `aa_tac`.

We also provide an alternative implementation (`aa_solve_tac` *atac*) that evaluates *all* outcomes of tactic applications to A, and selects one that discharges the goal or otherwise fails if none exists. Such a behaviour is faithful to the angelic interpretation of nondeterminism at the top level since the notion of success is clearly defined here as discharging the proof goal. This is also compatible with the mechanics of Angel which explores all possible paths of tactic executions.

In the next section we illustrate our implementation using *takeConjAsInv*.

4 Tactic Example

The tactic takeConjAsInv was presented in Fig. 2 in Section 2.1.1. It invokes four laws, strPost, seqComp, assign and iter. Each law is first formulated as a ProofPower-Z theorem, and afterwards declared and registered using the `TLawDecl` function. For example, the law strPost (see Fig. 3) is formalised by the theorem

$$
\begin{array}{l}
\vdash \forall\, u : MORGAN\_UNIVERSE;\ f : \mathrm{seq}\, M\_VAR\_NAME; \\
\quad preC : MORGAN\_CONDITION;\ postC, postC' : MORGAN\_POSTCOND \mid \\
\qquad (u, f, preC, postC) \in WF\_SpecStmt_M \land \\
\qquad (u, f, preC, postC') \in WF\_SpecStmt_M \land Tautology\,(postC' \Rightarrow_P postC) \bullet \\
\qquad\quad SpecStmt_M\,(u, f, preC, postC) \sqsubseteq SpecStmt_M\,(u, f, preC, postC')
\end{array}
$$

The law is specified in the context of a semantic encoding of Morgan's calculus based

on a mechanisation of the Unifying Theories of Programming (UTP), tailored for ProofPower-Z [13,17]. That mechanisation, and within it the semantic characterisation of Morgan's calculus, is for space considerations not discussed in detail. The functions $SpecStmt_M$ and $\Rightarrow_P$ encode the specification statement and the implication operator, and *Tautology* determines whether a given predicate is universally true. We also have semantic sets used as types: *MORGAN_UNIVERSE* contains all valid type constraints on the variables (universes), *M_VAR_NAME* the set of permissible frame variables, and *MORGAN_CONDITION* and *MORGAN_POSTCOND* the semantic domain for the pre and postconditions of a specification statement; the latter two are restricted forms of predicates. Lastly, the set $WF\_SpecStmt_M$ encapsulates well-definedness constraints for applying the $SpecStmt_M$ function.

The quantified variables u, f, $preC$ and $postC$ of the theorem are matched against the program when the law is instantiated. As already explained, the program is obtained as the right-hand side of the refinement conjecture to which the law is applied, and the matching is needed to instantiate the law so that its left-hand side equals the program to be refined. The variable $postC'$, on the contrary, is a parameter of the law. It is not matched but substituted by the actual argument when the law is invoked through `TLaw` with a specific list of arguments.

The antecedents of the law establish well-formedness constraints as well as the provisos of the law. The former here ensure that $SpecStmt_M\,(u, f, preC, postC)$ and $SpecStmt_M\,(u, f, preC, postC')$ are well-defined expressions in the semantic model. The only genuine proviso is $Tautology\,(postC' \Rightarrow_P postC)$. The theorem thus is a faithful encoding of the abstractly specified law.

The law is configured to be used by the implementation with the following command which identifies its formal parameters.

```
TLawDecl "strPost" [⌜postC' ⨁ ALPHA_PREDICATE⌝] strPost_thm;
```

We assume that the SML constant `strPost_thm` has been initialised to hold the law theorem. The first argument `"strPost"` specifies the name of the declared law in ArcAngel, and the second argument supplies the list of quantified variables used as parameters. The variables are given as (ProofPower) variable terms whose type must be explicitly specified by virtue of the '$\oplus$' operator. The presence of type information in formal arguments is exploited to inject possibly missing type information into the actual arguments when the law is applied; this makes the application of laws altogether more robust since a lack of type information in arguments might cause technical problems, like argument substitution to fail because of a discrepancy in types between substituting and substituted terms. It also allows for run-time type checks on parameters when the law is invoked.

