Under consideration for publication in Formal Aspects of Computing

Mechanised Support for Sound Refinement Tactics

Frank Zeyda[1], Marcel Oliveira[2] and Ana Cavalcanti[1]

[1]Department of Computer Science, University of York, Heslington, York, UK.

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

Abstract. ArcAngel is a 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 refinement strategies, and equipped with a formal semantics that additionally permits 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 the interplay between ArcAngel tactics and the native reasoning support of the prover. We also discuss several extensions of ArcAngel that have been entailed by our implementation effort. They are of practical importance and provide a unification of the related tactic languages Angel and ArcAngel*C*. Our main result is a mechanisation that reflects directly the ArcAngel semantics, and can be used with any programming model for refinement. The approach can be used to support other formal tactic languages using other theorem provers.

Keywords: tactic language; refinement; automation; Z; unifying theories

1. Introduction

Morgan's refinement calculus [Mor98] supports the derivation of programs from specifications by a series of correctness-preserving refinement steps. Each step is justified by the application of a refinement law that guarantees that the program obtained correctly implements its specification.

The ArcAngel language [OCW03] supports the documentation and automation of recurring sequences of refinement steps. It has a formal semantics, and an extensive set of algebraic laws that support reasoning about tactics. ArcAngel is an extension of Angel [Mar94, MGW96], a general-purpose tactic language; they owe their name to their support for the use of angelic choices in the process of solving proof goals.

A number of tools have been proposed that address the issue of interactive and automatic refinement [CHN+94]. Groves et al. and Vickers [GNU92, Vic90] present tools that can support refinement via tactics in Morgan's calculus. They, however, do not give a formal semantics to the underlying refinement language, and thereby it is not possible to independently verify the laws. A similar restriction applies to

Correspondence and offprint requests to: Frank Zeyda, Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK the Gabriel extension of the interactive refinement tool REFINE [OXC04]. At the other end of the spectrum, von Wright presents a proof-based approach that mechanises refinements in HOL [Wri94]. Derivation is sound in this work, however, the concern is not to provide a high-level language for refinement. Use of the tool requires direct interaction with the window-inference toolkit that drives the tool underneath. The PRT tool [CHN+98] aims to address both extensibility and support for sound derivation, but focuses on user interaction rather than automation. The Ergo theorem prover [UW94], which is implemented in Prolog, was extended to use Angel, but not ArcAngel, as a tactic language [MNU97a].

In this paper, we present an implementation of ArcAngel; our main contribution is an approach to provide sound automated support for tactic-based refinement. Our mechanisation (and its underlying design) is distinctive. (1) It is based on a tactic language with a formal semantics, so that we enable reasoning about the tactics themselves. (2) It supports the sound derivation of refinements with the protection of the LCF approach. (3) It enables independent verification of refinement laws with respect to a program model. (4) It is extensible and can be used in conjunction with arbitrary refinement languages and underlying semantic models. (5) It handles proof obligations and their discharge via the same mechanisms that apply refinement tactics. (6) It supports the seamless integration of native reasoning facilities of the prover.

None of the existing tools do justice to all these features at once. On the one hand, we have refinement editors tailored to a particular language. They do not provide proof support to validate refinements, and are often interactively driven with limited support for automation. On the other hand, there are works that tie in with automatic provers, but do not provide a level of abstraction that facilitates the development of readable and powerful high-level strategies nor permit integration of custom languages. Our work bridges this gap by supporting a user-friendly, high-level tactic language that is suitable for the automation of complex strategies, and allows the user to combine native proof support and tactics in a flexible manner.

Main-stream provers do not generally perform automatic backtracking upon tactic failure. Our tool makes this feature available by faithfully implementing ArcAngel's angelic choice. In addition, conventionally tactics apply to goals (sequents). The purpose of a refinement strategy, on the other hand, is to transform program expressions. Our tool provides support for coupling transformation-based reasoning and goal-based proof. An ArcAngel tactic is defined in terms of refinement laws, and, whenever possible, produces a theorem that establishes that a given original specification is refined by a program (derived by the application of the tactic). Proof obligations are either discharged or kept as hypotheses; they can be predicates over the program variables (or logical constants) or refinement conjectures.

We also identify contributions to ArcAngel itself: we propose four extensions. A first extension is the notion of a program model, which formalises the assumptions underlying the sound use of ArcAngel. This allows the core of our tool to be used with computational models other than Morgan's calculus. It also makes the tool useful in proving equivalences of programs in addition to genuine refinements.

Another extension to ArcAngel is a new tactical that supports the combination of tactics to refine programs with tactics to discharge (or simplify) the raised proof obligations. This allows us to take full advantage of the capabilities of a theorem prover in mechanising and automating sound refinements.

As a further extension, we have also addressed the treatment of expressions used as arguments of tactics and law applications. The original semantics of ArcAngel makes a simplifying assumption that all expressions are fully evaluated. From a practical point of view, this is not always adequate; for instance, the shape of expressions may have an effect on the use of laws that are applicable to a program. We provide an annotation mechanism that supports fine control of expression evaluation.

Finally, we consider the practical treatment of termination in ArcAngel. We define and implement extra tacticals for definition of recursive tactics. They limit the number of recursive calls to avoid nontermination. They also allow us to decide whether, by exceeding the allowed number of recursive calls, the tactic aborts or fails. This gives more control to the construction of useful tactics.

A third contribution of this paper is a mechanisation of Morgan's refinement calculus in ProofPower-Z, which is based on an encoding of the Unifying Theories of Programming [HJ98, ZC08]. It completes the implementation of ArcAngel by defining its program model, but can also be used independently. In addition, it provides further validation for the work in [ZC08], which supports the definition of programming theories.

Our work leads in parts to a unification of ArcAngel with its kin Angel, as well as the more specialised derivate ArcAngel*C*, a variant tailored for action and process refinement in the *Circus* language [OC08]. A by-product of this unification is a framework that fosters the development of other derivatives of Angel.

To demonstrate our approach, we have used ProofPower, a flexible and extensible theorem prover based on HOL. It has an open architecture, and has been successfully employed on industrial projects (for example, in the verification of avionics control systems [AC05]). ProofPower also provides an embedding of Z [WD96] known as ProofPower-Z. This is useful in defining semantic models for refinement languages, and we also take advantage of the expressiveness of Z in examples discussed later on.

A preliminary discussion of our mechanisation is provided in [ZOC09]. Here, however, we present the design that actually fulfills our goals of soundness and generality. The approach in [ZC10b] does not incorporate support for integration of a variety of programming models for refinement, discharge of proof obligations via ArcAngel tactics, and fine-tuned handling of expression evaluation.

In Section 2 we introduce the relevant preliminary material. The following Section 3 introduces the extensions and generalisations that we propose. Section 4 discusses the fundamental design of our implementation and its relationship to the ArcAngel semantics. Section 5 is dedicated to the formalisation and mechanisation of four programming models, and shows how they are used in conjunction with our tool. Section 6 illustrates the use of the tool, and in Section 7 we finally draw our conclusions.

2. Preliminaries

In this section, we introduce the relevant preliminary material: Morgan's calculus, ArcAngel and its semantics, and ProofPower and its implementation language, Standard ML, which is also used in our implementation.

2.1. Refinement calculus

Morgan's calculus provides a language in which sequential programs and their specifications can be written. In that context, and here, the term program refers in general to abstract specifications, executable programs, and designs that include specifications combined using program constructors. The language includes familiar constructs from imperative programming, such as assignments, conditionals, local variables and loops. We can also write a specification statement $w : [pre, post]$ to capture the abstract behaviour of a program that can update the variables in the list w, called the frame, and has precondition pre and postcondition $post$. In the postcondition, we can use 0 subscripts to refer to the initial values of variables. For example, $x : [x \geq 0, x = x_0 + 1 \lor x = x_0 - 1]$ specifies a computation that either increments or decrements any positive x; for a negative x, it behaves abortively and thus may exhibit any behaviour, including nontermination.

Table 1 presents an overview of the core constructs of Morgan's calculus. After the specification statement, the next five are those of Dijkstra's guarded command language [Dij76]. Conditional and iteration are defined in terms of a list of guarded commands. The conditional nondeterministically chooses one program p_i whose guard g_i is true, and if no such guard exists, aborts. Iteration repetitively executes one of the programs whose guard is true and terminates if none exists. If there is more than one program with a true guard the choice is nondeterministic. Logical constants are a convenient way to refer to values of interest during program development. For procedures, we follow Back's approach based on parameterised commands [Bac87]. Procedures may have arguments which can be passed using any of the call-by-copy mechanisms.

The calculus is equipped with a collection of refinement laws that allow step-by-step transformations of specifications into executable programs. A comprehensive list of laws is presented in [Mor88].

2.2. ArcAngel

ArcAngel [OCW03] includes basic tactics, like **skip** or the application of a refinement law, tacticals, which are general operators on tactics, and structural combinators, which facilitate the application of tactics to components of a program, that is, arguments of the program operators. The basic tactics and tacticals are inherited from Angel [MGW93, MGW96], albeit adapted to deal with refinement laws to programs.

A tactic program in ArcAngel consists of a sequence of tactic declarations. We declare a tactic name with body t and arguments $args$ as **Tactic** name $(args) \mathrel{\widehat{=}} t$ **end**. The body t of the declaration can be any tactic expression involving the variables introduced through $args$. For documentary purposes only, an optional clause **proof obligations** can be included to enumerate the proof obligations produced by the application of t. An additional optional clause **generates** records the shape of the derived program.

The most basic tactic is **law** name$(args)$, which performs the application of a single law; 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

Name	Syntax
Specification Statement	$w : [pre, post]$
Skip	**skip**
Assignment	$w := E$ where w and E may be lists
Sequential Composition	$p_1 \; ; \; p_2$
Conditional	**if** $g_1 \rightarrow p_1 \; [\!] \; g_2 \rightarrow p_2 \; [\!] \; \ldots \; [\!] \; g_n \rightarrow p_n$ **fi**
Iteration	**do** $g_1 \rightarrow p_1 \; [\!] \; g_2 \rightarrow p_2 \; [\!] \; \ldots \; [\!] \; g_n \rightarrow p_n$ **od**
Local Variable	**var** $x : T \bullet p$
Logical Constant	**con** $x : T \bullet p$
Procedure Declaration	**procedure** $name \mathrel{\widehat=} p$ **procedure** $name \mathrel{\widehat=} (\ldots,$ **val** $x : T, \ldots \bullet p)$ **procedure** $name \mathrel{\widehat=} (\ldots,$ **res** $x : T, \ldots \bullet p)$ **procedure** $name \mathrel{\widehat=} (\ldots,$ **vres** $x : T, \ldots \bullet p)$
Procedure Call	$name$ and more generally $name(arg_1, arg_2, \ldots)$

Table 1. Constructs of Morgan's refinement calculus.

generating proof obligations corresponding to the provisos of the law 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 while 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 effectively compute it. With regards to implementability of angelic choice, failure must always be inferable from tactic execution. Hence comes the need to distinguish **fail** and **abort** at the semantic level.

Tactics can be composed using tacticals. A sequential composition $t_1 ; t_2$ applies the tactics t_1 and t_2 in sequence. If t_1 fails (or aborts), the whole tactic $t_1 ; t_2$ fails (or aborts), otherwise the outcome is determined by subsequently applying t_2. An alternation $t_1 \mid t_2$ first attempts to apply t_1, and if this leads to failure at any point, applies t_2. An important feature of the alternating choice is that it is angelic: it always finds a successful execution, if there is one, by making the right choices. Alternation is strict with respect to **abort** in its first operand, but not the second one, because, whenever t_1 succeeds, application of t_2 is not carried out. The entire tactic aborts if either t_1 aborts, or t_1 fails and t_2 aborts.

The cut operator $!\,t$ is unary. It applies t, but considers only the first result when there is more than one possible outcome due to alternation. It acts like the cut in Prolog with regards to the backtracking search for a feasible path of execution. The tactical $!\,t$ fails when t fails; similarly, it aborts if t aborts.

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. The fixed-point construction $\mu X \bullet f(X)$ is used for this purpose; here f is a function on tactics. Recursive tactics may introduce nontermination; for example, $\mu X \bullet$ **skip**$\,; X$ repeatedly applies **skip** without ever yielding a result. It is equivalent to **abort**.

The tactic **applies to** p **do** t guards the application of t by checking whether the program to which t 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, we consider the application of **applies to** $w : [pre, post_1 \wedge post_2]$ **do** t to the program $x, y : [true, x = 1 \wedge y = 2]$. The pattern 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. We observe that the pattern matching does not make use of associativity of operators like '$\wedge$' and ','. It is in essence syntactical and hence there is no possibility of multiple matches.

Name	Syntax	Description
Law Application	**law** name(*args*)	Application of a simple refinement law.
Tactic Application	**tactic** name(*args*)	Application of a declared tactic.
Skip	**skip**	Tactic that succeeds not altering the program.
Fail	**fail**	Tactic that always fails.
Abort	**abort**	Tactic that may not terminate.
Sequence	$t_1 \mathbin{;} t_2$	Sequential composition of tactics t_1 and t_2.
Alternation	$t_1 \mid t_2$	Alternating choice between tactics t_1 and t_2.
Cut	$!\, t$	Only considers the first result when applying t.
Recursion	$\mu X \bullet f(X)$	Least fixed-point operator on tactics.
Assertion 1	**succs** t	Behaves like **skip** if t succeeds, and fails if t fails.
Assertion 2	**fails** t	Behaves like **skip** if t fails, and fails if t succeeds.
Pattern Matching	**applies to** p **do** t	Guards the application of t by a pattern p.

Table 2. List of ArcAngel tactics and tacticals. Structural combinators are omitted.

Finally, structural combinators allow us to apply individual tactics to subprograms of some program operator. For example, the tactic $t_1 \boxed{;} t_2$ transforms programs of the form $p_1 \mathbin{;} p_2$ 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 justified by monotonicity, namely here of sequential composition in both operands. In ArcAngel, we have a structural combinator for each syntactic construct of the refinement calculus. For instance, to apply the tactics $t_1, t_2, \ldots$ to the programs $p_1, p_2, \ldots$ of a conditional **if** $g_1 \rightarrow p_1 \mathbin{[\!]} g_2 \rightarrow p_2 \mathbin{[\!]} \ldots$ **fi**, we employ the n-ary structural combinator $\boxed{\textbf{if}}\, t_1 \boxed{[\!]}\, t_2 \boxed{[\!]} \ldots \boxed{\textbf{fi}}$.

Table 2 summarises the ArcAngel constructs, apart from the structural combinators, which are identifiable as boxed versions of the program operators. Significant examples of ArcAngel tactics can be found in [OCW03, OZC11]. We use in the sequel a series of small examples that illustrate well our mechanisation approach.

Semantics of ArcAngel In what follows, we summarise some elements of the existing semantic model of ArcAngel [OCW03] that are relevant to the presentation of the extensions we discuss in Section 3, and that have also instructed the design of our implementation.

Tactics in ArcAngel are characterised by functions that map refinement cells to (possibly infinite) lists of refinement cells. A refinement cell captures a program and a sequence of proof obligations to derive it.

$$RCell \mathrel{\widehat=} Program \times \mathrm{seq}\ Predicate \quad\text{and}\quad Tactic \mathrel{\widehat=} RCell \rightarrow\!\!\!\!\!\!+\;\; pfiseq\ RCell$$

Program is the domain for program expressions, and *Predicate* represents proof obligations which are characterised by predicates. The list generated by a tactic application can potentially be infinite, namely if there is an infinite number of possible outcomes, and also can be 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 its evaluation failing to terminate. In [Mar96] Martin presents a model for **p**artial, **f**inite and **i**nfinite lists (pfi lists). The function *pfiseq* used above is the constructor for such lists. Appendix A includes its formal definition and a number of list operators.

The semantic function $[\![\ldots]\!]$ for tactics is parameterised by two environments that record declared laws and tactics: *LEnv* is the set of law environments, and *TEnv* that of tactic environments.

$$LEnv \mathrel{\widehat=} Name \rightarrow\!\!\!\!\!\!+\;\; \mathrm{seq}\, Expression \rightarrow\!\!\!\!\!\!+\;\; Program \rightarrow\!\!\!\!\!\!+\;\; RCell$$
$$TEnv \mathrel{\widehat=} Name \rightarrow\!\!\!\!\!\!+\;\; \mathrm{seq}\, Expression \rightarrow\!\!\!\!\!\!+\;\; Tactic$$

The type *Name* is that of law and tactic names, and *Expression* that of expressions used as arguments.

