Under consideration for publication in Formal Aspects of Computing

ArcAngel: a Tactic Language For Refinement

Marcel Oliveira[1], Ana Cavalcanti[1], and Jim Woodcock[2]

[1]Centro de Informática, Universidade Federal de Pernambuco, Brazil
[2]Computing Laboratory, University of Kent, UK

Abstract. Morgan's refinement calculus is a successful technique for developing software in a precise and consistent way. This technique, however, can be hard to use, as developments may be long and repetitive. Many authors have pointed out that a lot can be gained by identifying commonly used development strategies, documenting them as tactics, and using them as single transformation rules. Also, it is useful to have a notation for describing derivations, so that they can be analysed and modified. In this paper, we present ArcAngel, a language for defining such refinement tactics; we present the language, its semantics, and some of its algebraic laws. Apart from Angel, a general-purpose tactic language that we are extending, no other tactic language has a denotational semantics and proof theory of its own.

Keywords: Formal methods, program development, refinement calculus.

1. Introduction

The refinement calculus of Morgan [Mor94] is a modern technique for formal program development. By repeatedly applying transformation rules to an initial specification, we produce a program that implements it correctly; however, this may be a hard task, since developments are often long and repetitive. Some development strategies may be captured as sequences of rule applications, and used in different developments, or even several times within a single development. Identifying these strategies, documenting them as refinement tactics, and using them as single transformation rules brings a profit in time and effort. Also, a notation for describing derivations can be used for modifying and analysing formal derivations.

In this paper we present a refinement-tactic language called ArcAngel, derived from the more general tactic language, Angel [MGW96, Mar96]. In [OC00], we give an informal description of ArcAngel and use it to formalise commonly used refinement strategies; here we provide a formal

Correspondence and offprint requests to: Marcel Oliveira, Centro de Informática, Universidade Federal de Pernambuco, Brazil. e-mail: mvmo@cin.ufpe.br

semantics. Based on these definitions, we have proved over seventy laws of reasoning, which support a strategy of reduction to a normal form for a subset of ArcAngel. We discuss this work here; all the laws, their proofs, and a detailed account of the reduction strategy may be found in [Oli00].

Angel is a general-purpose tactic language that is not tailored to any particular proof tool; it assumes only that rules transform proof goals. A refinement-tactic language must take into account the fact that, when applying refinement laws to a program, we get not only a program, but proof obligations as well. So, the result of applying a tactic is a program and a list of all the proof obligations generated by the individual law applications. ArcAngel's constructs are similar to Angel's, but are adapted to deal with the application of refinement laws to programs. In particular, ArcAngel's structural combinators are used to apply tactics to program components. Angel is distinctive because it is the only tactic language to have a denotational semantics and an accompanying proof theory. In this paper, the semantics and laws are specialised to ArcAngel.

In Section 2, we give an overview of the refinement calculus. In Section 3, we introduce ArcAngel, and in Section 4, we give its formal semantics and present some of its laws. In Section 5, we give some examples of tactics and their application. Finally, in Sections 6 and 7 we discuss related and future work. Appendix A presents the model used in the semantics of ArcAngel for infinite lists, and Appendix B presents the refinement laws used in this paper.

2. Refinement Calculus

Morgan's refinement calculus [Mor94] is based upon a unified language for specification, design, and implementation; its syntax is presented in Figure 1. The syntactic category *name* is the set of valid names; we use $name^*$ to denote a possibly empty list of (comma-separated) names, and similarly for $expression^*$. The syntactic categories *predicate* and *expression* are, as expected, those of first-order predicates and expressions over data types like integers, sets, and others; their definitions are standard and are omitted.

A program may be a specification statement, and program development is the application of a sequence of refinement laws to transform a specification into program code that correctly implements it. Specification statements take the form $w : [\, pre, post \,]$, where the frame w lists the variables that may be changed, *pre* is the precondition, and *post* is the postcondition. When a state satisfies the precondition, the execution of the specification statement changes the variables listed in the frame so that the final state satisfies the postcondition. References in the postcondition to the initial value of a variable x are written x_0; 0-subscripted variables like this are called initial variables. A precondition *true* may be omitted.