Our implementation supports law theorems that are either given in pure HOL or the Z sub-language of ProofPower-Z. It also permits laws to have free variables (which are just treated as quantified ones, so that the outer universal quantification is optional), and finally enables the law theorems to contain provisos and assumptions. A general form for a law theorem is thus

$$\Gamma \vdash \forall\, v_1 : T_n;\ v_2 : T_n;\ \ldots \mid p_1 \wedge p_2 \wedge \ldots \bullet q_1 \wedge q_2 \wedge \ldots \Rightarrow A \sqsubseteq B$$

where Γ as well as the p_i and q_i collectively contribute as assumptions of the law.

The responsibility of proving theorems for laws resides with the user, and the proofs are formally justified within our semantic model of the Morgan calculus.

We omit the encoding of the remaining laws seqComp, assign and iter. The encoding of the iter law (see Fig. 3) is challenging as it is parametrised by a sequence of guarded commands, and the implementation in its present form can only support individual instances of this law with a fixed number of guarded commands.

We now declare the compound tactic whose body invokes the four laws.

```
TTacDecl "takeConjAsInv" [
    ⌜z invBound ⊕⊕ ALPHA_PREDICATE⌝ , ⌜z lstVar ⊕⊕ seq M_VAR_NAME⌝ ,
    ⌜z lstVal ⊕⊕ seq EXPRESSION⌝ , ⌜z variantExp ⊕⊕ EXPRESSION⌝ ] (
        TAppliesTo ⌜z SpecStmt_M (u, w, preC, invConj ∧P (¬P guard))⌝ TDo (
            (TLaw "strPost" [⌜z (invBound ∧P invConj) ∧P (¬P guard)⌝]) TSeq
            (TLaw "seqComp" [⌜z invBound ∧P invConj⌝]) TSeq
            ((TLaw "assign" [⌜z lstVar⌝ , ⌜z lstVal⌝])
                TSCSeq (TLaw "iter_unary" [⌜z variantExp⌝]))));
```

As with law declarations, tactic declarations have to provide a name for the tactic and a list of terms for the formal arguments, each being a variable with fully-qualified type information. The third argument specifies the body of the tactic. The translation that encodes the tactic is very direct, simply replacing ArcAngel operators and structural combinators by their corresponding ML functions. Fig. 4 may be used as a reference here, and a similar list of functions exists for the structural combinators of ArcAngel. For instance, `TSCSeq` encodes the structural combinator for sequential composition as an infix operator on tactics.

Variables introduced in a tactic declaration via `TTacDecl` or the `TAppliesTo` construct become local and can be used in its body. For example, the local variable *invBound* is introduced by the tactic declaration, and used in specifying the arguments for the law applications of strPost and seqComp. To illustrate the application of the tactic, we first create a program that encodes the specification statement

$$q, r : [a \geq 0 \land b > 0, a = q * b + r \land \neg\, r \geq b]$$

It calculates the quotient and remainder of two positive numbers a and b; they are respectively recorded in the variables q and r. Its encoding in the semantic model is slightly tedious, but inherently not difficult. We associate it to a constant `prog`.

We apply the tactic takeConjAsInv now to the above program. For this purpose, we use the function `aa_rule`. It expects an ArcAngel tactic and a program expression, and automatically creates an initial refinement theorem $\vdash P \sqsubseteq P$ to which it applies the tactic. Here, all this takes place outside the subgoal package of ProofPower, although, as we explained in Section 3.3, ArcAngel tactics may also be directly used within the standard backward proof engine.

```
aa_rule (TTactic "takeConjAsInv"
    [⌜z TrueP u⌝ , ⌜z ⟨q, r⟩⌝ , ⌜z ⟨Val(Int(0)), Var(a)⟩⌝ , ⌜z Var(r)⌝])
    prog
```