The semantic function for tactics maps tactic expressions and environments to elements of *Tactic*.

$$[\![\_]\!] : \mathit{TacticExpr} \to \mathit{TEnv} \to \mathit{LEnv} \to \mathit{Tactic}$$

As defined above, *Tactic* specifies the semantic domain for tactics; *TacticExpr*, on the other hand, contains all syntactic expressions for tactics. The semantics of basic tactics is defined as given below.

$$[\![\textbf{skip}]\!]\ \Psi_T\ \Psi_L\ r\ =\ [r] \qquad [\![\textbf{fail}]\!]\ \Psi_T\ \Psi_L\ r\ =\ [\,] \qquad [\![\textbf{abort}]\!]\ \Psi_T\ \Psi_L\ r\ =\ \perp$$

For **skip** the outcome is a finite singleton list containing only the refinement cell (program) r to which the tactic is applied. For **fail** it is the empty list, and for **abort** it is a partial list $\perp$ whose elements are left completely unspecified. The semantics of the tactic **law** name($args$) is a singleton list with the refinement cell containing the transformed program and possibly additional proof obligations, or an empty list if application fails. The law definition is inferred from the law environment. Similarly, **tactic** name($args$) executes the tactic name by inferring its definition from the tactic environment.

For sequential composition, we have the following definition.

$$[\![t_1\ ;\ t_2]\!]\ \Psi_T\ \Psi_L\ r\ =\ \overset{\infty}{\frown}/\ ([\![t_2]\!]\ \Psi_T\ \Psi_L)*\ ([\![t_1]\!]\ \Psi_T\ \Psi_L\ r)$$

Here, $\overset{\infty}{\frown}/$ 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 refinement cells obtained by first applying t_1, and flatten the resulting list of lists into a single list of results. Alternation has the following simple definition.

$$[\![t_1 \mid t_2]\!]\ \Psi_T\ \Psi_L\ r\ =\ ([\![t_1]\!]\ \Psi_T\ \Psi_L\ r)\ \overset{\infty}{\frown}\ ([\![t_2]\!]\ \Psi_T\ \Psi_L\ r)$$

The $\overset{\infty}{\frown}$ operator is concatenation for pfi lists. As an example, we consider the tactic **skip** | **abort**. Applying it to p yields $[p]\ \overset{\infty}{\frown}\ \perp$, a partial list whose first element is the only one that can be safely evaluated. On the other hand, if we consider **abort** | **skip**, we obtain $\perp\ \overset{\infty}{\frown}\ [p]$, which is equal to $\perp$.

We omit a discussion of the semantics of the remaining tacticals as well as structural combinators. Recursion is defined using Kleene's theorem, that is, as the least upper bound of approximation chains.

$$\mu\, X \bullet f(X)\ =\ \bigsqcup\{i : \mathbb{N} \bullet f^i(\textbf{abort})\}\ \text{ where } f^0 \text{ is the identity function.}$$

This requires a complete partial ordering on tactics with respect to which the tactic operators are continuous. It is defined by $t_1 \sqsubseteq_T t_2\ \equiv\ \forall\, r : \mathit{RCell} \bullet t_1\ r \sqsubseteq_\infty\ t_2\ r$ where $\sqsubseteq_\infty$ is a generalised prefix ordering on infinite lists. Intuitively, if t_1 is refined by t_2, then for every program, t_2 can produce at least as many outcomes as t_1, and whenever t_1 is guaranteed to terminate, so is t_2. Appendix A includes a formal definition.

The notion of semantic equivalence and refinement of tactics provided the opportunity to specify and prove algebraic laws about the tactic language [MGW96, OCW03]. In the context of our work, the laws allow us to test the correctness of the implementation with respect to the semantics of ArcAngel.

2.3. ProofPower and Standard ML

ProofPower [AJ05] resulted from a re-engineering of the Cambridge HOL proof system, a descendant of LCF; hence ProofPower has much in common with the original LCF prover. For example, it is implemented in (Standard) ML [MTH90], and takes advantage of its type system to ensure that theorems can be constructed only by means of logical inference. There is an abstract data type THM for proved theorems whose exposed constructor functions invariably correspond to valid inferences in the logic.

ProofPower promotes and facilitates the semantic embedding of languages; in particular, Z has been 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 terms. The open architecture and flexibility of ProofPower encouraged the development of several tools that enabled its use on industrial-scale projects [ACOS00, CO06].

Our implementation of ArcAngel directly integrates with the implementation of 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 can compromise soundness, and neither can potential bugs in our implementation. As explained in Section 4, tactics generate theorems, that is, elements of the data type THM that record a refinement proof.

Name	Model theorem
Reflexivity of Refinement	$\forall X : T \bullet X \sqsubseteq X$
Transitivity of Refinement	$\forall X, Y, Z : T \bullet X \sqsubseteq Y \land Y \sqsubseteq Z \Rightarrow X \sqsubseteq Z$
Equivalence	$\forall X, Y : T \bullet X \equiv Y \Leftrightarrow X \sqsubseteq Y \land Y \sqsubseteq X$

Table 3. Theorems that need to be provided for ArcAngel models.

Soundness of the theorem follows from the soundness of the ProofPower treatment of theorems. A bug or the use of inadequate tactics may fail to generate a (desired) theorem, but not an unsound result.

Standard ML is a strongly-typed, strict and impure functional language. The implementations of SML supported by ProofPower are Poly/ML (recommended) and New Jersey ML. Being impure, it permits the use of global mutable data structures by means of reference types. Being strict implies that, unlike in lazy languages, arguments of functions are always evaluated prior to the function itself.

A comprehensive account of the ML language and its facilities can be found in [Pau96]. For New Jersey Standard ML, we refer to the on-line resource and documentation at http://www.smlnj.org.

3. Extensions to ArcAngel

In this section, we discuss several extensions to ArcAngel that have been motivated by our implementation. We present them, however, in an implementation-independent way as each of them reflects a more general conceptual issue, which is relevant to every implementation of ArcAngel and similar languages on other theorem-proving platforms. All of them have been implemented in ProofPower.

3.1. Program model

In ArcAngel, we apply tactics to programs of the refinement calculus. The mechanics of ArcAngel, however, generally do not depend on the kind of objects to which tactics are applied. Tactic application is in essence a syntactic transformation process, and the meaning of those transformations is something that must be established outside the semantics of the tactic language. It depends on the type of objects we apply tactics to, and properties of the underlying refinement relation and program operators. For instance, if the basic refinement laws are not correct, the refinement relation does not have the right properties, or the program operators are not monotonic with respect to it, then we cannot claim that tactic application generally yields a correct refinement. This, however, is fundamentally of no relevance to the ArcAngel semantics.

This has advantages; it fosters a high level of generality in the tactic language. Namely, we can abstract from the particular details of the mathematical objects to which tactics are applied, and from the particular semantic definitions of program operators and refinement. On the other hand, we have no formal guarantee stemming from the use of ArcAngel per se that programs obtained via tactics are valid refinements.

In our tool, we provide support for the notion of a program model: it allows us to exploit the useful aspects of the generality of ArcAngel, while eradicating concerns about soundness. Our technique concretises and formalises the assumptions that we make outside the tactic language to justify its use.

3.1.1. Specification of the program model

A program model consists, first of all, of two relations over the program type T of the model. The first relation, $\sqsubseteq$, captures refinement, and the second relation, $\equiv$, program equivalence. The introduction of equivalence, in addition to refinement, enables us to use ArcAngel to prove equivalence between programs where this is possible. Section 3.1.4 explains this feature in more detail.

A second ingredient is the set of three specific theorems listed in Table 3; we call them model theorems. The first two establish that refinement is a preorder: reflexive and transitive. The third one requires that equivalence corresponds to mutual refinement. These three theorems, together, establish the suitability of the model for use with ArcAngel to establish program refinement and equivalence.

It should be noted that in order to verify the theorems, we clearly need a semantic theory for the underlying program objects as a point of reference. Furthermore, refinement and equivalence laws need to be justified by proof within that semantic theory. In return, we can claim that tactic application is sound. Soundness here means that if we apply a tactic to a program p of the model, all possible outcomes can be proved to be valid refinements of p, provided the indicated proof obligations are discharged.

We can prove this general result using the theorems in Table 3 and the definitions of the ArcAngel operators. This is, however, not necessary: our implementation provides evidence on a per application basis by constructing the refinement theorems established by the application of each tactic and tactical operator (by exploiting the model theorems). The LCF paradigm then ensures that all derivations are sound.

3.1.2. Structural combinators

A further aspect that needs attention in a program model is the properties of program operators for which we provide structural combinators. Their sound use relies on monotonicity theorems for the program operators to justify the application of tactics to program arguments to construct a refinement of the overall program.

For operators with a fixed number of arguments, the monotonicity theorems are straightforward. For those with a variable number of program arguments, like a conditional **if** $g_1 \rightarrow p_1 [\!] \ldots [\!] g_n \rightarrow p_n$ **fi** in the guarded command language, the monotonicity theorems are specified in terms of sequences of programs. If we name the operator for a conditional **if**, for instance, we have a theorem like that shown below.

$$\vdash \forall\, gs : T_1;\ ps, ps' : \mathrm{seq}\ T_2 \mid WF(gs, ps, ps') \land \#\, ps = \#\, ps' \bullet$$
$$(\forall\, i : \mathrm{dom}\, ps \bullet ps(i) \sqsubseteq ps'(i)) \ \Rightarrow\ \mathbf{if}(gs, ps) \sqsubseteq \mathbf{if}(gs, ps')$$

The operator **if** is defined to take a sequence of guards and, crucially, a corresponding sequence of programs as arguments. Accordingly, in the above theorem, the type T_1 is that of guards. The type T_2 of the elements of the sequences ps and ps' is that of programs. A condition $WF(gs, ps, ps')$, which we omit here, captures restrictions on the domain of the program operator, **if** in this case. The monotonicity theorem establishes that, if each element of the sequence ps refines the corresponding element of the sequence ps' (of the same size), then the operator application constructed from the programs in ps', here **if**(gs, ps'), is a refinement of the program obtained by applying the operator to the programs of ps.

Accordingly, an implementation of the structural combinator for the corresponding operator has to carry out a rewrite step that splits the universal quantification in the antecedent into individual refinements before they can be subject to tactic applications. This is an example of the interaction that is in many cases required between our tool and low-level proof facilities to support more generic tactics.

Additional provisos on the arguments, like $WF(gs, ps, ps')$ in the example above, capture the well-formedness of the operator application. They do not affect the mechanics of the proofs, and our work in [VZC10] explains in detail how they can be discharged automatically.

3.1.3. Predicative model

As already said, in our tool we provide support for the use of ArcAngel to refine programs encoded using any model that provides the relations and theorems described in the previous two sections. To support the use of ArcAngel to discharge proof obligations raised during refinement, including those that involve program refinements themselves, however, we introduce a particular program model, namely the (boolean) model of predicates. As explained in the sequel, it can be used alongside other program models of interest.

In the predicative model, we introduce refinement as reverse implication ($\Leftarrow$), and equivalence as bi-implication ($\Leftrightarrow$). In this special case, the model theorems are trivial laws of classical logic, and refining a predicate means to strengthen it. In particular, if we have a tactic that refines a predicate P into $true$, it effectively constructs a proof for P. (As detailed later on, the refinement theorem generated is $\vdash P \Leftarrow true$, which is equivalent to $\vdash P$.) Hence, tactics in the predicative model can be used for conventional proofs.

Refinement laws in the predicative model are simply laws about predicates. For example, the following law $\vdash P \lor Q \Leftarrow P$ can be used to reduce a boolean program $P \lor Q$ to P. (A symmetric law reduces it to Q). Although we cannot reduce conjunction in the same way, we may define a structural combinator $\boxed{\land}$ which exploits that conjunction is monotonic in both operands with respect to (reverse) implication. Using it as in $t_1 \boxed{\land}\ t_2$, we are able to reduce the proof of $P \land Q$ to a proof of P and Q individually via the two tactics t_1 and t_2. A trivial law $true \land true \Leftarrow true$ completes the refinement.

Refinement proof obligations can arise as a result of applying, for instance, laws for introduction of recursive procedure calls in Morgan's calculus. They take the form $\Gamma \vdash A \sqsubseteq B$ and can be discharged by reducing the first program A to B by a means of a suitable tactic. To support this approach, we introduce a unary structural combinator $\boxed{\sqsubseteq}$ that enables the application of a tactic to the first operand of the refinement $A \sqsubseteq B$ in the boolean program model. It is monotonic since $\Gamma \vdash A \sqsubseteq A'$ implies that $\Gamma \vdash (A \sqsubseteq B) \Leftarrow (A' \sqsubseteq B)$. This basically follows from transitivity of refinement. If, using this combinator, we obtain a refinement $\Gamma, \Gamma' \vdash B \sqsubseteq B$, a simple law can transform it to *true*.

We observe that $\boxed{\sqsubseteq}$ is a hybrid structural combinator: whilst its argument tactic applies to a program, the result is a refinement in the predicative model. Our use of program models does restrict the use of ArcAngel unnecessarily. We can handle, for example, a combinator for a preconditioned program operator $pre \mid p$. We can have a first tactic to weaken the precondition pre, and a second tactic to refine the program p. Soundness is justified in different semantic theories; this is handled in the monotonicity theorem.

The predicative program model does not require a custom semantic theory when encoded in a theorem prover, since its laws are easily provable by appealing to the deductive rules and core theorems of the prover. To support and use it we, however, require support for higher-order logic.

3.1.4. Proving equivalences

In this section, we explain how the refinement and equivalence relations are used in evaluating tactic applications that can either yield a refinement or an equivalence, depending on the laws used.

A tactic t can follow various paths of execution, depending on the particular program p to which it is applied. It is, therefore, not trivial to carry out an initial analysis of t to determine whether the program that it generates is necessarily equivalent to, or a refinement of, p. Depending on the laws used in the particular path taken by t when applied to p, we can have either.

In spite of that, instead of taking the safe route of regarding all generated programs as refinements, our tool produces a stronger equivalence theorem whenever possible. For that, it determines whether the generated program is an equivalence or refinement during the tactic application. This relies on the same properties we previously required for program refinement, that is, reflexivity and transitivity, to hold for equivalence. The third theorem in Table 3 is used to establish these properties.

3.2. Discharging proof obligations

Our tool implements a tactical **discharge** $t \ \langle t_1, t_2, \ldots, t_n \rangle$ that applies a tactic t and then uses the tactics t_1, $t_2, \ldots$ to simplify, or in the ideal case discharge and remove, the proof obligations generated by the application of t. It is not available in the original account of ArcAngel, and in this section we define its semantics.

The tactical **discharge** $t \ \langle t_1, t_2, \ldots, t_n \rangle$ collects the proof obligations generated by the application of t, say $\langle b_1, b_2, \ldots, b_n \rangle$, and applies each t_i to the i-th proof obligation b_i in that sequence. The proof obligations generated by the application of t are replaced by the transformed ones produced by the application of the t_i. Crucially, the application of each t_i to the respective b_i is realised in the predicative program model.

The semantics of **discharge** caters for the fact that applying t as well as each of the t_i may either yield a list of outcomes, or otherwise aborts or fails. The formalisation is provided below, where we consider the application of **discharge** to a tactic t and a list of tactics *pob_tacs*.

$$\begin{array}{l} [\![\textbf{discharge}\ t\ pob\_tacs]\!]\ \Psi_T\ \Psi_L\ r = \\ \quad \textbf{let}\ tacs = (\lambda\, tac \bullet [\![tac]\!]\ \Psi_T\ \Psi_L)\ \mathsf{map}\ pob\_tacs \bullet \overset{\infty}{\frown}/\ (discharge\_rcell\ tacs\ r) * ([\![t]\!]\ \Psi_T\ \Psi_L\ r) \end{array}$$

A local constant *tacs* records the semantics of each tactic in *pob_tacs* when applied to the environments Ψ_T and Ψ_L. This is determined by mapping $(\lambda\, tac \bullet [\![tac]\!]\ \Psi_T\ \Psi_L)$ over *pob_tacs*; the result is a list of type seq *Tactic*. It is used as an argument to *discharge_rcell*, along with the original refinement cell r. We use map for the map operator for the Z sequence constructor seq, and $*$ is the map for *pfiseq*.

Each cell of the (infinite) list $([\![t]\!]\ \Psi_T\ \Psi_L\ r)$ gives a possible outcome of the application of t to (the program in) r, along with the associated proof obligations. The function $(discharge\_rcell\ tacs\ r)$, when applied to such a refinement cell, gives the possible results of applying the tactics in *tacs* to each of the new proof obligations. This is in itself a possibly infinite list or refinement cells. The result of mapping $(discharge\_rcell\ tacs\ r)$ to all elements $([\![t]\!]\ \Psi_T\ \Psi_L\ r)$ is, therefore, a list of type *pfiseq* (*pfiseq RCell*). It is flattened into a list of type *pfiseq RCell* containing all the possible outcomes of the application of the tactics.