The language of the refinement calculus includes all the constructs of Dijkstra's language of guarded commands [Dij76]. Local variables, logical constants, and procedures may be declared in block constructs that restrict the scope of definitions. For procedures, we follow the approach in [Bac87, CSW97, CSW98], which is based on parameterised commands. Arguments may be passed by value, by result, or by value-result. A recursive procedure is declared using a variant block, which declares the procedure, a variant, and the main program. The variant may be used to develop a recursive implementation for the procedure's specification.

A detailed presentation of the refinement calculus can be found in [Mor94]; we introduce refinement laws as we need them in examples. Appendix B lists the refinement laws we use here. Related work on refinement calculi can be found in [Bac78, Mor87, BvW98].

3. ArcAngel

The syntax of ArcAngel is displayed in Figure 2. The syntactic category *args* is the set of possibly empty comma-separated lists of arguments enclosed in parentheses; an argument is a predicate or an expression; and *pars* is the set of possibly empty comma-separated lists of (parameter) names enclosed in parentheses. The definitions of these syntactic categories are standard and are omitted. Finally, the notation $tactic^+$ is used to represent a non-empty list of tactics; in examples we number this list. A pair of square brackets represents optional clauses.

$$
\begin{array}{rcll}
\mathit{program} & ::= & \mathit{name}^* : [\mathit{predicate}, \mathit{predicate}] & [\textbf{specification statement}] \\
& | & \mathit{name}^* := \mathit{expression}^* & [\textbf{multiple} - \textbf{assignment}] \\
& | & \mathit{program};\ \mathit{program} & [\textbf{sequence}] \\
& | & \textbf{if}\ [\!]\ i \bullet \mathit{predicate} \rightarrow \mathit{program}\ \textbf{fi} & [\textbf{conditional}] \\
& | & \textbf{do}\ [\!]\ i \bullet \mathit{predicate} \rightarrow \mathit{program}\ \textbf{od} & [\textbf{loop}] \\
& | & [\![\ \textbf{var}\ \mathit{varDec} \bullet \mathit{program}\ ]\!] & [\textbf{variable block}] \\
& | & [\![\ \textbf{con}\ \mathit{varDec} \bullet \mathit{program}\ ]\!] & [\textbf{constant block}] \\
& | & [\![\ \textbf{proc}\ \mathit{name} \mathrel{\widehat{=}} \mathit{procBody} \bullet \mathit{program}\ ]\!] & [\textbf{procedure block}] \\
& | & [\![\ \textbf{proc}\ \mathit{name} \mathrel{\widehat{=}} \mathit{procBody}\ \textbf{variant}\ \mathit{name}\ \textbf{is}\ \mathit{expression} \bullet \mathit{program}\ ]\!] & [\textbf{variant block}] \\
& | & \mathit{name} & [\textbf{procedure call}] \\
& | & \mathit{name}(\mathit{expression}^+) \ \ |\ \ (\mathit{parDec} \bullet \mathit{program})(\mathit{expression}^+) & \\
\mathit{procBody} & ::= & \mathit{program} \ \ |\ \ (\mathit{parDec} \bullet \mathit{program}) & \\
\mathit{parDec} & ::= & \textbf{val}\ \mathit{varDec} \ \ |\ \ \textbf{res}\ \mathit{varDec} \ \ |\ \ \textbf{val-res}\ \mathit{varDec} \ \ |\ \ \mathit{parDec};\ \mathit{parDec}\ \ 1\mathrm{g} & \\
\mathit{varDec} & ::= & \mathit{name}^+ : \mathit{Type} \ \ |\ \ \mathit{varDec};\ \mathit{varDec} &
\end{array}
$$

Fig. 1. Abstract Syntax of the language of Morgan's Refinement Calculus

There are three distinct kinds of tactics: basic tactics are the simplest tactics; tacticals are combination of tactics; and structural combinators are tactics used to handle parts of a program. Some of the basic tactics and most of the tacticals are original to ArcAngel; most of the structural combinators in ArcAngel do not exist in Angel.