Analogous to `TLaw`, `TTactic` supports the application of a declared tactic. The parameters given are $\mathit{True}_P\ u$ for *invBound*, $\langle q, r\rangle$ for *lstVar*, $\langle \mathit{Val}(\mathit{Int}(0)), \mathit{Var}(a)\rangle$ for *lstVal*, and $\mathit{Var}(r)$ for *variantExp*. The *invBound* parameter is used to provide an additional predicate to strengthen the invariant, for example to encapsulate some

boundary conditions on indexed variables. It is not relevant here, hence set to be *true*. The *lstVar* and *lstVal* parameters determine how the variables altered by the loop are to be initialised. Here we want to carry out the initialisation $q, r := a, 0$ before entering the loop. Further, *variantExp* provides a variant. The result of the tactic application is the following refinement theorem, which we obtained as a result of applying the tactic in ProofPower-Z and type-setting the output.

$$
\begin{array}{l}
\ldots \vdash \\
\mathit{SpecStmt}_M\,(u, \langle q, r\rangle, \mathit{Rel}_P\,(u, (\_ \geq_V \_), \mathit{Var}\ a, \mathit{Val}(\mathit{Int}\ 0)) \wedge_P \mathit{Rel}_P\,(u, (\_ >_V \_), \mathit{Var}\ b, \mathit{Val}(\mathit{Int}\ 0)), \\
\quad (=_P\ (u, \mathit{Var}(\mathit{dash}\ a), \mathit{Fun}_2((\_ +_V \_), \mathit{Fun}_2((\_ *_V \_), \mathit{Var}\ q, \mathit{Var}\ b), \mathit{Var}\ r)) \wedge_P \\
\qquad \mathit{Rel}_P\,(u, (\_ \leq_V \_), \mathit{Val}(\mathit{Int}\ 0), \mathit{Var}\ r)) \wedge_P \neg_P\ (\mathit{Rel}_P\,(u, (\_ \geq_V \_), \mathit{Var}\ r, \mathit{Var}\ b))) \\
\qquad \sqsubseteq \\
\mathit{Assign}_M\,(u, \langle q, r\rangle, \langle \mathit{Val}(\mathit{Int}\ 0), \mathit{Var}\ a\rangle)\ ;_M \\
\quad \mathit{do}_M\ (\langle \mathit{Rel}_P\,(u, (\_ \geq_V \_), \mathit{Var}\ r, \mathit{Var}\ b)\rangle, \\
\qquad \langle \mathit{SpecStmt}_M\,(u, \langle q, r\rangle, \\
\qquad (\mathit{True}_P\ u \wedge_P =_P (u, \mathit{Var}(\mathit{dash}\ a), \mathit{Fun}_2((\_ +_V \_), \mathit{Fun}_2((\_ *_V \_), \mathit{Var}\ q, \mathit{Var}\ b), \mathit{Var}\ r)) \wedge_P \\
\qquad\quad \mathit{Rel}_P\,(u, (\_ \leq_V \_), \mathit{Val}(\mathit{Int}\ 0), \mathit{Var}\ r)) \wedge_P \mathit{Rel}_P\,(u, (\_ \geq_V \_), \mathit{Var}\ r, \mathit{Var}\ b), \\
\qquad (\mathit{True}_P\ u \wedge_P =_P (u, \mathit{Var}(\mathit{dash}\ a), \mathit{Fun}_2((\_ +_V \_), \mathit{Fun}_2((\_ *_V \_), \mathit{Var}\ q, \mathit{Var}\ b), \mathit{Var}\ r)) \wedge_P \\
\qquad\quad \mathit{Rel}_P\,(u, (\_ \leq_V \_), \mathit{Val}(\mathit{Int}\ 0), \mathit{Var}\ r)) \wedge_P \mathit{Rel}_P\,(u, (\_ \leq_V \_), \mathit{Val}(\mathit{Int}\ 0), \mathit{Var}\ r) \wedge_P \\
\qquad\quad \mathit{Rel}_P\,(u, (\_ <_V \_), \mathit{Var}\ r, \mathit{Subst}_E\,(\mathit{Var}\ r, \mathit{zero}))\rangle)\ \mathit{od}_M
\end{array}
$$