We define *discharge_rcell* below; as indicated above, it takes as parameter a list *tacs* of tactics to discharge the proof obligations, the original refinement cell *r_orig*, and a single refinement cell r' (which is one of those obtained from the application of t to *r_orig*).

$$
\begin{array}{l}
\mathit{discharge\_rcell}\ (\mathit{tacs} : \mathrm{seq}\ \mathit{Tactic})\ (\mathit{r\_orig} : \mathit{RCell})\ (r' : \mathit{RCell})\ = \\
\quad \mathbf{let}\ \mathit{pobs} = r'.2 - \mathit{r\_orig}.2 \bullet \\
\quad \mathbf{let}\ \mathit{pob\_rcells} = (\lambda\, \mathit{pob} \bullet (\mathit{pob}, \langle\rangle))\ \mathsf{map}\ \mathit{pobs} \bullet \\
\quad \mathbf{let}\ \mathit{pob\_rcells}' = \mathit{tacs}\ \mathsf{mapl}\ \mathit{pob\_rcells} \bullet \\
\quad \mathbf{let}\ \mathit{combs\_rcells}' = \mathsf{combine}_{\infty}\ \mathit{pob\_rcells}' \bullet \\
\quad \mathbf{let}\ \mathit{pobs}' = \mathit{pobs\_for\_comb} * \mathit{combs\_rcells}' \bullet \\
\qquad (\lambda\, \mathit{pob} \bullet (r'.1, \mathit{pob} \frown \mathit{r\_orig}.2)) * \mathit{pobs}'
\end{array}
$$

The constant *pobs* records the residual proof obligations in r': those generated by the application of t. The refinement cells in the sequence *pob_rcells* have as programs the proof obligations in *pobs*, and no proof obligations. By applying the tactics *tacs* to these refinement cells, we get a resulting sequence of sequences of refinement cells $\mathit{pob\_rcells}'$: each inner sequence is the outcome of simplifying a particular proof obligation with one of the tactics. The function mapl performs the pointwise mapping; it does not require the lists to have the same length: superfluous tactics are simply ignored, and surplus refinement cells are left untouched.

The function $\mathsf{combine}_{\infty}$ defined in Appendix A selects elements from a sequence of infinite sequences, and combines them in every possible way to obtain a possibly infinite sequence of all combinations (each combination is given as a standard sequence). Each element in $\mathit{combs\_rcells}'$ thus records a possible outcome of concurrently applying the tactics in *tacs* to the proof obligation refinement cells corresponding to *pobs*.

The function *pobs_for_comb* takes such a list of refinement cells and defines a corresponding list of plain predicates (that can be used to characterise the proof obligations in the resulting refinement cell). Its simple definition is omitted; *pobs_for_comb* just flattens all programs and proof obligation sequences of all refinement cells in *rs* into a single sequence of predicates.

The semantics of *discharge_rcell* is given by mapping $(\lambda\, \mathit{pob} \bullet (r'.1, \mathit{pob} \frown \mathit{r\_orig}.2))$ over the list pobs' of these proof-obligation sequences. This constructs, for each sequence *pob* in pobs', a cell whose program is that resulting from the application of t, and whose proof obligations are *pob* concatenated with the initial ones. In this way, the proof obligations generated by t are replaced with the result of applying *tacs* to them.

If any of the proof obligations are themselves refinements, the corresponding tactic supplied to **discharge** needs the refinement structural combinator $\boxed{\sqsubseteq}$ discussed in the previous section. In the next section we look at another practical aspect that has not been dealt with in existing work.

3.3. Evaluation of expressions

One issue not addressed by the semantics of ArcAngel is the treatment of expressions. The work in [OCW03] only considers (fully evaluated) values. In our tool, however, for practical reasons, we need to treat and control expression evaluation. It may be necessary to evaluate or avoid evaluation of particular expressions before and after the application of a law. The next two examples illustrate the issues.

First, we consider a tactic that takes a string s and applies the copy rule twice by suffixing s with each of the strings "`_action1`" and "`_action2`". We may specify this tactic as follows.

Tactic copy-rule-twice $(s) \mathrel{\widehat=}$ **law** copy-rule$(s \frown$ "`_action1`"$)$; **law** copy-rule$(s \frown$ "`_action2`"$)$ **end**

Whenever the tactic is applied, say as in **tactic** copy-rule-twice("`foo`"), the formal parameter s must be substituted by the argument provided. In this case this yields the following instantiation.

law copy-rule("`foo`" $\frown$ "`_action1`") ; **law** copy-rule("`foo`" $\frown$ "`_action2`")

The copy-rule expects a literal string as an argument (not a concatenation), hence here we need an implicit evaluation step. In [OZC11] analogous tactics are presented to chain the applications of the copy rule.

As a second example, we consider the iter law presented below. It refines specifications of the form

$w : [inv, inv \wedge \neg\, GG]$, where GG is an arbitrary disjunction of guards $G_1 \vee G_2 \vee \dots$.

Law iter($\langle G_1, G_2, \dots, G_n \rangle$, V)

$$w : [inv, inv \wedge \neg\, GG] \sqsubseteq \textbf{do}\ [\!]\ i \bullet G_i \rightarrow w : [inv \wedge G_i, inv \wedge 0 \leq V < V_0\,]\ \textbf{od}$$

provided neither inv nor any of the G_i contain initial variables.

We use $\textbf{do}\ [\!]\ i \bullet g(i) \rightarrow p(i)\ \textbf{od}$ to represent an iteration whose list of guarded commands has as its i-th element $g(i) \rightarrow p(i)$, with the guard $g(i)$ and the program $p(i)$ defined in terms of i.

The disjunction GG needs to be represented by means of a function, named GG for instance, which takes sequences of predicates as argument. Otherwise, we would require an instance of the law for each possible length of the disjunction of guards. Turning $(p \vee q)$ and $(p \vee q \vee r)$, for example, into $GG\,\langle p, q \rangle$ or $GG\,\langle p, q, r \rangle$, is then an implicit step that is necessary before the application of iter. Additionally, postprocessing is necessary to eliminate the uses of GG after applying the iter law.

We use annotations to control evaluation. In arguments of tactics and laws, annotations indicate the expressions that require evaluation before they are applied. Similarly, annotations in expressions within laws indicate that they should be preevaluated or postevaluated upon application. Precisely, when we apply a law stating $p_1 \sqsubseteq p_2$, annotations in p_1 lead to evaluation before application, and in p_2 lead to postevaluation.

Annotations are defined using a tagging function; formally, it is just an identity function. Utilising $Eval$ as the tagging function, our first example tactic is recast as follows.

Tactic copy-rule-twice (s) $\widehat{=}$

  law copy-rule($Eval(s \frown$ "`_action1`")) ; **law** copy-rule($Eval(s \frown$ "`_action2`"))

 end

Semantically, $Eval$ can always be eliminated. Syntactically, it causes evaluation according to a set of predefined rules. In general, $Eval$ operates in a top-down manner on expressions, carrying out successive rewrites.

During evaluation and instantiation of arguments, we cater for the possibility of type mismatch. Typing is not part of the ArcAngel semantics, since it depends on the underlying type systems of program models. We, therefore, carry out dynamic type checking for arguments of tactics (and laws). Types of formal parameters of law and tactics are recorded, and the types of actual arguments are checked against them. We define that the behaviour of tactics and laws applied to arguments of the wrong type is **abort**.

3.4. Termination in recursive tactics

For pragmatic reasons, we introduce a tactical for recursive tactics that imposes an upper limit on the number of unfoldings that are performed. The recursive tactical in this case effectively monitors the number of iterations performed, and behaves like **abort** if a certain threshold n is reached.

Pragmatically, this supports the implementation of more robust mechanisms for error-catching, as well as the possibility to utilise safely tactics that may fail to terminate. This is especially useful when it is not evident if, and under what conditions, tactics are guaranteed to terminate. By choosing a sufficiently high value for n we obtain a reasonable approximation of the behaviour of the recursive tactic.

Formally, we can describe the bounded recursive tactic $\mu_n\, X \bullet f(X)$ as shown below.

$$\mu_n\, X \bullet f(X) \;=\; \bigsqcup \{ i : \mathbb{N} \mid i \leq n \bullet f^i(\textbf{abort}) \} \;=\; f^n(\textbf{abort})$$

Unlike in the case of unbounded recursion, we only take the limit of the first n elements of the approximation chain $f^0(\textbf{abort}), f^1(\textbf{abort}), f^2(\textbf{abort}), \dots$. Because of monotonicity of f, this is equal to $f^n(\textbf{abort})$, and in terms of tactic refinement we have $\mu_n\, X \bullet f(X) \sqsubseteq \mu\, X \bullet f(X)$.

If we do not want to treat nontermination as an abnormal case, but use it to control the behaviour of other tactics, the alternative tactical $\mu^*_n\, X \bullet f(X)$ yields failure rather than abortion when exceeding the threshold. It fails if a certain number of unfoldings does not produce a result. The tactic $\mu^*_n\, X \bullet f(X)$ is equivalent to $f^n(\textbf{fail})$ where n is again the maximum number of recursive calls, however it is neither an approximation nor a refinement of $\mu\, X \bullet f(X)$. It is not an approximation since, for instance, we have $(\mu^*_n\, X \bullet X) = \textbf{fail}$ and $\mu\, X \bullet X = \textbf{abort}$, but $\textbf{abort} \not\sqsupseteq \textbf{fail}$. It is also not a refinement because $\mu^*_n\, X \bullet f(X)$ may produce fewer outcomes. It is nevertheless a useful description of recursion in practical terms, identifying detected nontermination with failure of a tactic to be applied.

Although it would be possible to engineer a model for ArcAngel that has only one semantic concept of failure, the fusion of **fail** and **abort** would cause implementability issues that cannot be overcome. The tactic $\mu_n^* X \bullet f(X)$ closes this gap by simulating this unified notion of failure by only looking a finite number of iterations into the future, whereas the implementation of angelic choice otherwise would have to achieve the impossible feat of solving the problem of termination in general for recursive tactic applications.

In the following section we explain the details of our implementation of ArcAngel in ProofPower, taking into account all we have discussed in terms of extensions and unification in this section.

4. Mechanisation 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 explain how ArcAngel tactics can be used together with the backward proof facilities of ProofPower.

4.1. Encoding of ArcAngel tactics

A naïve approach to encode 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. Although theorem provers typically provide functions to rewrite expressions, in LCF-based provers these functions have to act on theorems, and not expressions themselves, to ensure that inference is always sound.

To bridge this gap, we first introduce the notion of a refinement theorem in ProofPower. It is a theorem of the form $\Gamma \vdash A \sqsubseteq B$ or $\Gamma \vdash A \equiv B$, where Γ is a list of assumptions (provisos). Because refinement theorems are of central importance, for documentation purposes, we introduce a type abbreviation `REF_THM` for them. Although `REF_THM` is equated with `THM`, the standard type for theorems in ProofPower, it allows us though to indicate when functions expect or return refinement theorems.

ArcAngel tactics in ProofPower apply to refinement theorems, instead of refinement cells. Roughly, the program of a refinement cell is encoded as the refining program, and the proof obligations as the assumptions. Accordingly, the application of a tactic typically results in a transformation (rewrite) of the refining program, possibly with the addition of assumptions. We observe, however, that, as opposed to the list of proof obligations of a refinement cell, the assumptions of a refinement theorem cannot contain repetitions. This can have some impact on the programming of tactics, but since the order of assumptions is preserved, our experience shows that this is not an important restriction in practice.

Basically, the successful application of a law to a refinement theorem $\Gamma_1 \vdash A \sqsubseteq B$ produces a theorem $\Gamma_1, \Gamma_2 \vdash A \sqsubseteq B'$ where B' is a valid refinement of B under the additional provisos Γ_2. It is obtained by matching the refined program of the law against B to give an instantiation $\Gamma_2 \vdash B \sqsubseteq B'$, which, by transitivity, permits the prover to conclude $\Gamma_1, \Gamma_2 \vdash A \sqsubseteq B'$. If both the refinement theorem and the law are equivalences, the program model theorems in Table 3 allow us to prove the stronger result $\Gamma_1, \Gamma_2 \vdash A \equiv B'$.

This reconciles ArcAngel's approach of applying tactics to programs with the design of ProofPower which is centred around theorem-generating functions. The conventional application of an ArcAngel tactic to a program X can be simulated in ProofPower by first creating an initial refinement theorem $\vdash X \equiv X$ that is trivially proved by definition of equivalence and reflexivity of refinement. We apply to it the encoding of the ArcAngel tactic; if successful, a transformation of X to Y is captured by a theorem $\Gamma \vdash X \sqsubseteq Y$ or $\Gamma \vdash X \equiv Y$. The validity of the refinement or equivalence is established by the soundness of core inferences of ProofPower; it is independent of our actual implementation of ArcAngel which merely drives the prover.

Backtracking and infinite behaviours As formalised by the semantics of ArcAngel, to handle backtracking, we have to record all possible outcomes of a tactic. In our implementation, we therefore characterise tactics as functions mapping refinement theorems to lists of refinement theorems. This closely resembles the semantic model where tactics are functions mapping refinement cells to (infinite) lists of refinement cells.

To support infinite lists, we adopt lazy evaluation when computing the outcomes of tactic applications. Specifically, we introduce a data type `lazylist` that allows us to defer evaluation of tactics until we actually require their results. Since, unlike, for example, Haskell, evaluation in SML is generally strict, lazy evaluation must be simulated by means of additional layers of functions with a spurious argument. Our list model is a lazy variant of the join list model. It provides the flexibility and expressiveness required to implement ArcAngel operators in a correct, concise, and efficient way. In particular, it enables us to implement the tactic combinators for alternation and recursion concisely.

Environments The tactics and law environments are encoded as global variables. They are updated by the declaration of new laws and tactics (using functions discussed in the next section).

On the other hand, to implement parametrised tactics and the **applies to** p **do** t operator, it is necessary to incorporate an environment that binds (meta)variables to expressions. It is represented by a list of pairs of ProofPower terms (values of type `TERM`), 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. This is defined in SML as follows.

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

Environments are in most cases just propagated to the operands in tactic combinators, the exceptions are **applies to** p **do** t, and law and tactic applications which need to process them.

In the next section we look at the implementation of some of the ArcAngel operators.

4.2. Implementation of operators

Each operator of ArcAngel is implemented by a designated SML function. The implementations of the basic tactics directly mirror their semantic definitions. `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`.

For law applications, `TLaw` essentially carries out the steps explained in the previous section. In addition, it substitutes the meta-variables occurring free in the arguments. For laws to be applied, they have to be declared first 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 that formalises the law. Upon application, using `TLaw`, implications in the conclusion of a law theorem are moved to assumptions to make them provisos.

Similarly, we declare a tactic using the `TTacDecl` function and apply it using `TTactic`. The declaration of tactics with `TTacDecl` corresponds to the **Tactic** name$(args) \mathrel{\widehat=} t$ **end** construct of ArcAngel. Examples of the use of all these operator and declaration functions are provided in the next section.

The implementations of the tacticals in most cases also mirror their semantic definitions. As an example, we present the implementation of `TSeq`, a binary function on tactics that implements $t_1 ; t_2$.

```
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 literal translation of the semantics (see Section 2.2), where $\overset{\infty}{/}$ is encoded by `lazyflat`, and $*$ by `lazymap`. These SML functions define operators on lazy lists similar to those used in the semantic functions for infinite lists. A further interesting function is `TRec`, which implements the ArcAngel recursion construct.

```
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 tactics. The local constant `trec` is introduced as a recursively-defined value; it is the result of `TRec`. In defining `trec`, the recursive unfolding takes place incrementally and application of the tactic 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 immediate application of `t` to the environment and program.

An operator that needs to adjust the environment is **applies to** p **do** t. It is made available through the

SML function TAppliesTo, whose definition is given below.

```
fun TAppliesTo (patt : TERM) (t : AA_TACTIC) : AA_TACTIC =
   (fn env => (fn p =>
      let val rhs = (rhs_ref_thm p);
      val (_, tmm) = (term_match rhs patt);
      val applies_env = (map swap tmm) in
         t (override_env env applies_env) p
      end
      handle Fail msg => case (get_id msg) of
         3054 => (TFail env p) | _ => raise Fail msg));
```

This function first extracts the refined program (rhs) of the refinement theorem (p) to which the tactical is applied. It then uses the built-in ProofPower function term_match to match it against the pattern patt. The result is stored in tmm, and, providing the matching succeeds, it records the association of expressions with matched variables as a list of expression and variable pairs. The subsequent mapping of swap to this list turns around the components of the pairs; thus, applies_env yields a corresponding list of variable and expression pairs capturing the binding of (meta-) variables to terms. When finally applying the tactic t, we override the current environment with the new bindings; this last step is realised by override_env. The remaining bottom part of the code causes the tactic to fail whenever the matching does not succeed. (A failure exception with number 3054 is raised then by term_match). In such cases, the implementation ensures that the behaviour of the tactic is the same as that of TFail.

The last function that we discuss in this section is the implementation of the new **discharge** tactical (see Section 3.2). It is mostly a direct translation of the semantic description. In this translation, however, the encoding of the refinement cells as refinement theorems is not entirely trivial. First, we have to interpret the proof obligations defined in a refinement cell. In the semantics, they record assumptions that need to be discharged to establish that the associated program is a refinement (of whatever was the original program). Accordingly, in the implementation, they are assumptions of a refinement cell.

In the semantics of **discharge**, for each new proof obligation b, we define a refinement cell $(b, \langle\rangle)$, to which we apply the provided tactics (see *pob_rcells* in the definition of *discharge_rcell*). In the mechanisation, we encode this refinement cell as a refinement theorem $\Gamma \vdash b \Leftarrow b$, where the assumptions Γ are all those already in context, that is, all the proof obligations, excluding b itself. This enables their use when discharging b, and characterises, in the context of our mechanisation, the weakest proof obligation defined by b.