The most basic tactic is a simple law application: **law** $n(a)$. The application of this tactic to a program has two possible outcomes. If the law n with arguments a is applicable to the program, then the application actually occurs and the program is changed, possibly generating proof obligations. If the law is not applicable to the program, then the application of the tactic fails. The construct **tactic** $n(a)$ applies a previously defined tactic as though it were a single law. The trivial tactic **skip** always succeeds, and the tactic **fail** always fails; neither generates any proof obligations. The tactic **abort** neither succeeds nor fails, but runs indefinitely.

3.1. Tacticals

In ArcAngel, tactics may be sequentially composed: $t_1;\ t_2$. This tactic first applies t_1 to the program, and then applies t_2 to the outcome of the application of t_1. If either t_1 or t_2 fails, then so does the whole tactic. When it succeeds, the proof obligations generated by the application of this tactic are those resulting from the application of t_1 and t_2.

For example, consider the program $x : [\, x \geq 1\,]$. We could implement this by first strengthening the postcondition to $x = 1$, and then replacing it with a simple assignment. The necessary tactic is **law** $\mathit{strPost}(x = 1)$; **law** $\mathit{assign}(x := 1)$. After the application of the first tactic, the resulting program is $x : [\, x = 1\,]$ and the proof obligation is $x = 1 \Rightarrow x \geq 1$. After the application of the assignment introduction law, we get the program $x := 1$ with the two proof obligations $x = 1 \Rightarrow x \geq 1$ and $\mathit{true} \Rightarrow 1 = 1$.

As already explained, arguments are expressions or predicates. In examples, however, for clarity, sometimes we use programs and declarations as arguments as a syntactic sugar. For instance,

$$
\begin{array}{lcll}
\mathit{tactic} & ::= & \textbf{law}\ \mathit{name}\ \mathit{args} \mid \textbf{tactic}\ \mathit{name}\ \mathit{args} & [\textbf{law and tactic application}] \\
& \mid & \textbf{skip} \mid \textbf{fail} \mid \textbf{abort} & \\
& \mid & \mathit{tactic};\ \mathit{tactic} & [\textbf{sequence}] \\
& \mid & \mathit{tactic} \mid \mathit{tactic} & [\textbf{alternation}] \\
& \mid & !\,\mathit{tactic} & [\textbf{cut}] \\
& \mid & \mu\,\mathit{name} \bullet \mathit{tactic} & [\textbf{recursion}] \\
& \mid & \textbf{succs}\ \mathit{tactic} \mid \textbf{fails}\ \mathit{tactic} & [\textbf{assertions}] \\
& \mid & \mathit{tactic}\ \boxed{;}\ \mathit{tactic} & [\textbf{structural combinators}] \\
& \mid & \boxed{\textbf{if}}\ \mathit{tactic}^+\ \boxed{\textbf{fi}} \mid \boxed{\textbf{do}}\ \mathit{tactic}^+\ \boxed{\textbf{od}} & \\
& \mid & \boxed{\textbf{var}}\ \mathit{tactic}\ \boxed{[]} \mid \boxed{\textbf{con}}\ \mathit{tactic}\ \boxed{[]} & \\
& \mid & \boxed{\textbf{pmain}}\ \mathit{tactic}\ \boxed{[]} \mid \boxed{\textbf{pmainvariant}}\ \mathit{tactic}\ \boxed{[]} & \\
& \mid & \boxed{\textbf{pbody}}\ \mathit{tactic}\ \boxed{[]} \mid \boxed{\textbf{pbodyvariant}}\ \mathit{tactic}\ \boxed{[]} & \\
& \mid & \boxed{\textbf{pbodymain}}\ \mathit{tacticBody}\ \mathit{tacticMain}\ \boxed{[]} & \\
& \mid & \boxed{\textbf{pmainvariantbody}}\ \mathit{tacticBody}\ \mathit{tacticMain}\ \boxed{[]} & \\
& \mid & \boxed{\textbf{val}}\ \mathit{tactic} \mid \boxed{\textbf{res}}\ \mathit{tactic} \mid \boxed{\textbf{val-res}}\ \mathit{tactic} & \\
& \mid & \textbf{con}\ v \bullet \mathit{tactic} & [\textbf{constants}] \\
& \mid & \textbf{applies to}\ \ \mathit{program}\ \textbf{do}\ \mathit{tactic} & [\textbf{patterns}] \\
\\
\mathit{tacDec} & ::= & \textbf{Tactic}\ \mathit{name}\ \mathit{pars}\ \mathit{tactic} & \\
& & [\,\textbf{proof obligations}\ \mathit{predicate}^+\,] & \\
& & [\,\textbf{program generated}\ \mathit{program}\,]\ \textbf{end} & \\
\\
\mathit{tacProg} & ::= & \mathit{tacDec}^*\,\mathit{tactic} &
\end{array}
$$