The above theorem encodes the program refinement

$$
\begin{array}{l}
q, r : [a \geq 0 \wedge b > 0, a = q * b + r \wedge \neg\, r \geq b] \\
\sqsubseteq \\
\quad q, r := 0, a\,; \\
\quad \mathbf{do}\ r \geq b \rightarrow q, r : \left[\begin{array}{c} a = q * b + r \wedge 0 \leq r \wedge r \geq b, \\ a = q * b + r \wedge 0 \leq r \wedge r \geq b \wedge r \leq r_0 \end{array}\right] \mathbf{od}
\end{array}
$$

For readability, the assumptions of the theorem have been omitted. Most of them carry constraints regarding the well-definedness of operator applications which were accumulated through application of the laws and monotonicity theorems. In practice, we anticipate that most of these assumptions are provable automatically without any user intervention. The remaining assumptions encapsulate the provisos of the laws. For example, we find the following assumption encoding the first proof obligation of the law, that is, the provisos of the strPost law.

$$
\begin{array}{l}
\textbf{\textit{Tautology}}(((\mathit{True}_P\ u \wedge_P =_P (u, \mathit{Var}(\mathit{dash}\ a), \mathit{Fun}_2((\_ +_V \_), \mathit{Fun}_2((\_ *_V \_), \mathit{Var}\ q, \mathit{Var}\ b), \mathit{Var}\ r)) \\
\qquad \wedge_P \mathit{Rel}_P\,(u, (\_ \leq_V \_), \mathit{Val}(\mathit{Int}\ 0), \mathit{Var}\ r)) \wedge \neg\, (\mathit{Rel}_P(u, (\_ \leq_V \_), \mathit{Var}\ r, \mathit{Var}\ b))) \\
\quad \Rightarrow_P (=_P (u, \mathit{Var}(\mathit{dash}\ a), \mathit{Fun}_2((\_ +_V \_), \mathit{Fun}_2((\_ *_V \_), \mathit{Var}\ q, \mathit{Var}\ b), \mathit{Var}\ r)) \wedge_P \\
\qquad \mathit{Rel}_P\,(u, (\_ \leq_V \_), \mathit{Val}(\mathit{Int}\ 0), \mathit{Var}\ r)) \wedge_P \neg\, (\mathit{Rel}_P\,(u, (\_ \geq_V \_), \mathit{Var}\ r, \mathit{Var}\ b)))
\end{array}
$$

The proof of provisos like the above is expected in most cases to require human interaction and knowledge, although automation for restricted domains of application has been successful even in industry [1]. Here we have to show that $\mathit{true} \wedge a' = q * b + r \wedge 0 \leq r \wedge \neg\, r \leq b \Rightarrow a' = q * b + r \wedge 0 \leq r \wedge \neg\, r \geq b$. This can be trivially proved by using $\mathit{true} \wedge P \Rightarrow P \equiv \mathit{true}$.

When applying ArcAngel tactics outside the subgoal package of ProofPower as above, tactic applications usually increase the number of assumptions of the generated refinement theorem. In backward proofs we use the refinement theorem to

simplify a refinement goal, and in this case the assumptions contribute as additional subgoals of the proof and need to be discharged separately. Fortunately, ProofPower offers a lot of flexibility in implementing programming utilities and tactics to automate proofs, and for this purpose we have already developed a collection of specialised (ProofPower) tactics that can automatically discharge most of the well-definedness constraints encountered so far so that the only real proof effort that has to be invested is in discharging the provisos of the laws.

We applied the takeConjAsInv tactic to several other examples to calculate the product, exponentiation and factorial. The corresponding source code for the examples, including the semantic embedding of Morgan's refinement calculus, can be found at http://www.cs.york.ac.uk/circus/tp/tools.html. The tactic was originally used as an example in [14] and [11], and we did in fact encode the other tactics given in those publications too.

5 Conclusions

In this paper we have presented an implementation of the ArcAngel tactic language for the ProofPower theorem prover. We discussed several aspects of the implementation as well as decisions made in its design, and illustrated it by virtue of an example. Notably, we managed to realise a very direct translation of the ArcAngel semantics in which refinement cells in the semantic model are identified with refinement theorems in the implementation model. A faithful representation of partial and infinite lists was achieved through the use of lazy evaluation, and recursive ArcAngel tactics are directly supported through recursive ML functions on tactics. Hence, the encoding of particular tactics is mostly trivial and amenable to automation.