Secondly, in the semantics of **discharge** we construct the refinement cells in the list that defines the result of the tactic directly. More precisely, we take apart (using the function *pobs_for_comb*) the refinement cells that result from the application of the tactics for the proof obligations, and add the new proof obligations to the refinement cells resulting from the application of the tactic for the program.

In the implementation, refinement cells are encoded as theorems, and we cannot construct theorems explicitly. We, therefore, make use of a rule, namely replace_asm_rule, that (soundly) derives the refinement theorems. This rule takes two arguments: a theorem of the form $\Gamma \vdash P \Leftarrow P'$, which is that of the theorems that result from the application of a tactic to a proof obligation P, and a refinement theorem $\Gamma', P \vdash S \sqsubseteq T$ (resulting from the application of the program tactic), in which P is one of the assumptions. The rule replace_asm_rule replaces P by P' in $\Gamma', P \vdash S \sqsubseteq T$, essentially by using the cut rule after moving P' into the assumptions of the first theorem. Hence, replace_asm_rule yields $\Gamma, \Gamma', P' \vdash S \sqsubseteq T$. If P' is true, which is the case when the proviso has been discharged by the tactic, it is removed.

In the next section, we discuss how evaluation of expressions is handled in our tool.

4.3. Expression evaluation

To handle expressions in our tool using the annotation approach discussed in Section 3.3, we use conversions. These are general mechanisms (available in ProofPower and several other theorem provers) to rewrite subexpressions of some term by appealing to referential transparency.

Unlike ArcAngel laws, conversions do not work only at the top level of a program expression, but can rewrite arbitrary subexpressions. In our tool, they are used to eliminate the presence of *Eval* annotations. We, however, also provide more general support for the use of conversions as part of ArcAngel tactics (and avoid the need for their application to take place outside ArcAngel).

Like laws and tactics, conversions are declared using an SML function, `TConvDecl`. It expects the name of a conversion, a list of formal parameters, and a (ProofPower) conversion. To apply conversions, we use the function `TConv (name : string) (args : TERM list)`. Effectively, this makes available inside ArcAngel native support for rewriting. In ProofPower, the conversion language provides a very rich set of constructs.

Conversions can be used to carry out transformations that cannot be directly specified by laws in ProofPower. For instance, they are useful to perform postprocessing of refinement theorems after applying laws; an example is provided in Section 6. If a conversion fails, this results in failure of the underlying tactic.

Strictly, conversions are already accounted for by the semantics of ArcAngel, since we can think of them as families of laws. Hence we treat them as an implementation feature.

In the next section we clarify the integration of ArcAngel tactics with ProofPower's subgoal package.

4.4. Backward proofs

The embedding of ArcAngel is outside the subgoal package of ProofPower, because we cannot easily unify ArcAngel and ProofPower tactics, which drive backward proofs. We, however, support the use of ArcAngel tactics to facilitate the proof of refinement conjectures in a backward manner.

We provide a function `aa_tac`; it takes an ArcAngel tactic *atac* as parameter and lifts it into a corresponding ProofPower tactic. Its behaviour for a goal $A \sqsubseteq B$ is as follows. First, *atac* is applied. If that fails, this also results in the failure of `aa_tac` *atac*. Otherwise, the first element $\Gamma \vdash A \sqsubseteq A'$ of the list of generated refinement theorems is considered. By adding the provisos Γ as subgoals to the current proof tree, we justify the addition of $A \sqsubseteq A'$ to the goal hypotheses. The ProofPower tactic `asm_tac` achieves this for theorems without assumptions, but we use a more general version that also handles assumptions. The additional hypothesis either immediately discharges the goal if $A' = B$. Otherwise, it is used to reduce the goal to $A' \sqsubseteq B$ using transitivity of refinement, since $A \sqsubseteq A'$ and the new subgoal $A' \sqsubseteq B$ imply the initial goal $A \sqsubseteq B$. The low-level steps of this reduction are carried out by `aa_tac` using the model theorems.

We also provide an alternative lifting function, `aa_solve_tac` *atac*, which evaluates all outcomes of tactic applications to A, and selects one that discharges the goal or otherwise fails if none exists.

5. Examples of program models

In this section, we discuss the encoding and mechanisation of four specific program models that can be used with ArcAngel. In each instance we specify the semantics of the model, prove the model and monotonicity theorems for structural combinators, and configure the model with our tool. The ProofPower source for all theories is available at `http://www.cs.york.ac.uk/circus/tp/tools.html`.

5.1. Model for Morgan's calculus

We encode the semantics of Morgan programs in terms of our mechanisation of the Unifying Theories of Programming (UTP) in [ZC10c]. The UTP [HJ98] defines a general framework in which the semantics of a variety of programming and specification languages with different computational paradigms can be uniformly described. The calculus of the UTP is in essence Tarski's [Tar41], presented in a predicative style.

Because predicates P in the UTP describe relations, they are associated with an alphabet denoted by αP. For example, the predicate $x' > x$ with alphabet $\{ x, x' \}$ describes a computation that increments the value of x. By convention, undecorated variables refer to their values in an initial observation, and dashed variables to their values in subsequent observations (just like in Z).

The UTP has a notion of theory: a set of predicates over a given alphabet that fulfil certain healthiness conditions. Of interest here is the theory of designs, whose alphabets include, besides the program variables and their dashed counterparts, two boolean variables, ok and ok', to capture program termination. A design with precondition P and postcondition Q is written $P \vdash Q \mathrel{\widehat=} P \land ok \Rightarrow Q \land ok'$. Accordingly, we can observe that if a design is started in a state where P holds, it terminates in a state where Q holds.

A notable feature of UTP theories are a unified notion of sequential composition, nondeterminism, and refinement. Sequential composition is generally characterised by relational composition, and nondeterminism by disjunction of predicates. Refinement is characterised by universal (reverse) implication: $S \sqsubseteq P \mathrel{\widehat=} [P \Rightarrow S]$.

Here, $[\_]$ is the universal closure operator, namely $[P] \mathrel{\widehat=} \forall\, w \bullet P$, where w is the list of all the variables in the alphabet of P. Iteration and recursion are dealt with through fixed points.

Next, we present our mechanisation of the UTP in ProofPower-Z in [OCW06, OCW07, ZC10c], and discuss our embedding of Morgan's calculus as a restricted theory of designs.

5.1.1. Mechanisation of the UTP

Here we explain some of the core definitions of our UTP framework in ProofPower-Z. They are written in Z.

Variables and alphabets Variables are encoded by a schema type VAR, whose definition is included in Appendix B. It has components that determine the name of the variable, a number of dashes, a possible subscript, and its type as a subset of the semantic domain $VALUE$. Alphabets are introduced through a type $ALPHABET$, which is defined as all finite subsets of VAR.

Values and expressions Two free types $VALUE$ and $EXPRESSION$ capture the semantic domain of values and the syntax of expressions. The first one represents all values in the semantic universe.

$$VALUE ::= Int(\mathbb{Z}) \mid Bool(\mathbb{B}) \mid Real(\mathbb{R}) \mid \ldots \mid Set(\mathbb{F}\ VALUE) \mid Pair(VALUE \times VALUE)$$

The abstract syntax for expressions is encoded by a free type $EXPRESSION$. It supports the syntax of constant values, variables, relations, unary operators, and binary operators.

$$\begin{aligned}&EXPRESSION ::= Val(VALUE) \mid Var(VAR) \mid\\&\quad Rel(REL \times EXPRESSION \times EXPRESSION) \mid\\&\quad Fun_1(UNARY\_FUN \times EXPRESSION) \mid\\&\quad Fun_2(BINARY\_FUN \times EXPRESSION \times EXPRESSION)\end{aligned}$$

The sets REL, $UNARY\_FUN$ and $BINARY\_FUN$ define the model for relations and functions.

Alphabetised predicates The semantic model for an alphabetised predicate is a set of bindings describing the valuations of its alphabet variables that render it true. For example, the predicate $x = 1 \lor x = 2$ with alphabet $\{x\}$ is characterised by the bindings $x \leadsto 1$ and $x \leadsto 2$.

The formal definition of $ALPHA\_PREDICATE$, the set of alphabetised predicates, is given below.

$$ALPHA\_PREDICATE \mathrel{\widehat=} \{a : ALPHABET;\ bs : BINDINGS \mid (\forall\, b : bs \bullet \operatorname{dom} b = a)\}$$

The type $BINDINGS \mathrel{\widehat=} \mathbb{P}\, BINDING$ includes all sets of bindings. The set $BINDING$ contains the partial functions from variables (VAR) to values ($VALUE$) that associate values of the correct type with the respective variables. It is defined in Appendix B. The restriction $\forall\, b : bs \bullet \operatorname{dom} b = a$ ensures that we are only including bindings whose domain corresponds to the alphabet of the predicate.

Core operators Semantic functions for alphabetised predicates are defined for the logical connectives, equality, substitution, renaming, lattice operators for fixed-point constructions, and utility functions that allow us to determine whether predicates are tautologies, contradictions, or contingencies. These operators are distinguished by a subscript $\__P$. They usually explicitly construct the alphabet and binding set of the results. For example, for conjunction we have the alphabet being the union of the alphabets of the operands, and the binding set being the intersection of the binding sets of the operands after extending their domain to the common alphabet. We do not provide a detailed description of all operators here, but instead explain their rôle and arguments as needed in the text. Operators for designs are distinguished by a subscript $\__D$.

5.1.2. Encoding of Morgan's calculus

Programs in Morgan's calculus are in essence representable by UTP designs. Our encoding, therefore, reuses our mechanisation of the design theory, while imposing one additional restriction: the alphabet of the predicates has to be homogeneous, that is, the set of dashed program variables has to exactly match the set of undashed variables. In this setting, we can define all program operators of Morgan's calculus as design operators, except for logical constant blocks: we do not consider them here, although we treat them in [CWD06]. A treatment of procedures and parameters also requires further work as explained in [HJ98].

The set of predicates in our UTP theory of Morgan's programs are those in $\mathit{MORGAN\_PROGRAM}$.

$$\begin{array}{l} \mathit{MORGAN\_PROGRAM} \mathrel{\widehat=} \\ \quad \{p : \mathit{ALPHA\_PREDICATE} \mid (\exists\, th : \mathit{MORGAN\_THEORY} \bullet p \in \mathit{TheoryPredicates}\ th)\} \end{array}$$

This definition makes use of a function $\mathit{TheoryPredicates}$ which yields the predicates of a UTP theory in our semantic encoding. The constant $\mathit{MORGAN\_THEORY}$ refers to the family of all UTP theories of Morgan programs, each with a different alphabet corresponding to different sets of program variables. Hence, p is a valid (Morgan) program if it is a predicate of a Morgan theory instance $th \in \mathit{MORGAN\_THEORY}$.

Morgan operators The quintessential construct in Morgan's refinement calculus is the specification statement $w : [pre, post]$. We use its design characterisation $pre \vdash post[w_0, w \backslash w, w'] \wedge \mathbf{II}_{A \setminus w}$. The alphabet A is the set of variables occurring in either the pre or the postcondition, with possible 0 subscripts removed, and their dashed counterparts. The conjunction with $\mathbf{II}_{A \setminus w}$ ensures that variables outside the frame retain their values. For example, $x : [y \geq 0, x = x_0 + 1]$ is encoded as $y \geq 0 \vdash x' = x + 1 \wedge y' = y$.

We provide the function $\mathit{SpecStmt}_M\ (a, w, pre, post)$ to encode the specification statement. Here, a is the alphabet of the underlying design, w the frame variables, and pre and $post$ are relational predicates with suitable alphabets that provide the precondition and postcondition. The definition is in Appendix B.

Two other operators defined are Skip and Assignment. The functions we provide for this purpose are $\mathbf{II}_M\ a$ and $\mathit{Assign}_M\ (a, ns, es)$; they are just the corresponding functions $\mathbf{II}_D$ and Assign_D of the design theory encoding. The only difference is that they have a more restrictive domain and range by requiring their alphabet a to be an element of $\mathit{MORGAN\_ALPHABET}$, and furthermore guaranteeing the result to be in $\mathit{MORGAN\_PROGRAM}$ defined above. The set $\mathit{MORGAN\_ALPHABET}$ includes all valid design alphabets that encode Morgan computations. Similarly, sequential composition $p_1\ ;_M\ p_2$ of programs is simply defined as their relational composition $p_1\ ;_R\ p_2$.

The next operator we consider is the conditional $\mathbf{if}\ g_1 \rightarrow p_1 \mathbin{[\!]} g_2 \rightarrow p_2 \mathbin{[\!]} \ldots \mathbin{[\!]} g_n \rightarrow p_n\ \mathbf{fi}$. To define a semantics for it, we first introduce a type for representing guarded commands.

$$\mathit{GUARDED\_CMD} \mathrel{\widehat=} \{g : \mathit{MORGAN\_CONDITION};\ p : \mathit{MORGAN\_PROGRAM} \mid g.1 \subseteq p.1\}$$

A guarded command is represented by a pair: a guard (of type $\mathit{MORGAN\_CONDITION}$) and a program (of type $\mathit{MORGAN\_PROGRAM}$), where the alphabet of the guard is a subset of that of the program. A condition is a predicate whose alphabet includes only undashed variables.

We can define the semantics of a guarded command as $g \rightarrow (P \vdash Q) \mathrel{\widehat=} (g \Rightarrow P) \vdash (g \wedge Q)$. Similarly, a preconditioning operator can be characterised by $p \mid (P \vdash Q) \mathrel{\widehat=} (p \wedge P) \vdash Q$. In our encoding of the design theory, we have functions $g \rightarrow_D D$ and $p \mid_D D$ for these operators; they allow us to give a concise definition for a conditional. We first define a function $\mathit{GCmdsBody}$ to give semantics to the body of a conditional: a pair of sequences $(gs, cmds)$, where gs contains the guards, and $cmds$ the corresponding commands.

$$\begin{array}{|l} \mathit{GCmdsBody} : \mathit{GUARDED\_CMDS} \rightarrow \mathit{MORGAN\_PROGRAM} \\ \hline \forall\, gs : \operatorname{seq} \mathit{MORGAN\_CONDITION};\ cmds : \operatorname{seq} \mathit{MORGAN\_PROGRAM} \mid \\ \quad (gs, cmds) \in \mathit{GUARDED\_CMDS} \bullet \\ \qquad \mathit{GCmdsBody}\ (gs, cmds) = \\ \qquad\quad \mathit{if}\ \#cmds = 0 \\ \qquad\quad \mathit{then}\ \mathit{Magic}_D\ \varnothing \\ \qquad\quad \mathit{else}\ ((\mathit{head}\ gs) \rightarrow_D (\mathit{head}\ cmds)) \sqcap_D \mathit{GCmdsBody}\ (\mathit{tail}\ gs, \mathit{tail}\ cmds) \end{array}$$

The type $\mathit{GUARDED\_CMDS}$ defines restrictions that ensure that $\mathit{GCmdsBody}$ is applied to pairs $(gs, cmds)$ that are valid as a conditional body. (For instance, all programs in $cmds$ are required to be from the same Morgan theory.) The semantics of these pairs is a nondeterministic choice $g_1 \rightarrow p_1 \sqcap g_2 \rightarrow p_2 \sqcap \ldots \sqcap g_n \rightarrow p_n$. When the sequences of guards and commands are empty, the semantics is $\mathit{Magic}_D\ \varnothing$ (that is, the miraculous statement $\mathit{true} \vdash \mathit{false}$ with empty alphabet), which is the unit of nondeterministic choice. The head and tail functions yield the head and tail of a sequence. Thus, $(\mathit{head}\ gs) \rightarrow_D (\mathit{head}\ cmds)$ constructs a standalone guarded design from the first element of each sequence. The # operator gives the length of a sequence. Nondeterministic choice of designs is encoded by the $\sqcap_D$ operator as the disjunction of the predicates.

In the semantics of the conditional, we also have to model divergence when none of the guards hold. To do so, it is useful to encode a function GG that yields the disjunction of the guards of a sequence of guarded commands. The type of this function is $\mathit{GUARDED\_CMDS} \rightarrow \mathit{MORGAN\_CONDITION}$.

Given all the above definitions, the conditional can now be concisely specified as shown below.

$$\begin{array}{|l} \mathit{if}_M\ \_\ \mathit{fi}_M : \mathit{GUARDED\_CMDS} \rightarrow \mathit{MORGAN\_PROGRAM} \\ \hline \forall\, \mathit{gcmds} : \mathit{GUARDED\_CMDS} \bullet \mathit{if}_M\ \mathit{gcmds}\ \mathit{fi}_M = (\mathit{GG}\ \mathit{gcmds})\ |_D\ (\mathit{GCmdsBody}\ \mathit{gcmds}) \end{array}$$

We use the semantic characterisation $(\exists\, i \bullet g_i) \mid (g_1 \rightarrow p_1 \sqcap g_2 \rightarrow p_2 \sqcap \ldots \sqcap g_n \rightarrow p_n)$. The choice of standalone guards yields nondeterministic execution of any programs p_i whose guard g_i is enabled, and the precondition forces the statement to abort if all guards are false.