Fig. 2. Abstract Syntax of ArcAngel

above, we write **law** *assign*$(x := 1)$, but the arguments are actually x and 1. We use the same sort of notation for parameter names, and for arguments and parameters of tactics.

Tactics may also be combined in alternation: $t_1 \mid t_2$. First t_1 is applied to the program. If the application of t_1 leads to success, then the composite tactic succeeds; otherwise t_2 is applied to the program. If the application of t_2 leads to success then the composite tactic succeeds; otherwise the composite tactic fails. If one of the tactics aborts, the whole tactic aborts. When a tactic contains many choices, the first choice that leads to success is selected. It is the angelic nature of this nondeterminism, in which alternative tactics that lead to success are chosen over those that lead to failure, that earned Angel and ArcAngel (A Refinement Calculus for Angel) their names.

For example, suppose that we have a useful tactic t that relies on the frame containing only the variable x, and that we want to generalise it to t', which applies to specification statements with frame x, y. We could define t' to be **law** *contractFrame*(y); t . Unfortunately, the resulting tactic no longer applies where we found t to be useful, since contracting the frame will surely fail in such cases. Instead, we can use the compound tactic (**law** *contractFrame*(y) | **skip**); t. Now, if contracting the frame works, we do it; if it does not, then we ignore it; either way, we apply t next. The tactic (**skip** | **law** *contractFrame*(y)); t has the same effect. In this case, the tactic t is applied without any prior change to the specification statement. If this application does not succeed, the law *contractFrame* is applied before a new attempt to apply t.

The angelic nondeterminism can be implemented through backtracking: in the case of failure, law applications are undone to go back to the last point where further alternatives are available and can be explored. This, however, may result in inefficient searches. Some control over this is given to

the programmer through the cut operator. The cut tactic $!\,t$ behaves like t, except that it returns the first successful application of t. If a subsequent tactic application fails, then the whole tactic fails. Consider our previous example once more, supposing that **law** *contractFrame*(y) succeeds, but that t subsequently does not. There is no point is applying the trivial tactic **skip** and then trying t again. Instead, we should cut [CM81] the search: $!(\,\mathbf{law}\ \mathit{contractFrame}(y) \mid \mathbf{skip}\,);\ t$.

Suppose that we have a tactic u that performs some simple task, like identifying an equation in a specification postcondition and then applying the rule of *following-assignment*. Such a simple tactic may be made more useful by applying it repeatedly, until it can be applied no more. ArcAngel has a fixed-point operator that allows us to define recursive tactics. Using this operator, we can define a tactic that applies u exhaustively: the tactic $\mu\, X \bullet (\,u;\ X \mid \mathbf{skip}\,)$ applies u as many times as possible, terminating with success when the application of u fails. Recursive application of a tactic may lead to nontermination, in which case the result is the same as the trivial tactic **abort**.

The tactic $\mathbf{con}\, v \bullet t$ introduces v as a set of free variables ranging over appropriate syntactic classes in t, angelically chosen so that as many choices as possible succeed. An example of its use is presented in Section 4.