To verify that the design correctly reflects the semantics of ArcAngel, we conducted a series of tests in which we verified (in specific pathological cases) that tactics which can be proved equal in the semantics also exhibit similar behaviours in the implementation. This provided some empirical evidence for the correctness of the implementation and indeed revealed deficiencies in earlier designs, which have been gradually refined to culminate into the one presented in this paper.

The implementation has suggested several extensions and generalisation that due to space limitations could not be discussed in the paper. A first extension we realised is to simultaneously deal with the proof of equivalences and genuine refinements in the application of ArcAngel tactics. Conceptually, ArcAngel is oblivious to the interpretation of the underlying refinement relation — it is at core a method for transforming programs. In ProofPower, the application of ArcAngel tactics results in the generation of refinement theorems, making the refinement relation explicit. This offers the possibility to deal with various kinds of relations at once, namely $A \sqsubseteq B$ and $A \equiv B$. This feature was implemented as to always generate the strongest theorem that can be asserted based on the shape of the applied laws.

A second extension ventured a step further by isolating the program model from the core implementation of ArcAngel and thereby making it dynamically configurable. We hereby unified the application of ArcAngel tactics to various kinds of objects (not just Morgan computations) by identifying what the minimal requirements are for the equivalence and refinement relations necessary for the mechanics

of the implementation to work. In particular, the essence of an ArcAngel model can be captured in a small number of theorems about the equivalence and refinement relations whilst their actual definitions and types are irrelevant. Factoring out the program model retains ArcAngel's flexibility as a general method for transforming terms, and provides future opportunities for the implementation to be applied in different scenarios. It also constitutes a unification with ArcAngel*C* [11] and in parts with Angel as both can be obtained as derivatives by means of suitable models.

A third extension that we provided addresses the problem of nontermination in tactics. Tactics that get trapped in a nonterminating loop are not uncommon in theorem provers, for example, if repetitive rewrites of expressions continually succeed. In our implementation it is possible to concisely deal with nontermination because the only point where it may occur is in recursive tactics. We do this by imposing a limit on the number of recursive calls, and give a semantic interpretation of bounded recursion in terms of an approximation to the exact meaning of the recursive tactic as a fixed point respective to the refinement ordering on tactics. Although the bounded recursive tactic 'loses' some behaviour of the respective unbounded one, in practice the limit on the recursive calls may be set to a sufficiently high value to justify the use of the former in place of the latter.

Related work apart from REFINE and Gabriel is the implementation of Angel in Ergo [16,9], a theorem prover developed in Prolog. A major difference between Ergo and ProofPower is that Ergo does not have a core object logic, whereas ProofPower is based upon a formalisation of HOL logic. The implementation of Angel in Ergo is more general in that it allows tactics to be applied not just to single goals, but sequences of goals that in turn may result from the application of tactics. This in particular would be an interesting extension to our implementation as it might allow us to define tactics that specify how provisos should be handled in the generation of refinement theorems. For example, they could in some cases be further subject to proof through application of (Arc)Angel tactics.

Future work first consists of enhancing and extending the implementation. An issue that has been pointed out in the previous section relates to specifying the iter law in its general form. To do so, we require certain pre and post-processing steps in the application of the law, namely to rewrite instantiated theorems to carry out possible syntactic operations in processing the law application. ProofPower's conversion mechanism may be conveniently used for that.

A second crucial area of improvement is the handling of provisos generated by tactic applications. These can amount to a considerable number of assumptions which all need to be proved when the generated refinement theorem is used, for example, as part of some standard backward proof. To reduce these assumptions, a number of approaches may be considered. First, we may try to prove them in-line using specialised tactics. Additionally, rules may be specified that compress the assumptions of refinement theorems by attempting to prove individual assumptions from the residual ones, and using the cut rule to eliminate them. Features in the interface are required to first identify which provisos should be proved in-line, and secondly what ProofPower or even ArcAngel tactic should be used to discharge them.