A second operator that expects a sequence of guarded commands is the iteration construct, given by **do** $g_1 \rightarrow p_1 \mathbin{[\!]} g_2 \rightarrow p_2 \mathbin{[\!]} \ldots \mathbin{[\!]} g_n \rightarrow p_n$ **od**. Its meaning is defined in terms fixed points as usual.

The UTP already provides facilities for variable declaration in the theory of relations, and they are directly reused to encode declarations for Morgan programs as a function var_M.

This completes the presentation of our encoding of Morgan's calculus; omitted definitions are in Appendix B. The next section explains how we integrate this model into the ArcAngel core implementation.

5.1.3. Configuration of the program model

For the configuration of an ArcAngel model, we first have to provide a refinement and an equivalence relation. For Morgan's model, refinement is encoded by the following relation on alphabetised predicates.

$$\begin{array}{|l} \_ \sqsubseteq \_ : \mathbb{P}(\mathit{ALPHA\_PREDICATE} \times \mathit{ALPHA\_PREDICATE}) \\ \hline \forall\, p_1, p_2 : \mathit{ALPHA\_PREDICATE} \bullet p_1 \sqsubseteq p_2 \Leftrightarrow \alpha\, p_1 = \alpha\, p_2 \wedge \mathit{Tautology}\,(p_2 \Rightarrow_P p_1) \end{array}$$

The *Tautology* function determines whether an alphabetised predicate is a tautology, in other words equal to $\mathit{True}_P\ a$ for some alphabet a, where True_P encodes the predicate *true*. The function α yields the alphabet of a predicate, that is, it selects its first component. Equivalence $p_1 \equiv p_2$ is defined as mutual refinement.

As explained in Section 3.1, we also require a collection of model theorems to hold for the two relations. They are the ones in Table 3 with T being *ALPHA_PREDICATE*. We have proved them in ProofPower-Z.

The model is configured by invoking the `add_model` SML function.

```
add_model ("Morgan", ⓩ(_ ≡ _)⌝, ⓩ(_ ⊑ _)⌝, typeof ⓩX ⦂ ALPHA_PREDICATE⌝,
   ref_refl_thm, ref_trans_thm, eq_def_thm);
```

The quotes ⓩ and ⌝ are used to invoke the Z-term parser to treat the expression in quotes. The fourth parameter provides the ProofPower type of the terms that encode programs in the model. It is obtained by enquiring the type of a free variable X whose type as a Z term is explicitly constrained to *ALPHA_PREDICATE* using the operator ⦂. Finally, in the example above, the values of the variables `ref_refl_thm`, `ref_trans_thm`, and `eq_def_thm` are assumed to be the required model theorems.

The second part of the model consists of the definition of structural combinators. Here, we require the construction of unary, binary, and n-ary structural combinators. Unary structural combinators are introduced for the (**var** $x \bullet p$) construct, which supports declaration of local variables.

For unary structural combinators, the constructor function requires the name of the operator for which a combinator is to be provided, the position of the program argument to which tactics are to be applied, and a monotonicity theorem for refinement (and equivalence) as explained in Section 3.1.2. To give an example, `TSCvar`, which implements the structural combinator for variable declarations, is configured as follows.

```
val TSCvar = MakeUnaryTSC("TSCvar", ⓩvar_M⌝, 2, var_ref_mon_thm, var_eq_mon_thm);
```

The program argument of the var_M function is at the second position, as indicated by the argument 2 above. The monotonicity theorems for refinement and equivalence are provided by `var_ref_mon_thm` and `var_eq_mon_thm`; the theorem for refinement is in Appendix B. Along the same lines, we use `MakeUnaryTSC` to provide structural combinators for the val_M, res_M and vres_M functions.

To cater for sequential composition, encoded by the function $(\_ \,;_M \_)$, we use a different constructor function `MakeBinaryTSC` to provide the binary (infix) structural combinator `TSCSeq`. In this case, we have to specify two operand positions (one for each program argument), and the monotonicity theorem has to be of a slightly different shape as to capture monotonicity in both arguments.

Finally, the combinators for conditional **if** ... **fi** and iteration **do** ... **od** deal with a variable number of tactics depending on the number of guarded commands. They are n-ary combinators, and for their construction we provide the function `MakeNaryTSC`. In this case, the monotonicity theorem has a more elaborate shape (see Section 3.1). We include below the theorem used in the refinement of the conditional statement, an n-ary program constructor whose corresponding structural combinator is specified using `MakeNaryTSC`.

$$\begin{aligned}&\vdash \forall\, gs : \mathrm{seq}\, \mathit{MORGAN\_CONDITION};\ cmds, cmds' : \mathrm{seq}\, \mathit{MORGAN\_PROGRAM} \mid \\ &\quad (gs, cmds) \in \mathit{GUARDED\_CMDS} \land \#cmds = \#cmds' \land \\ &\quad cmds \in \mathit{WF\_MORGAN\_PROGRAM\_SEQ} \land cmds' \in \mathit{WF\_MORGAN\_PROGRAM\_SEQ} \bullet \\ &\qquad (\forall\, i : \mathrm{dom}\, cmds \bullet cmds(i) \sqsubseteq cmds'(i)) \Rightarrow \mathit{if}_M\ (gs, cmds)\ \mathit{fi}_M \sqsubseteq \mathit{if}_M\ (gs, cmds')\ \mathit{fi}_M\end{aligned}$$

This is the encoding of the theorem in Section 3.1.2. Its proof is by induction over the length of the sequences.

The constructor function `MakeNaryTSC`, which we use to implement `TSCiffi`, the structural combinator for a conditional, has arguments similar to those of the function `MakeUnaryTSC`. The given position is that of the program sequence (*cmds* above) as an argument of the program constructor: $\mathit{if}_M\_\ \mathit{fi}_M$, for instance.

The three constructor functions `MakeUnaryTSC`, `MakeBinaryTSC`, and `MakeNaryTSC` for structural combinators have been sufficient to obtain a full coverage of the tactic language for ArcAngel.

5.2. Model for generalised substitutions

Our second program model is for the generalised substitution language (GSL) of the B formalism [Abr96]; it is based on a weakest predicate-transformer semantics. Our encoding illustrates how we use our tool in the context of a deep language embedding. We also present an example here of a hybrid structural combinator.

5.2.1. Encoding of the GSL semantics

As in the previous example, we require Z types for the variables and values in the semantic universe. We call them *VAR* and *VAL* here. Whereas *VAR* is introduced as a given type, we again use a free type for *VAL*, albeit with only two type constructor functions *Int* and *Bool*. Further value constructors may be introduced as required. For the sake of the example, the two are sufficient.

Valuations of the variables are captured by total functions from *VAR* to *VAL*. This gives rise to a notion of state, and the underlying type *STATE* is simply equated with $\mathit{VAR} \rightarrow \mathit{VAL}$. With it, we are able to define a semantic notion of expressions and predicates. Expressions are modelled by functions from *STATE* to *VAL*, and predicates by functions from *STATE* to *BOOL*. Here, *BOOL* is the HOL type for predicates.

$$\mathit{EXPR}\ \widehat{=}\ \mathit{STATE} \rightarrow \mathit{VAL} \quad \text{and} \quad \mathit{PRED}\ \widehat{=}\ \mathit{STATE} \rightarrow \mathit{BOOL}\quad .$$

Finally, the type $\mathit{PRED\_TRANS}\ \widehat{=}\ \mathit{PRED} \rightarrow \mathit{PRED}$ is introduced for predicate transformers, and provides the semantic model for commands. We do not characterise predicate transformers syntactically, by functions on program syntax, but semantically as functions on the semantic model for predicates.

The syntax of the GSL is encoded by a free type *CMD*, whose definition we include below.

$$\begin{aligned}&\mathit{CMD} ::= \mathit{SkipCMD}\\ &\quad \mid \mathit{AssignCMD}\,(\mathit{VAR} \times \mathit{EXPR})\\ &\quad \mid \mathit{SeqCMD}\,(\mathit{CMD} \times \mathit{CMD})\\ &\quad \mid \mathit{GuardCMD}\,(\mathit{PRED} \times \mathit{CMD})\\ &\quad \mid \mathit{PreCMD}\,(\mathit{PRED} \times \mathit{CMD})\\ &\quad \mid \mathit{ChoiceCMD}\,(\mathit{CMD} \times \mathit{CMD}) \mid \mathit{UChoiceCMD}\,(\mathit{VAR} \times \mathit{CMD})\end{aligned}$$

The type constructor *SkipCMD* constructs the **skip** command, *AssignCMD* an assignment $x := E$, *SeqCMD* the sequential composition $S\ ;\ T$, *GuardCMD* a guarded command $g \longrightarrow S$, *PreCMD* a preconditioned command $p \mid S$, *ChoiceCMD* the choice $S\ [\!]\ T$, and *UChoiceCMD* an unbounded choice $@\, v \bullet S$. For each syntactic command we introduce a semantic function that specifies its semantics as an element from *PRED_TRANS*. For example, for *AssignCMD* we have the following semantic definition.

$$\begin{array}{|l} \mathit{Assign} : \mathit{VAR} \times \mathit{EXPR} \rightarrow \mathit{PRED\_TRANS} \\ \hline \forall\, n : \mathit{VAR};\ e : \mathit{EXPR} \bullet \mathit{Assign}\,(n, e) = (\lambda\, p : \mathit{PRED} \bullet (\lambda\, s : \mathit{STATE} \bullet p\,(s \oplus n \mapsto e(s))))\end{array}$$

Similar definitions exist for the remaining program constructors, however, we do not include them here. They implement the standard predicate-transformer semantics of GSL [Abr96].

With the above, we define the denotational semantics of GSL as a function wp mapping commands to predicate transformers. Its standard recursive definition is sketched below.

$$
\begin{array}{|l}
wp : CMD \rightarrow PRED\_TRANS \\
\hline
wp\ SkipCMD = Skip \land \\
(\forall n : VAR;\ e : EXPR \bullet wp\,(AssignCMD\,(n, e)) = Assign\,(n, e)) \land \\
(\forall c_1, c_2 : CMD \bullet wp\,(SeqCMD\,(c_1, c_2)) = Seq\,(wp\ c_1, wp\ c_2)) \land \\
\ldots
\end{array}
$$

To conclude the model, we lastly consider refinement of generalised substitutions. For this, we first introduce implication of predicates and predicate transformers in the semantic model. The predicate implication $p_1 \Rightarrow_{PR} p_2$ holds if p_1 implies p_2 in every state, that is $\forall s : STATE \bullet p_1\ s \Rightarrow p_2\ s$, and predicate transformer implication $pt_1 \Rightarrow_{PT} pt_2$ holds if $pt_1\ p \Rightarrow_{PR} pt_2\ p$ for every predicate p. Refinement of commands is subsequently introduced as the implication of their corresponding transformers: $c_1 \sqsubseteq c_2 \Leftrightarrow (wp\ c_1) \Rightarrow_{PT} (wp\ c_2)$.

The model we present does not consider the frame of a generalised substitution, that is the set of variables that a command may alter. This is not a limitation since we do not treat parallel composition.

5.2.2. *Configuration of the program model*

As before, we first prove the model theorems in Table 3, this time with CMD as the program type T. Equivalence is again just defined as mutual refinement. Unlike in the previous example, equivalence here does not imply that the commands are mathematically the same object, since two different commands can have the same semantics. This illustrates the need for our tool to maintain and provide means to dynamically configure the equivalence relation, and not just assume it to be equality of the respective entities in HOL.

The model is configured as before by invoking the `add_model` SML function.

```
add_model ("GSL", ⓩ(_ ≡ _)⌝, ⓩ(_ ⊑ _)⌝, typeof ⓩX ⦂ CMD⌝,
   wp_ref_refl_thm, wp_ref_trans_thm, wp_eq_def_thm);
```

The variables `wp_ref_refl_thm`, `wp_ref_trans_thm` and `wp_eq_def_thm` record the model theorems. The fact that the model is deep does not impinge in its configuration or the functionality of our tool, as the latter is not at all concerned with what kind of object it transforms; this reflects the spirit of ArcAngel.

In the GSL model, we have structural combinators for sequence, guarded statements, preconditioned statements, nondeterministic choice, and unbounded choice. They are either unary or binary, but the combinator for preconditioned statements is a hybrid. To configure it, we require an additional program model for predicates, where refinement is identified with implication, and equivalence with bi-implication. We observe that this is different from the boolean program model discussed in Section 3.1.3.

In this case, it is simple to prove the model theorems using standard properties of implication. We can then configure the model as shown below; the program type is the set $PRED$.

```
add_model ("PRED", ⓩ(_ ⇔_P _)⌝, ⓩ(_ ⇒_P _)⌝, typeof ⓩX ⦂ PRED⌝,
   PRED_refl_thm, PRED_ref_trans_thm, PRED_eq_def_thm);
```

As before, the variables `PRED_ref_refl_thm`, `PRED_ref_trans_thm` and `PRED_eq_def_thm` record the model theorems. The monotonicity theorem for the precondition command is now specified as follows.

$$\vdash \forall p, p' : PRED;\ c, c' : CMD \bullet p \Rightarrow_P p' \land c \sqsubseteq c' \bullet PreCMD\,(p, c) \sqsubseteq PreCmd\,(p', c')$$

A note on monotonicity proofs in the GSL model is that they require monotonicity of wp. Formally, if $p \Rightarrow q$ then $wp\,(c, p) \Rightarrow wp\,(c, q)$ for $p, q : PRED$ and $c : CMD$. This is proved by induction over the type CMD.

The combinator for $PreCMD$ is configured as a binary combinator via the SML function call below.

```
val TSCPreCMD =
   MakeBinaryTSC "TSCPreCMD" ⓩPreCMD⌝ ((fst o dest_PreCMD), (snd o dest_PreCMD))
      (PreCMD_eq_mon_thm, PreCMD_ref_mon_thm);
```

The pair of functions provided as the third argument are the destructor functions to isolate the program arguments: `dest_PreCMD` generally destroys a preconditioned term. Also, `PreCMD_eq_mon_thm` and `PreCMD_ref_mon_thm` are the monotonicity theorems.

This shows that no special treatment is required for hybrid combinators. Our tool dynamically infers the program models of the operands of the program operators to which a structural combinator is applied.

5.3. Model for Z operation refinement

Z and its schema calculus have been designed for specification, not program refinement. Later, a calculus in Morgan's style, ZRC, has been developed for Z [CW99]. It is justified using a weakest-precondition semantics, so support for the use of tactics in ZRC can be provided in the way already discussed and exemplified.

It remains, however, that most of the Z schema calculus operators, among them conjunction, disjunction, and sequence, are not monotonic with respect to refinement. In [Gro02], Groves identifies constraints under which monotonicity can be ascertained. This makes the schema calculus amenable to compositional refinement. In the following we first formalise the notion of Z operation refinement, and then report on the configuration of structural combinators that exploit the restricted monotonicity theorems in [Gro02].

The support for Z in ProofPower-Z is not deep enough to allow us to consider the set of all state schemas, or of all operation schemas. For the sake of the example, we use a trivial state schema $STATE \mathrel{\widehat=} [x : \mathbb{N}]$ with a true invariant and single component, and consider a rather restricted model of programs as Z operations on $STATE$ (without inputs or outputs). We can provide a proper model for Z in ProofPower, by encoding its relational semantics for instance, but for the point of our example here, what we propose is enough.

In accordance with the Z relational semantics, operations are represented by sets of bindings (records). The constant $STATE\_OP \mathrel{\widehat=} \mathbb{P}(\Delta\, STATE)$ defines the program model in our simple example. Bindings in $STATE\_OP$ include all components of $STATE$ as well as their primed versions. Refinement is defined in the usual way [Gro02]. We define it in ProofPower-Z as follows; we use the Z **pre** operator directly available.

$$\_ \sqsubseteq \_ : \mathbb{P}\,(STATE\_OP \times STATE\_OP)$$

$$\begin{array}{l} \forall\, p_1, p_2 : STATE\_OP \bullet p_1 \sqsubseteq p_2 \Leftrightarrow \\ \quad (\forall\, STATE \bullet \mathbf{pre}\ p_1 \Rightarrow \mathbf{pre}\ p_2) \land (\forall\, \Delta STATE \mid \mathbf{pre}\ p_1 \bullet p_2 \Rightarrow p_1) \end{array}$$

Equivalence, as before, is just mutual refinement. Since all schema calculus operators are available in ProofPower-Z, it is not necessary to define them again. They are not Z functions; they exist as HOL functions within the formalisation of Z inside the ProofPower HOL kernel. Our tool is designed to deal with both Z and HOL functions at the level of the program models, so can readily cope with this.