The tactic $\mathbf{applies\,to}\ p\ \mathbf{do}\ t$ is used to define a pattern for the programs to which the tactic t can be applied. It introduces a meta-program p that characterises the programs to which this tactic is applicable; the meta-variables used in p can then be used in t. For example, the meta-program $w : [\,\mathit{pre}, \mathit{post}_1 \vee \mathit{post}_2\,]$ characterises those specifications whose postcondition is a disjunction; here, pre, post_1, and post_2 are the meta-variables. Consider as an example a commonly used refinement tactic: strengthening a postcondition by dropping a disjunct. This tactic is formalised as $\mathbf{applies\,to}\ w : [\,\mathit{pre}, \mathit{post}_1 \vee \mathit{post}_2\,]\ \mathbf{do}\ \mathbf{law}\ \mathit{strPost}(\mathit{post}_1)$.

Two tactics are used to make tactic assertions that check the outcome of applying a tactic. The tactic $\mathbf{succs}\, t$ fails whenever t fails, and behaves like **skip** whenever t succeeds. On the other hand, $\mathbf{fails}\, t$ behaves like **skip** if t fails, and fails if t succeeds. If the application of t runs indefinitely, then these tacticals behave like **abort**. A simple example is a test to see whether a program is a specification statement. We know that it is always possible (but seldom desirable) to strengthen a specification statement's postcondition to *false*; however, the tactic applies only to specification statements. So our test may be coded as $\mathit{succs}(\,\mathbf{law}\ \mathit{strPost}(\mathit{false})\,)$.

3.2. Structural Combinators

Very often, we want to apply individual tactics to subprograms. The tactic $t_1\,\boxed{;}\,t_2$ applies to programs of the form $p_1;\ p_2$. It returns the sequential composition of the programs obtained by applying t_1 to p_1 and t_2 to p_2; the proof obligations generated are those arising from both tactic applications. Combinators like $\boxed{;}$ are called *structural combinators*. These combinators correspond to the syntactic structures in the programming language. Essentially, there is one combinator for each syntactic construct.

For alternation, there is the structural combinator $\boxed{\mathbf{if}}\,t_1\,\boxed{[]}\,\ldots\,\boxed{[]}\,t_n\,\boxed{\mathbf{fi}}$, which applies to an alternation $\mathbf{if}\ g_1 \to p_1\ [\!]\ \ldots\ [\!]\ g_n \to p_n\ \mathbf{fi}$. It returns the result of applying each tactic t_i to the corresponding program p_i. For example, if we have the program

$$\mathbf{if}\ a < b \to x : [\,x < 0\,] [\!] a = b \to x : [\,x = 0\,] [\!] a > b \to x : [\,x > 0\,]\ \mathbf{fi}$$

and tactic

$$\boxed{\mathbf{if}}\,\mathbf{law}\ \mathit{assign}(x := -1)\ \boxed{[]}\,\mathbf{law}\ \mathit{assign}(x := 0)\ \boxed{[]}\,\mathbf{law}\ \mathit{assign}(x := 1)\ \boxed{\mathbf{fi}}$$

we obtain three proof obligations $\mathit{true} \Rightarrow -x < 0$ and $\mathit{true} \Rightarrow 0 = 0$, and $\mathit{true} \Rightarrow 1 > 0$, and the resulting program is

$$\mathbf{if}\ a < b \to x := -1\ [\!]\ a = b \to x := 0\ [\!]\ a > b \to x := 1\ \mathbf{fi}$$

For iterations $\mathbf{do}\ g_1 \to p_1\ [\!]\ \ldots\ [\!]\ g_n \to p_n\ \mathbf{od}$, we have a similar structural combinator $\boxed{\mathbf{do}}\,t_1\,\boxed{[]}\,\ldots\,\boxed{[]}\,t_n\,\boxed{\mathbf{od}}$.