A final area for future work is the development of case studies for realistic applications. Here, we anticipate to formulate the refinement strategy for control

laws presented in [3] as a collection of ArcAngel tactics and thereby automate the application of the refinement strategy to arbitrary and sizable examples.

Acknowledgements

We are grateful to Matthew Naylor for suggestions regarding the use of lazy lists. We also acknowledge EPSRC for funding this work under the 'Programming from Control Laws' research grant EP/E025366/1. INES and CNPq partially supports the work of Marcel Oliveira: grants 550946/2007-1, 620132/2008-6, and 573964/2008-4.

References

[1] Adams, M. and P. Clayton, *ClawZ: Cost-Effective Formal Verification of Control Systems*, in: *Formal Methods and Software Engineering*, Lecture Notes in Computer Science **3785** (2005), pp. 465–479.

[2] Arthan, R., P. Caseley, C. O'Halloran and A. Smith, *ClawZ: Control laws in Z*, in: *3rd International Conference on Formal Engineering Methods* (2000), pp. 169–176.

[3] Cavalcanti, A., *From Control Laws to Ada via Circus*, Technical report, University of York, York, YO10 5DD, UK (2008), available from http://www.cs.york.ac.uk/ftpdir/reports/2008/YCS/428/YCS-2008-428.pdf.

[4] Clayton, P. and C. O'Halloran, *Using the Compliance Notation in Industry*, in: *Refinement Techniques in Software Engineering*, Lecture Notes in Computer Science **3167** (2006), pp. 269–314.

[5] Kirchner, F. and C. Muñoz, *PVS#: Streamlined Tacticals for PVS*, Electronic Notes in Theoretical Computer Science **174** (2007), pp. 47–58.

[6] Martin, A., "Machine-Assisted Theorem Proving for Software Engineering," Ph.D. thesis, Oxford, UK (1994), technical Monograph PRG-121.

[7] Martin, A., *Infinite Lists for Specifying Functional Programs in Z*, in: *Proceedings of Fifth Australasian Refinement Workshop* (1996).

[8] Martin, A., P. Gardiner and J. Woodcock, *A Tactical Calculus – Abridged Version*, Formal Aspects of Computing **8** (1996), pp. 479–489.

[9] Martin, A., R. Nickson and M. Utting, *A Tactic Language for Ergo*, in: *Formal Methods – Pacific 97*, Discrete Mathematics and Theoretical Computer Science, 1997, pp. 186–207.

[10] Morgan, C., "Programming from Specifications," Prentice Hall International Series in Computer Science, Prentice Hall, 1998.

[11] Oliveira, M. and A. Cavalcanti, *ArcAngelC: a Refinement Tactic Language for Circus*, Electronic Notes in Theoretical Computer Science **214** (2008), pp. 203–229.

[12] Oliveira, M., A. Cavalcanti and J. Woodcock, *ArcAngel: a Tactic Language for Refinement*, Formal Aspects of Computing **15** (2003), pp. 28–47.

[13] Oliveira, M., A. Cavalcanti and J. Woodcock, *Unifying Theories in ProofPower-Z*, in: *Unifying Theories of Programming, First International Symposium*, Lecture Notes in Computer Science **4010** (2006), pp. 123–140.

[14] Oliveira, M., M. Xavier and A. Cavalcanti, *Refine and Gabriel: Support for Refinement and Tactics*, in: *Proceedings of the Second International Conference on Software Engineering and Formal Methods* (2004), pp. 310–319.

[15] Paulson, L. C., "ML for the Working Programmer, 2nd Edition," Cambridge University Press, 1996.

[16] Utting, M. and K. Whitwell, *Ergo User Manual – An Interactive Theorem Prover* (1994).

[17] Zeyda, F. and A. Cavalcanti, *Mechanical Reasoning about Families of UTP Theories*, in: *SBMF 2008, Brazilian Symposium on Formal Methods*, 2008, pp. 145–160.