The program model is configured as before; the model theorems are easily proved by exploiting the default reasoning support for proofs about Z schemas in ProofPower-Z. The interesting part of this example are the structural combinators, since the monotonicity theorems require additional provisos on the program arguments. Some operators have several combinators to reflect the various monotonicity results. For instance, the following monotonicity theorem holds for schema disjunction.

$$\begin{array}{l} \vdash \forall\, p_1, p_1' : STATE\_OP;\ p_2, p_2' : STATE\_OP \mid \\ \qquad (\forall\, \Delta STATE \bullet \mathbf{pre}\, p_1 \land p_2' \Rightarrow p_1 \lor p_2) \land (\forall\, \Delta STATE \bullet \mathbf{pre}\, p_2 \land p_1' \Rightarrow p_1 \lor p_2) \bullet \\ \qquad\quad p_1 \sqsubseteq p_1' \land p_2 \sqsubseteq p_2' \Rightarrow (p_1 \lor_s p_2) \sqsubseteq (p_1' \lor_s p_2') \end{array}$$

The subscript $\_{}_s$ is used in ProofPower-Z for operators of the schema calculus. This is used to configure a structural combinator $TSCSchOr1$ for schema disjunction. Applying the combinator raises the provisos $(\forall\, \Delta STATE \bullet \mathbf{pre}\, p_1 \land p_2' \Rightarrow p_1 \lor p_2)$ and $(\forall\, \Delta STATE \bullet \mathbf{pre}\, p_2 \land p_1' \Rightarrow p_1 \lor p_2)$, where p_1 and p_2 are the original operations and p_1' and p_2' the operations transformed by the combinator tactics. These proof obligations persist as assumptions of the constructed refinement theorem, and have to be discharged in order to obtain an unconditional theorem. (To achieve that within ArcAngel we can use the `TDischarge` tactical and laws and combinators of the predicate program model.)

Several laws are presented in [Gro02] that establish monotonicity of conjunction and disjunction, and they differ by their provisos. As all of them may be useful in a given context, we have a structural combinator for each monotonicity theorem. The approach of Groves is relatively unconventional; our tool enables us to safely utilise it and further explore its possibilities. Small examples are presented in Section 6.

5.4. Model for *Circus* and CSP

A more specialised tactic language for refinement based on ArcAngel is ArcAngel*C*. It is tailored for a state-rich process algebra called *Circus* [OCW07], which is based on Z and the CSP process algebra [Ros98]. ArcAngel*C* extends ArcAngel in two fundamental ways: a program model that includes three constructs that represent computations (namely, actions, processes, and programs) to which we can apply refinement, and a collection of novel structural combinators. Since our tool treats the program model as a separate component, it can be used as basis to support ArcAngel*C* tactics, as we illustrate below.

For the program model, we can use our existing embedding of *Circus* and CSP [ZC10c], which is based on the mechanisation of the UTP presented in Section 5.1.1. We can even reuse the proofs of the model theorems for refinement and equivalence. In the following we focus our discussions on structural combinators that are not handled using the functions we have already encountered.

ArcAngel*C* provides program and process structural combinators. For example, **program** $(name, tac)$ applies to a *Circus* program, which is a list of process paragraphs. The result is the application of the tactic tac to the process $name$ defined in the program. The position of the program component to which the tactic is applied is not fixed, but identified by the name of the component. For this reason, this structural combinator cannot be simply specified using the unary, binary, or n-ary constructor functions presented before.

To facilitate support for the definition of this kind of combinator, we provide a general recursive structural combinator `TSCRec`. It takes as parameter a testing function `test` that applies to any HOL term and characterises those to which a base tactic `base_tac`, also given as parameter, is to be applied.

```
fun TSCRec (test : TERM -> bool) (base_tac : AA_TACTIC) : AA_TACTIC;
```

Unlike other combinators, `TSCRec` does not apply to a particular program operator. Instead, it propagates itself recursively through any term, provided a combinator for its top operator is registered, and while the testing function yields false. If the test succeeds, `base_tac` is applied instead. We use this combinator to target the application of a tactic to a particular subterm. It is analogous to recursive rewriting tactics in theorem provers, but operates within ArcAngel constructing sound refinements rather than equalities.

In our encoding of the semantics of *Circus* processes as predicates, we remove all references to processes by applying the copy rule and the fixed point operator. References thus substituted are tagged by a function $LocalProcess_P\ (name, p)$ defined as the projection in its second argument; its first argument records the original name of the process. It is trivially monotonic, and a unary structural combinator `TSCLocalProcess` applies a tactic to its argument p. A testing function `fun LocalProcessTest (name : string)` determines if a term is of the form $LocalProcess_P\ (name, p)$ and $name$ is syntactically equal to `name`. With that, the structural combinator **program** $(name, tac)$ is concisely defined as follows.

```
val TSCProgram (name : string) (tac : AA_TACTIC) =
   TSCRec (LocalProgramTest name) (TSCLocalProcess tac);
```

If the test succeeds, `tac` is applied to the body of the named process. Otherwise, the application of the tactic is propagated through the entire program. In consequence, only instances of $LocalProcess_P\ (name, p)$ where $name$ equals the `name` parameter of `TSCProgram` are affected. They, however, may be located anywhere in the top-level program. Our previous work in [ZOC09] does not envisage support for the above structural combinator, and with its implementation we can now claim full support for ArcAngel*C* via our tool.

Our experiments with the ArcAngel*C* encoding have revealed that a large number of provisos are raised by model, law, and monotonicity theorems. This is due to the rich structure of *Circus* models, which involves three (nested) constructs to define computations. As a consequence, our tool provides special treatment for well-definedness constraints as part of the refinement theorem. This supports refinement theorems of the from $wd\ A \vdash A \sqsubseteq B \wedge wd\ B$, where $wd\ A$ and $wd\ B$ are well-definedness constraints. Our implementation manages and discharges such constraints, where possible, by default as part of the mechanics of applying tactics. Details of our derived ArcAngel*C* tool can be found in [ZC10a].

The next section provides several examples that illustrate the use of our implementation of ArcAngel to carry out program refinements using the embeddings in this section.

Law name	Definition	Provisos
strPost$(post')$	$w : [pre, post] \sqsubseteq w : [pre, post']$	$post' \Rightarrow post$
seqComp(mid)	$w : [pre, post] \sqsubseteq w : [pre, mid]\ ;\ w : [mid, post]$	mid and $post$ have no free initial variables
assign(w, E)	$w : [pre, post] \sqsubseteq w := E$	$pre \Rightarrow post[w \backslash E]$
assignIV(w, E)	$w, x : [pre, post] \sqsubseteq w := E$	$x = x_0 \wedge pre \Rightarrow post[w \backslash E]$

Table 4. Some laws of Morgan's calculus.

6. Tactic examples

We present here a mechanisation of the ArcAngel tactic in [OCW03]. It refines a specification statement into a loop preceded by an initialisation of the variables modified by the iteration. Its declaration is given below.

```
Tactic takeConjAsInv $(invBound, lstVar, lstVal, variantExp) \widehat{=}$
  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) $inv \wedge invBound \wedge \neg\, guard \Rightarrow inv \wedge \neg\, guard$    (from strPost)
     (2) $pre \Rightarrow (inv \wedge invBound)\,[lstVar \backslash lstVal]$    (from assign)
  generates
    $lstVar := lstVal$ ;
    do $guard \rightarrow w : \left[\begin{array}{l} inv \wedge invBound \wedge guard, \\ inv \wedge invBound \wedge 0 \leq variantExp < variantExp[w \backslash w_0] \end{array}\right]$ od
end
```

The iter law is presented in Section 3.3. It refines a specification statement $w : [inv, inv \wedge \neg\, GG]$ into an iteration **do** $[\!]\ i \bullet G_i \rightarrow w : [inv \wedge G_i, inv \wedge 0 \leq V < V_0]$ **od**. 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 strictly decreasing expression V serving as the loop variant.

The tactic is parameterised by an additional invariant constraint $invBound$ that bounds the range of the variables used in the variant, the (list of) variable(s) $lstVar$ to be initialised, the (list of) corresponding expression(s) $lstVal$ to be assigned, and the variant expression $variantExp$. The use of **applies to _ do _** requires the program to be of the form $w : [pre, inv \wedge \neg\, guard]$ for the tactic to be applicable. Its body applies the laws strPost, seqComp, assign and iter; the definitions of the first three are given in Table 4.

As explained in Section 5.1.2, GG is an abbreviation for a predicate: semantically, a function mapping sequences of predicates to predicates. In the theorem below for iter we use the encoding of GG in Appendix B.

$$\begin{array}{l} \vdash \forall\, a : MORGAN\_ALPHABET;\ w : \mathrm{seq}\, M\_VAR;\ inv : MORGAN\_CONDITION; \\ \quad guards : \mathrm{seq}\, MORGAN\_CONDITION;\ V : EXPRESSION \mid \\ \qquad (a, w, inv, inv \wedge_P \neg_P (GG\ guards)) \in WF\_SpecStmt_M \wedge \\ \qquad (a, w, inv, guards, V) \in WF\_iterLawGCmds \bullet \\ \quad\ \ SpecStmt_M\,(a, w, inv, inv \wedge \neg\,(GG\ guards)) \sqsubseteq do_M\ iterLawGCmds\,(a, w, inv, guards, V)\ od_M \end{array}$$

The two parameters of the law are provided through $guards$ specifying the sequence of guards, and V specifying the variant expression. In the antecedent, the memberships to $WF\_SpecStmt_M$ and $WF\_iterLawGCmds$ constitute well-definedness conditions. The latter establishes, for example, that the alphabet of every predicate is a subset of the alphabet a of the specification statement. The rôle of $iterLawGCmds$ is to construct a sequence of guarded commands whose guards G_i are obtained from *guards*, and whose command is of the form $G_i \rightarrow w : [inv, inv \wedge (0 \leq V < V_0)]$. We omit its encoding here, but include it in Appendix B.

Certain steps of the application of iter, namely, the evaluation of the applications of the functions GG and *iterLawGCmds*, are implicitly assumed to be syntactic operations. The necessary pre and postprocessing steps require additional effort that is not catered for in the mere application of the law. We solve this problem with the support for conversions in tactics as discussed in Section 3.3. Here, in particular, two conversions are required: one that rewrites a disjunction of predicates into an application of GG to a sequence, and another that eliminates *iterLawGCmds* using its (recursive) definition after application of the law.

To implement the conversions in ProofPower, we first realise general conversions that rewrite applications of recursively-defined functions over sequences, providing the arguments are explicitly enumerated sequences (sequence displays). The introduction of GG is realised by a general conversion that gradually rewrites each disjunction of a predicate, starting with the right-most one. It uses laws such as $GG\ \langle\rangle = \mathit{false}$ and $p_1 \vee GG\ \langle p_2, p_3, \ldots\rangle = GG\langle p_1, p_2, \ldots\rangle$. The elimination of *iterLawGCmds* is realised by the general conversion for rewrite of recursive functions. These conversions are declared for use in our implementation using the `TConvDecl` function. Namely, we declare the conversion for preprocessing as follows.

```
TConvDecl "iter_pre" iter_GG_intro_conv;
```

The conversion `iter_GG_intro_conv` applies `GG_intro_conv`, the conversion to introduce GG, to a particular term in a specification statement. The conversion tactic is `iter_pre`. Similarly, we declare a conversion `iter_post` to eliminate application of the function *iterLawGCmds* from any term.

Finally, we define a tactic that executes the law iter and performs the pre and postprocessing as follows.

```
 TTacDecl "iter" [ⓩV ⦂ EXPRESSION⌝]
     ((TConv "iter_pre" []) TSeq (TLaw "iter" [ⓩV⌝]) TSeq (TConv "iter_post" []));
```

Since the sequence of guards can be matched by the right-hand of the iter law, it is not strictly required as an argument, hence the only argument is the variant expression. It is propagated to the law application. Conversion tactics may in general have arguments, however the ones used here do not require any.

The encoding of iter illustrates how the new conversion mechanism supports the encoding of more general laws in our implementation, and accommodates the implicit steps in processing law applications. We omit the encoding of the two remaining laws used in takeConjAsInv as it is conventional.

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

```
TTacDecl "takeConjAsInv" [
    ⓩinvBound ⦂ ALPHA_PREDICATE⌝ , ⓩlstVar ⦂ seq  M_VAR⌝ ,
    ⓩlstVal ⦂ seq  EXPRESSION⌝ , ⓩvariantExp ⦂ EXPRESSION⌝] (
        TAppliesTo ⓩSpecStmt_M (a,  w,  preC,  invConj ∧_P  (¬_P  guard))⌝ TDo (
            (TLaw "strPost" [ⓩ(invBound ∧_P  invConj) ∧_P  (¬_P  guard)⌝]) TSeq
            (TLaw "seqComp" [ⓩinvBound ∧_P  invConj⌝]) TSeq
            ((TLaw "assign" [ⓩlstVar⌝ , ⓩlstVal⌝]) TSCSeq (TLaw "iter" [ⓩvariantExp⌝]))
        )
    );
```

As with law declarations, tactic declarations have to provide a name for the tactic, and then a list of terms for the formal arguments: variables 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 SML functions.

Variables introduced in a tactic declaration via `TTacDecl` or the `TAppliesTo` construct become local and can be used in the body of the respective construct, for example, when specifying arguments of law and tactic applications. For instance, the local variable *invBound* is introduced by the tactic declaration, and used in the specification of the arguments for the applications of the laws strPost and seqComp. Similarly, the `TAppliesTo` construct introduces the local meta-variables a, w, $preC$, $invConj$, and $guard$, of which $invConj$ and $guard$ are furthermore used (in arguments) in some of the law applications.

To illustrate application of the tactic, we first create a program that encodes the specification statement

$$q, r : [a \geq 0 \wedge b > 0, (a = q * b + r \wedge 0 \leq r) \wedge \neg\ r \geq b]$$

It calculates the quotient and remainder of two numbers a and b; they are recorded in the variables q and

r. Its encoding is slightly tedious using the functions in Section 5.1.1.

To apply takeConjAsInv to the above program, we use the function `aa_rule` as shown below, where `ss` refers to the encoding of the specification statement above. The function `aa_rule` expects an ArcAngel tactic and a program expression, and creates an initial refinement theorem $\vdash P = P$ to which it applies the tactic.

`aa_rule (TTactic "takeConjAsInv" [`$\ulcorner_{\mathrm{Z}} \mathit{True}_P\ u \urcorner$`,` $\ulcorner_{\mathrm{Z}} \langle q,\ r\rangle \urcorner$`,` $\ulcorner_{\mathrm{Z}} \langle \mathit{Val}(\mathit{Int}(0)), \mathit{Var}(a)\rangle \urcorner$`,`$\ulcorner_{\mathrm{Z}} \mathit{Var}(r) \urcorner$`]) ss`

Analogously to `TLaw`, the `TTactic` function applies declared tactics. The parameters provided 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*. They encode expressions in the semantic model. The result is the following refinement theorem.

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

which encodes the program refinement

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

For readability, the assumptions of the theorem have been omitted. Most of them carry constraints regarding the well-definedness of operator applications, which are accumulated through application of the laws. In practice, we anticipate that most of these assumptions are provable automatically without any intervention (and the work in [ZC10a] develops an approach to discharge them as we go along.) The remaining assumptions encapsulate the provisos of the laws. For example, we find the following assumption encoding the first proof obligation of takeConjAsInv, that is, the proviso of strPost.

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

The proof of this proviso can be automated using the new **discharge** tactic. For this, we require the fact that $\mathit{Tautology}(\mathit{True}_P\ a\ \wedge_P\ p\ \wedge_P\ q \Rightarrow_P (p \wedge_P q))$ holds, for all p and q from *ALPHA_PREDICATE* and a from *ALPHABET*; this is a theorem called `tautology_law` in the mechanisation. With that, in takeConjAsInv we can apply the strPost law as part of an application of the **discharge** tactic as follows.

```
(TDischarge (TLaw "strPost" [$\ulcorner_{\mathrm{z}} (invBound \wedge_P invConj) \Rightarrow_P (\neg_P \;\; guard) \urcorner$])
  [TSkip, TSkip, TSkip, TSkip, TSkip, TSkip, TSkip, TSkip, TLaw "tautology_law" []])
```

The `TSkip` tactics are for the proof obligations that we do not touch; they are all well-definedness constraints of the semantic model. The last tactic applies the previous law. This results in the proof obligation being removed. This tactic always succeeds due to the fixed structure of the provisos.

The refinement theorem obtained can be subject to further tactic applications. For example, using an encoding of the law assignIV (see Table 4), we can refine the guarded command within the loop to the assignment $q, r := r, r - b$. The encoding of the law is similar to the ones we already presented. It is applied using structural combinators for sequential composition and loop.

```
(TSkip TSCSeq (TSCdood
   [TLaw "assignIV" [$\ulcorner_{\mathrm{z}} \langle q, r \rangle \urcorner$ , $\ulcorner_{\mathrm{z}} \langle Var(r), Fun_2(\_ -_V \_), Var(r), Var(b)) \rangle \urcorner$]]));
```

Since there is only one guarded command, the sequence of tactics supplied to `TSCdood` has only one element. The tactic `TSkip` ensures that the first operand of the sequential composition remains unchanged. The resulting refinement conjecture then encodes the following program refinement.

$$q, r : [a \geq 0 \wedge b > 0, a = q * b + r \wedge 0 \leq r \wedge \neg\, r \geq b] \;\sqsubseteq\; q, r := 0, a;\ \mathbf{do}\ r \geq b \rightarrow q, r := r, r - b\ \mathbf{od}$$

This example, although simple, already illustrates the interaction of native mechanisms, namely, the conversion framework, and our high-level tactic language. They are seamlessly integrated, and their joint use raises no issues of soundness. The gap between syntactic transformations and semantic rewriting is thus closed.

7. Conclusion

We have presented an implementation of ArcAngel that can be used to carry out refinements guaranteed to be sound. Our tool automates all the low-level steps involved in instantiating laws, using transitivity and monotonicity theorems, and so on. Although ProofPower has been our target platform, most of our work, and certainly our approach, is equally relevant to similar integration endeavours in other theorem provers. In essence, what we require is a functional language, a prover interface that permits the sound application of elementary rules of classical logic, and standard rewrite mechanisms to support the conversion tactics. Higher-order logic is needed to integrate the program model for Angel and implement **discharge**.

In the design of our tool, we have adopted a very direct translation of the ArcAngel semantics in which refinement cells in the semantic model are identified with refinement theorems in ProofPower. A faithful implementation of partial and infinite lists is achieved through the use of lazy evaluation. To acquire confidence that the design respects the semantics, we have conducted a series of tests based on the laws of ArcAngel, for which we have a proof of soundness. We have verified that tactics in the test set that are equal in the semantics also exhibit similar behaviour in the implementation. This provides some empirical evidence for the correctness of the implementation, and besides has revealed deficiencies in earlier designs.

Extensions and generalisations to ArcAngel have been suggested by the implementation of our tool. Most significantly, we have introduced the notion of a program model. This unifies the application of ArcAngel tactics to various kinds of objects, not only programs of Morgan's refinement calculus. We have identified the essential requirements for the sound application of the implementation mechanisms; they are captured by a number of required theorems for the program operators and the refinement relation.

Factoring out the program model retains the flexibility of ArcAngel as a general method for transforming terms, and provides opportunities for its use in a wider context. An additional benefit is the support to handle both proofs of equivalence and refinement. In ProofPower, the application of ArcAngel tactics results in the generation of refinement theorems that make the refinement relation explicit. Our implementation provides the strongest possible result, that is, equivalence rather than refinement, whenever possible.

Another extension supports the use of ArcAngel tactics to discharge proof obligations. To incorporate this feature seamlessly into the semantics, we use a predicative program model. It constitutes a unification of ArcAngel with its predecessor Angel, and makes proof goals amenable for reduction via ArcAngel tactics. This, in particular, allows us to use ArcAngel tactics to discharge proof obligations that are refinements themselves, and to do this without having to step outside the tool and ArcAngel tactic formalism.

A pragmatic extension has been developed to deal with the evaluation of expressions in tactics. Related to that, implicit pre and postprocessing steps in the application of laws and tactics have also been addressed. This uses our approach to integrate ArcAngel tactics with native rewrite facilities in a clean and sound way.

A final extension provides extra tactic constructors for iteration; they deal with the possibility of nontermination. Tactics that become trapped in nonterminating loops are not uncommon in theorem provers, for example, if repetitive rewrites of expressions successively succeed. In our tool, it is possible to deal with nontermination by limiting the number of recursive calls in a tactic.

We have also presented the definition of a program model for Morgan's calculus using the mechanisation of the UTP in [ZC08]. It is an extension of existing work on the UTP theory of designs, but includes constructs (like guarded conditionals and loops) and theorems specific for the use of Morgan's calculus. We have also provided full support for the ArcAngel*C* language, including program and process structural combinators.

Closely related work, apart from the refinement editor in [OXC04], is the implementation of Angel in Ergo [UW94], 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 on a formalisation of HOL. The implementation of Angel in Ergo does justice to its generality: it allows tactics to be applied not just to single goals, but sequences that may result from the application of tactics.The Ergo implementation of Angel is otherwise too specific to give heed to ArcAngel models: it assumes tactics to be applied to sequents.

The Program Refinement Tool (PRT) [CHN$^+$98] is based on Ergo and originally inherited its tactic language. Later developments use Gumtree [MNU97a, MNU97b], a tactic language based on Angel.

Groves, Nickson, and Utting present in [GNU92, Nic94] a refinement tool implemented in Prolog; it has a special emphasis on tactics. The encoding of various development patterns is in Nickson's thesis [Nic94]. The tactics are not expressed in any special language, but in Prolog. As a consequence, they use Prolog's control mechanisms and arbitrary computations in deciding what steps to take, and in constructing new components. Modifications to the program being constructed, however, are done by applying refinement rules.

The Red refinement tool is also implemented in Prolog [Vic90]. The developers of Red define a language to describe the refinement tree [VG94]. In this language, each construct of the target programming language has a corresponding construct in the transformation language. This work was the inspiration for Angel's structural combinators [MGW93]. The language's purpose, however, was not to describe tactics, but refinements. For instance, no constructs for alternation or recursion is present in the Red language.

The refinement tool presented in [Gru92] is based on the HOL theorem prover. It is programmable and we can add commands to automate refinements that are frequently repeated. For that, ML tacticals can be used. This work is extended in [BGL$^+$97] to provide a user-friendly GUI. In this tool, ML can be used to package transformations; this poses the difficulty of requiring expertise in ML. Another, proof-based tool is described by von Wright in [Wri94]. Like the aforementioned, it does not provide a high-level tactic language but requires low-level interaction with the underlying window-inference package to steer the refinement. Neither of the works address the handling of provisos arising, and high-level strategies.

More recent work has been done by the B community to support the specification and refinement-based development of complex systems. The Event-B [Abr10] notation is primarily used here, and the RODIN platform [ROD] is the state-of-the-art tool to support it. The method and tools show often impressive results in handling large developments, but they do not support well high-level strategies and transformational approaches. The work in [ITLR10] investigates the use of refinement patterns. It is not clear though how soundness of the patterns is established, and how we guard against instantiation of unsound patterns.

Future work first consists of providing a larger collection of proved laws for the refinement calculus in our UTP-based ProofPower-Z embedding. Based on that, we can also build a working collection of mechanised general-purpose tactics that can be used for practical program derivation.

A second area for future work is the development of a variety of case studies for realistic applications, with relevance to industry. In this respect, initial results concerning the mechanisation of the refinement strategy for control law implementations presented in [CC06] are reported in [OZC11]. We have a collection of tactics that automate the application of the refinement strategy, whose mechanisation builds on the framework presented here. Without the support provided by a mechanised (refinement) tactic language, automation of such a strategy would have to rely on *ad hoc* implementations. That could lead to tailored treatment of inputs and error messages, but, if soundness is to be guaranteed, use of a theorem prover is essential.

ArcAngel has so far proved to be very expressive, and can cope, for instance, with the quite elaborate refinement strategy in [CC06, OZC11]. With the use of our mechanisation, we completely eliminate the need for any interaction for application of refinement laws. Practical concerns, however, still arise from the accumulation of associated proof obligations. For the particular case study reported in [OZC11], this issue is partially addressed in [ZC10a]. Ultimately, the application of a refinement strategy at an industrial scale has to rely on both the automation afforded by the use of ArcAngel and on automation of the discharge of the associated proof obligations. Our work addresses automation in discharging proof obligations by providing a way of taking advantage of the native automation facilities of the prover without compromising soundness. In our case study, constraints on the architecture of both specifications and programs enable a very high level of automation of both refinement and proof. In a more general setting, the use of SMT solvers can be very helpful, and we are currently exploring this option as well.

Finally, a complementary strand of work is the mechanisation of the formal semantics of ArcAngel in ProofPower-Z. This will support reasoning about ArcAngel tactics. An interesting application is an integrated environment that supports the translation of semantically-encoded tactics into SML for our tool.

Acknowledgements

We thank Matthew Naylor for his 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. CNPq partially supports the work of Marcel Oliveira: grants 573964/2008-4, 476836/2009-3, and 560014/2010-4. We are also grateful to the reviewers, whose valuable suggestions improved the presentation of this work.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[Abr10] J.-R. Abrial. *Modelling in Event-B.* Cambridge University Press, May 2010.

[AC05] M. Adams and P. Clayton. ClawZ: Cost-Effective Formal Verification of Control Systems. In *Formal Methods and Software Engineering*, volume 3785 of *Lecture Notes in Computer Science*, pages 465–479. Springer, October 2005.

[ACOS00] R. Arthan, P. Caseley, C. O'Halloran, and A. Smith. ClawZ: Control laws in Z. In *3rd International Conference on Formal Engineering Methods*, pages 169–176. IEEE Computer Society Digital Library, September 2000.

[AJ05] R. Arthan and R. B. Jones. Z in HOL in ProofPower. *FACS FACTS*, 2005:39 – 55, 2005. Available at `www.bcs.org/upload/pdf/facts200503-compressed.pdf`.

[Bac87] R.-J. Back. Procedural Abstraction in the Refinement Calculus. Technical Report Ser. A, No 55, Department of Computer Science, Åbo Akademie, Finland, April 1987.

[BGL$^+$97] M. Butler, J. Grundy, T. Långbacka, R. Rukšėnas, and J. von Wright. The Refinement Calculator: Proof Support for Program Refinement. In *Formal Methods Pacific'97: Proceedings of the FMP'97*, pages 40–61. Springer, July 1997.

[CC06] A. L. C. Cavalcanti and P. Clayton. Verification of Control Systems using ***Circus***. In *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems*, pages 269–278. IEEE Computer Society, 2006.

[CHN$^+$94] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A Review of Existing Refinement Tools. Technical Report UQ-SVRC-94-8, Software Verification Research Centre, University of Queensland, Australia, June 1994.

[CHN$^+$98] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A Program Refinement Tool. *Formal Aspects of Computing*, 2(10):97–124, November 1998.

[CO06] P. Clayton and C. O'Halloran. Using the Compliance Notation in Industry. In *Refinement Techniques in Software Engineering*, volume 3167 of *Lecture Notes in Computer Science*, pages 269–314. Springer, October 2006.

[CW99] A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC—A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267—289, 1999.

[CWD06] A. L. C. Cavalcanti, J. C. P. Woodcock, and S. Dunne. Angelic Nondeterminism in the Unifying Theories of Programming . *Formal Aspects of Computing*, 18(3):288 – 307, 2006.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Series in Automatic Computation. Prentice-Hall, 1976.

[GNU92] L. Groves, R. Nickson, and M. Utting. A Tactic Driven Refinement Tool. In *Proceedings of the 5th Refinement Workshop, organised By BCS-FACS*, pages 272–297. Springer, January 1992. Available as a technical report at `http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-92-5.abs.html`.

[Gro02] L. Groves. Refinement and the Z Schema Calculus. *Electronic Notes in Theoretical Computer Science*, 70(3):70–93, 2002.

[Gru92] J. Grundy. A Window Inference Tool for Refinement. In *Proceedings of the 5th Refinement Workshop*, Workshops in Computing, pages 230–254. BCS FACS, Springer, January 1992.

[HJ98] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice Hall Series in Computer Science. Prentice Hall, February 1998.

[ITLR10] A. Iliasov, E. Troubitsyna, L. Laibinis, and A. Romanovsky. Patterns for Refinement Automation. In *Formal Methods for Components and Objects*, volume 6286 of *Lecture Notes in Computer Science*, pages 70–88. Springer, 2010.

[Mar94] A. Martin. *Machine-Assisted Theorem Proving for Software Engineering.* PhD thesis, Oxford University Computing Laboratory, Oxford, UK, 1994. Technical Monograph PRG-121.

[Mar96] A. Martin. Infinite Lists for Specifying Functional Programs in Z. In *Proceedings of Fifth Australasian Refinement Workshop*. University of Queensland, April 1996.

[MGW93] A. Martin, P. Gardiner, and J. Woodcock. Tactic Semantics and Reasoning. Technical report, Oxford University Computing Laboratory, Oxford, UK, December 1993.

[MGW96] A. Martin, P. Gardiner, and J. Woodcock. A Tactical Calculus – Abridged Version. *Formal Aspects of Computing*, 8(4):479–489, July 1996.

[MNU97a] A. Martin, R. Nickson, and M. Utting. A Tactic Language for Ergo. In *Formal Methods – Pacific 97*, Discrete Mathematics and Theoretical Computer Science, pages 186–207, July 1997.

[MNU97b] A. Martin, R. Nickson, and M. Utting. Improving Angel's Parallel Operator: Gumtree's Approach. Technical Report UQ-SVRC-97-15, Software Verification Centre, School of Information Technology, University of Queensland, Australia, December 1997.

[Mor88] C. C. Morgan. The Specification Statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[Mor98] C. C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall, 1998.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, February 1990.

[Nic94] R Nickson. *Tool Support for the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 1994.

[OC08] M. V. M. Oliveira and A. L. C. Cavalcanti. ArcAngel*C*: a Refinement Tactic Language for *Circus*. *Electronic Notes in Theoretical Computer Science*, 214:203–229, July 2008.

[OCW03] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. Woodcock. ArcAngel: a Tactic Language for Refinement. *Formal Aspects of Computing*, 15(1):28–47, July 2003.

[OCW06] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. Woodcock. Unifying Theories in ProofPower-Z. In *Unifying Theories of Programming, First International Symposium*, volume 4010 of *Lecture Notes in Computer Science*, pages 123–140. Springer, February 2006.

[OCW07] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. Woodcock. A UTP semantics for *Circus*. *Formal Aspects of Computing*, Online First, December 2007.

[OXC04] M. V. M. Oliveira, M. Xavier, and A. L. C. Cavalcanti. Refine and Gabriel: Support for Refinement and Tactics. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods*, pages 310–319. IEEE Computer Society, September 2004.

[OZC11] M. V. M. Oliveira, F. Zeyda, and A. L. C. Cavalcanti. A tactic language for refinement of state-rich concurrent specifications. *Science of Computer Programming*, 76(9):792 – 833, 2011.

[Pau96] L. C. Paulson. *ML for the Working Programmer, 2nd Edition*. Cambridge University Press, 1996.

[ROD] Event-B and the RODIN Platform. Available from `http://www.event-b.org/`.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[Tar41] A. Tarski. On the Calculus of Relations. *Journal of Symbolic Logic*, 6(3):73–89, September 1941.

[UW94] M. Utting and K. Whitwell. Ergo User Manual – An Interactive Theorem Prover, February 1994.

[VG94] T. Vickers and P. Gardiner. A Language of Refinements. Technical Report TR-CS-94-05, Computer Science Department, Australian National University, May 1994.

[Vic90] T. Vickers. An overview of a refinement editor. In *Proceedings of the Fith Australian Software Engineering Conference*, pages 39–44, 1990.

[VZC10] M. Vernon, F. Zeyda, and A. L. C. Cavalcanti. Communication Systems in ClawZ. In *Abstract State Machines, Alloy, B, and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 334 – 348. Springer-Verlag, 2010.

[WD96] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. International Series in Computer Science. Prentice Hall, July 1996.

[Wri94] J. Wright. Program refinement by theorem prover. In *Proceedings Of the 6th Refinement Workshop, organised by BCS-FACS*. Springer, January 1994.

[ZC08] F. Zeyda and A. L. C. Cavalcanti. Mechanical Reasoning about Families of UTP Theories. In *SBMF 2008, Brazilian Symposium on Formal Methods*, pages 145–160, August 2008.

[ZC10a] F. Zeyda and A. L. C. Cavalcanti. Automating Refinement of *Circus* Programs. In *Brazilian Symposium on Formal Methods*, Lecture Notes in Computer Science. Springer-Verlag, 2010.

[ZC10b] F. Zeyda and A. L. C. Cavalcanti. Encoding Circus Programs in ProofPower-Z. In *Unifying Theories of Programming 2008*, Lecture Notes in Computer Science. Springer, 2010.

[ZC10c] F. Zeyda and A. L. C. Cavalcanti. Mechanical Reasoning about Families of UTP Theories. *Science of Computer Programming*, 2010. DOI:10.1016/j.scico.2010.02.010.

[ZOC09] F. Zeyda, M. V. M. Oliveira, and A. L. C. Cavalcanti. Supporting ArcAngel in ProofPower-Z. *Electronic Notes in Theoretical Computer Science*, 2009.

A. Partial and infinite lists

The model adopted for infinite lists in ArcAngel is essentially that of [Mar96], including a few cosmetic adjustments. It is formalised in Z. Here, we only give the fundamental definitions regarding the types of lists and their refinement ordering, as well as of those operators primarily used in the paper. A comprehensive account of operator definitions on infinite lists can be found in [Mar96] and [OCW03].

Partial lists A free type *pfseq* is used to record whether a list, modelled by a finite sequence, is finite or partial.

$$\mathit{pfseq}\ X ::= \mathrm{finite}\ \langle\!\langle \mathrm{seq}\ X \rangle\!\rangle \mid \mathrm{partial}\ \langle\!\langle \mathrm{seq}\ X \rangle\!\rangle$$

We define a refinement ordering on partial and finite lists as follows.

$$\begin{array}{l}
[X] \\
\_ \sqsubseteq \_ : \mathit{pfseq}\ X \leftrightarrow \mathit{pfseq}\ X \\
\hline
\forall\, gs, hs : \mathrm{seq}\ X \bullet \\
\quad \mathrm{finite}\ gs \sqsubseteq \mathrm{finite}\ hs \Leftrightarrow gs = hs \land \\
\quad \mathrm{finite}\ gs \sqsubseteq \mathrm{partial}\ hs \Leftrightarrow \mathit{false} \land \\
\quad \mathrm{partial}\ gs \sqsubseteq \mathrm{finite}\ hs \Leftrightarrow gs \leq hs \land \\
\quad \mathrm{partial}\ gs \sqsubseteq \mathrm{partial}\ hs \Leftrightarrow gs \leq hs
\end{array}$$

Above, the relation $\leq$ denotes the standard prefix operator on sequences.

Infinite lists Infinite lists are represented by prefix-closed chains of partial lists.

The set *chain* contains all chains of partial lists.

$$\begin{array}{l}
[X] \\
\mathit{chain} : \mathbb{P}\,(\mathbb{P}\,(\mathit{pfseq}\ X)) \\
\hline
\mathit{chain} = \{c : \mathbb{P}\,(\mathit{pfseq}\ X) \mid \forall\, x, y : c \bullet x \sqsubseteq y \lor y \sqsubseteq x\}
\end{array}$$

The set *pchain* contains all chains which are also prefix-closed.

$$\begin{array}{l}
[X] \\
\mathit{pchain} : \mathbb{P}\,(\mathit{chain}[X]) \\
\hline
\mathit{pchain} = \{c : \mathit{chain}[X] \mid \forall\, x : c;\ y : \mathit{pfseq}\ X \bullet y \sqsubseteq x \Rightarrow x \in c\}
\end{array}$$

Infinite lists are now represent as elements from *pchain*.

$$\mathit{pfiseq}\ X \mathrel{\widehat{=}} \mathit{pchain}[X]$$

The information ordering on infinite lists is characterised by subset inclusion.

$$\begin{array}{l}
[X] \\
\_ \sqsubseteq_{\infty} \_ : \mathit{pfiseq}\ X \leftrightarrow \mathit{pfiseq}\ X \\
\hline
\forall\, x, y : \mathit{pfiseq}\ X \bullet x \sqsubseteq_{\infty} y \Leftrightarrow x \subseteq y
\end{array}$$

Definition of the function $\mathit{combine}_{\infty}$ for combination of the elements of lists of infinite lists. This uses map2, which is similar to the mapping function $*$, but takes two lists as arguments and a binary function, which is mapped over all possible combinations of arguments of elements from the two lists.

$$\begin{array}{l}
[X, Y, Z] \\
\mathsf{map2} : (X \times Y \rightarrow Z) \rightarrow \mathit{pfiseq}\ X \rightarrow \mathit{pfiseq}\ Y \rightarrow \mathit{pfiseq}\ Z \\
\hline
\forall\, f : X \times Y \rightarrow Z;\ s_1 : \mathit{pfiseq}\ X;\ s_2 : \mathit{pfiseq}\ Y \bullet \\
\quad f\ \mathsf{map2}\,(s_1, s_2) = \mathbf{let}\ g == (\lambda\, x : X \bullet (\lambda\, y : Y \bullet f(x, y)) * s_2) \bullet \overset{\infty}{\frown}/\,(g * s_1)
\end{array}$$

$$
\begin{array}{l}
[X] \\
\hline
combine_{\infty} : \mathrm{seq}(pfiseq\ X) \to pfiseq(\mathrm{seq}\, X) \\
\hline
combine_{\infty}\ \langle\rangle = \langle\langle\rangle\rangle_{\infty} \\
\forall\, l : pfiseq\ X;\ rest : \mathrm{seq}(pfiseq\ X) \bullet \\
\quad combine_{\infty}\ \langle l\rangle \frown rest = (\lambda\, x : X;\ y : \mathrm{seq}\, X \bullet \langle x\rangle \frown y)\ \mathsf{map2}\ (l, \mathsf{combine}_{\infty}\ rest)
\end{array}
$$

B. Mechanisation of the UTP

In this appendix, we present some of the missing definitions discussed in the presentation of our mechanisation of the UTP framework, and of Morgan's program model as a UTP theory.

Semantic domain for variables The free type *SUBSCRIPT* is used to record a possible subscript.

$$SUBSCRIPT ::= Sub(\mathbb{N}) \mid SubNone$$

Variables are characterised by a Z schema.

$$
\begin{array}{l}
VAR \\
\hline
name : STRING; \\
dashes : \mathbb{N}; \\
subscript : SUBSCRIPT; \\
type : TYPE
\end{array}
$$

The components of the schema specify the name of the variable as a string, the number of dashes as a natural number, a possible subscript, and the type of the variable. (The definition of *TYPE* is $\mathbb{P}_1\ VALUE$.)

Definitions for the predicate model The definition of *BINDING* guarantees that bindings are well-typed. Here it means that variables are only associated with values in their type.

$$BINDING \mathrel{\widehat=} \{b : VAR \rightarrow\!\!\!\!\!\!\!+\ \ VALUE \mid (\forall\, n : \mathrm{dom}\, b \bullet b(n) \in n.type)\}$$

The set *ALPHA_FUNCTION* contains all partial unary functions on *ALPHA_PREDICATE* that preserve compatibility as implied by membership to *WF_ALPHA_PREDICATE_PAIR*.

$$
\begin{array}{l}
ALPHA\_FUNCTION \mathrel{\widehat=} \{f : ALPHA\_PREDICATE \rightarrow\!\!\!\!\!\!\!+\ \ ALPHA\_PREDICATE \mid \\
\quad \forall\, p_1, p_2 : ALPHA\_PREDICATE \mid (p_1, p_2) \in WF\_ALPHA\_PREDICATE\_PAIR \land \\
\qquad \{p_1, p_2\} \subseteq \mathrm{dom}\, f \bullet (f(p_1), f(p_2)) \in WF\_ALPHA\_PREDICATE\_PAIR\}
\end{array}
$$

This definition states that, for f to be a member of the type *ALPHA_FUNCTION*, any two compatible predicates in the domain of f have to be mapped to predicates which are also compatible.

The *Tautology* function.

$$
\begin{array}{l}
Tautology : ALPHA\_PREDICATE \to \mathbb{B} \\
\hline
\forall\, p : ALPHA\_PREDICATE \bullet Tautology\ p \Leftrightarrow p = True_P\ p.1
\end{array}
$$

It determines if a given predicate is universally true: equal to *true* over its alphabet.

Definitions for the theories of relations and designs

Set of predicates that are (relational) conditions.

$$REL\_CONDITION \mathrel{\widehat=} \{p : REL\_PREDICATE \mid out_A\ p.1 \in unrestVars\ p\}$$

A condition is a relational predicate in which the variables of the output alphabet (given by out_A) are unconstrained. The function *unrestVars* generally yields all unconstrained variables in a predicate.

Alphabet for design predicates.

$$DES\_ALPHABET \mathrel{\widehat=} \{a : REL\_ALPHABET \mid ALPHABET\_OKAY \subseteq a\}$$

In the above definition, $ALPHABET\_OKAY$ includes the auxiliary variables ok and ok'.

Instantiation function for design theories.

$$\begin{array}{|l} InstDesTheory : DES\_ALPHABET \to UTP\_THEORY \\ \hline \forall a : DES\_ALPHABET \bullet InstDesTheory\ a = SpecialiseTheory\ (InstRelTheory\ a, \{H1, H2\}) \end{array}$$

A design theory instance is obtained by specialising a relational theory with the healthiness conditions $H1$ and $H2$. The alphabet must be a valid design alphabet, that is an element from $DES\_ALPHABET$.

Set of valid design predicates.

$$DESIGN \mathrel{\widehat=} \{p : ALPHA\_PREDICATE \mid (\exists\, th : DES\_THEORY \bullet p \in TheoryPredicates\ th)\}$$

For p to be a valid design, some design theory instance must exist such that p is a member of its predicates. The design Skip.

$$\begin{array}{|l} \mathbf{II}_D : REL\_ALPHABET\_HOM \to DESIGN \\ \hline \forall a : REL\_ALPHABET\_HOM \bullet \mathbf{II}_D\ a = TRUE_P \vdash_D (\mathbf{II}_R\ a) \end{array}$$

It has a true precondition ($TRUE_P$), and leaves the variables of the alphabet unchanged, which is achieved by the relational Skip $\mathbf{II}_R$ in the postcondition. The operator $\vdash_D$ is our encoding of the UTP design constructor.

The design assignment.

$$\begin{array}{|l} Assign_D : WF\_Assign_D \to DESIGN \\ \hline \forall a\_ns\_es : WF\_Assign_D \bullet Assign_D\ a\_ns\_es = TRUE_P \vdash_D (Assign_R\ a\_ns\_es) \end{array}$$

As with the design Skip, we express the design assignment by virtue of a design with a true precondition, and a relational assignment in the postcondition.

The top of the design lattice.

$$\begin{array}{|l} Magic_D : DES\_ALPHABET \to DESIGN \\ \hline a : DES\_ALPHABET \bullet Magic_D\ a = (\neg_P\ OKAY) \oplus_P a \end{array}$$

The function $Magic_D$ encodes the definition of the top of the design lattice (magic). The $\oplus_P$ encodes the UTP alphabet extension operator, and above it is used to add a to the alphabet of $\neg_P\ OKAY$. The predicate $OKAY$ is an encoding of $ok = true$ or simply ok, in the UTP notation.

Semantic restriction for a valid list of guarded commands.

$$\begin{array}{|l} GUARDED\_CMDS \mathrel{\widehat=} \{gs : \operatorname{seq} MORGAN\_CONDITION;\ cmds : \operatorname{seq} MORGAN\_PROGRAM \mid \\ \quad \#\, gs = \#\, cmds \land (\forall i : \operatorname{dom} gs \bullet (gs(i), cmds(i)) \in GUARDED\_CMD)\} \end{array}$$

Above, $GUARDED\_CMD$ captures the semantic restriction of a guarded command. Thus, each corresponding pair of elements in the gs and $cmds$ sequences has to be a valid guarded command.

Definitions for Morgan's program model

Permissible variables in Morgan programs.

$$\begin{array}{|l} M\_VAR : \mathbb{P}\, VAR \\ \hline M\_VAR \subseteq (undashed \cap plain) \setminus \{ok\} \end{array}$$

Predicates used in conditions of Morgan programs.

$$MORGAN\_CONDITION \mathrel{\widehat=} \{p : ALPHA\_PREDICATE \mid p.1 \subseteq M\_VAR\}$$

The predicates in $MORGAN\_CONDITION$ do not represent computations themselves. They are used, for example, as arguments when constructing specification statements or guards in conditionals and loops.

Predicates used in postconditions of specification statements.

$$MORGAN\_POSTCOND \mathrel{\widehat=} \{p : ALPHA\_PREDICATE \mid p.1 \in M\_VAR \land zero \,(\!|\, M\_VAR \,|\!)\}$$

Here $zero$ is a variable substitution operator that adds a 0 subscript to all variables.

Characterisation of variable substitutions.

$$VAR\_SUBST \mathrel{\widehat=} \{f : VAR \rightarrowtail\!\!\!\!\!\!\!\!\;\;\;\;\;\;\; VAR \mid (\forall n : \operatorname{dom} f \bullet n.type = (f\ n).type)\}$$

Substitution of variable names of a predicate.

$$\begin{array}{|l}
VarSubst_P : (ALPHA\_PREDICATE \times VAR\_SUBST) \to ALPHA\_PREDICATE \\
\hline
\forall p : ALPHA\_PREDICATE;\ f : VAR\_SUBST \bullet \\
\quad VarSubst_P\,(p, f) = (Subst_A\,(p.1, f), \{b : p.2 \bullet Subst_B\,(b, f)\})
\end{array}$$

The above function makes use of two utility functions, $Subst_A$ and $Subst_B$. Namely, $Subst_A$ substitutes the variables in an alphabet, and $Subst_B$ substitutes the variables of a binding.

The substitution used in the operator definition for the specification statement is as follows.

$$\begin{array}{|l}
Subst_M : VAR\_SUBST \\
\hline
\operatorname{dom} Subst_M = undashed \cup \operatorname{ran} zero \land \\
(\forall n : VAR \bullet (n \in undashed \Rightarrow Subst_M\ n = dash\ n) \land (n \in \operatorname{ran} zero \Rightarrow Subst_M\ n = (zero^{\sim})\ n)
\end{array}$$

The inverse of the $zero$ function is written $zero^{\sim}$.

The quintessential construct in Morgan's refinement calculus is the specification statement $w : [pre, post]$. The function below encodes this operator.

$$\begin{array}{|l}
SpecStmt_M : WF\_SpecStmt_M \to MORGAN\_PROGRAM \\
\hline
\forall a : MORGAN\_ALPHABET;\ w : \operatorname{seq} M\_VAR; \\
\quad preC : MORGAN\_CONDITION;\ postC : MORGAN\_POSTCOND \mid \\
\qquad (a, w, preC, postC) \in WF\_SpecStmt_M \bullet \\
\qquad\quad SpecStmt_M\,(a, w, preC, postC) = \\
\qquad\qquad preC \vdash_D (VarSubst_P\,(postC, Subst_M) \land_P \mathbf{II}_R\,(a \setminus (\operatorname{ran} w \cup dash\,(\!|\operatorname{ran} w|\!))
\end{array}$$

We use the semantic characterisation $pre \vdash post[w_0, w \backslash w, w'] \land \mathbf{II}_{A \setminus w}$ of the specification statement as a design. The alphabet A is the complete set of variables occurring in either the pre or postcondition, with possible 0 subscripts removed. The conjunction with $\mathbf{II}_{A \setminus w}$ in the postcondition ensures that variables outside the frame retain their value. For example, $x : [y \geq 0, x = x_0 + 1]$ is encoded as the design $y \geq 0 \vdash x' = x + 1 \land y' = y$ with alphabet $\{x, x', y, y', ok, ok'\}$.

Definition of the function GG used in the encoding of the Morgan conditional and iteration.

$$\begin{array}{|l}
GG : \operatorname{seq} MORGAN\_CONDITION \to MORGAN\_CONDITION \\
\hline
\forall gs : \operatorname{seq} MORGAN\_CONDITION \bullet \\
\quad GG\ gs\ =\ if\ \#\,gs = 0\ then\ FALSE_P\ else\ (head\ gs) \lor_P GG\,(tail\ gs)
\end{array}$$

The **var** $x \bullet p$ construct.

$$
\begin{array}{|l}
var_M : WF\_var_M \rightarrow MORGAN\_PROGRAM \\
\hline
\forall n : M\_VAR;\ p : MORGAN\_PROGRAM \mid (n, p) \in WF\_var_M \bullet \\
\quad var_M\,(n, p) \ = \ var_R\,(p.1, n) \ ;_M \ p \ ;_M \ end_R\,(p.2, n)
\end{array}
$$

Definition of the *iterLawGCmds* function used in the encoding of the iter law.

$$
\begin{array}{|l}
iterLawGCmds : WF\_iterLawGCmds \rightarrow GUARDED\_CMDS \\
\hline
\forall a : MORGAN\_ALPHABET;\ f : \operatorname{seq} M\_VAR;\ inv : MORGAN\_CONDITION; \\
\quad guards : \operatorname{seq} MORGAN\_CONDITION;\ V : WT\_EXPRESSION \mid \\
\qquad (a, f, inv, guards, V) \in WF\_iterLawGCmds \bullet \\
\qquad\quad iterLawGCmds\,(a, f, inv, guards, V) = \\
\qquad\qquad if \ \#\, guards = 0 \\
\qquad\qquad then \ (\langle\rangle, \langle\rangle) \\
\qquad\qquad else \ (\langle head\ guards \frown (iterLawGCmds\,(a, f, inv, tail\ guards, V\rangle)).1, \\
\qquad\qquad\qquad \langle SpecStmt_M\,(a, f, inv \wedge_P (head\ guards), \\
\qquad\qquad\qquad\quad inv \wedge_P Rel_P\,((\_\leq_V\_), Val(Int(0)), V) \wedge_P Rel_P\,((\_<_V\_), V, Subst_E\,(V, zero)))\rangle^\frown \\
\qquad\qquad\qquad (iterLawGCmds(a, f, inv, tail\ guards, V)).2)
\end{array}
$$

Monotonicity theorem.

$$
\begin{array}{l}
\vdash \forall n : M\_VAR;\ p, p' : MORGAN\_PROGRAM \mid \\
\quad (n, p) \in WF\_var_M \wedge (p, p') \in WF\_MORGAN\_PROGRAM\_PAIR \bullet \\
\qquad p \ \sqsubseteq \ p' \Rightarrow var_M\,(n, p) \ \sqsubseteq \ var_M\,(n, p')
\end{array}
$$

Some of the monotonicity theorems have additional provisos that ensure well-definedness of the operator applications and refinement relations. Above, these are first that (n, p) is a member of WF_var_M, the domain of the var_M function; it ensures that the application $var_M\,(n, p)$ is well-defined. Additionally, $(p, p') \in WF\_MORGAN\_PROGRAM\_PAIR$ ensures that $p \sqsubseteq p'$ is a meaningful (HOL) predicate. These assumptions are vital to establish the provability of the theorem.