The structural combinator $\boxed{\textbf{var}}\, t\, \boxed{[\![}$ applies to a variable block, and $\boxed{\textbf{con}}\, t\, \boxed{[\![}$ to a logical constant block; each applies its tactic t to the body of the block. For example, if we apply to

$$[\![\, \textbf{var}\ x : \mathbb{N} \bullet x : [\, x \geq 0\,]\,]\!]$$

the structural combinator

$$\boxed{\textbf{var}}\ \textbf{law}\ \mathit{strPost}(x > 0)\, \boxed{[\![}$$

we get the new variable block

$$[\![\, \textbf{var}\ x : \mathbb{N} \bullet x : [\, x > 0\,]\,]\!]$$

and the proof obligation $x > 0 \Rightarrow x \geq 0$.

In the case of procedure blocks and variant blocks, the structural combinators $\boxed{\textbf{pmain}}\, t\, \boxed{[\![}$ and $\boxed{\textbf{pmainvariant}}\, t\, \boxed{[\![}$ are used, respectively; they apply t to the main program of the blocks. For example, applying the tactic

$$\boxed{\textbf{pmain}}\ \textbf{law}\ \mathit{strPost}(x > 0)\, \boxed{[\![}$$

to the procedure block

$$[\![\, \textbf{proc}\ \mathit{nonNeg} \mathrel{\widehat{=}} x : [\, x > 0\,] \bullet x : [\, x \geq 0\,]\,]\!]$$

we get the program

$$[\![\, \textbf{proc}\ \mathit{nonNeg} \mathrel{\widehat{=}} x : [\, x > 0\,] \bullet x : [\, x > 0\,]\,]\!]$$

and the proof obligation $x > 0 \Rightarrow x \geq 0$.

To apply a tactic to a procedure body, we use the structural combinators $\boxed{\textbf{pbody}}\, t\, \boxed{[\![}$ and $\boxed{\textbf{pbodyvariant}}\, t\, \boxed{[\![}$, which apply to procedure and variant blocks, respectively. For argument declaration, the combinators $\boxed{\textbf{val}}\, t$, $\boxed{\textbf{res}}\, t$, and $\boxed{\textbf{val-res}}\, t$ are used, depending on whether the arguments are passed by value, result, or value-result. For example, when the following tactic

$$\boxed{\textbf{pbody}}\,\boxed{\textbf{val-res}}\ \textbf{law}\ \mathit{strPost}(x > 0)\, \boxed{[\![}$$

is applied to the procedure block

$$[\![\, \textbf{proc}\ \mathit{nonNegArg} \mathrel{\widehat{=}} (\textbf{val-res}\ x : \mathbb{N} \bullet x : [\, x \geq 0\,]) \bullet x : [\, x > 0\,]\,]\!]$$

it returns the proof obligation $x > 0 \Rightarrow x \geq 0$ and the program

$$[\![\, \textbf{proc}\ \mathit{nonNegArg} \mathrel{\widehat{=}} (\textbf{val-res}\ x : \mathbb{N} \bullet x : [\, x > 0\,]) \bullet x : [\, x > 0\,]\,]\!]$$

It is also possible to apply tactics to a procedure body and to the main program of a procedure block, or a variant block, at the same time. For this. we use the structural combinators $\boxed{\textbf{pbodymain}}\, t_b\ t_m\, \boxed{[\![}$ and $\boxed{\textbf{pmainvariantbody}}\, t_b\ t_m\, \boxed{[\![}$, which apply to procedure blocks and variant blocks, respectively. They apply t_b to the body of the procedure, and t_m to the main program.

We may declare a named tactic with arguments using the construct $\textbf{Tactic}\, n(a)\, t\ \textbf{end}$. For documentation purposes, we may include clauses **proof obligations** and **program generated**; the former lists the proof obligations generated by the application of t, and the latter shows the program generated. These two clauses are optional as this information can be inferred from the tactic itself. The effect of $\textbf{Tactic}\, n(a)\, t\ \textbf{end}$ is that of t which is named n and uses the arguments a; the presence of the optional clauses does not affect the behaviour.

A tactic program consists of a sequence of tactic declarations followed by a tactic that usually makes use of the declared tactics. Several examples of the use of ArcAngel can be found in Section 5 and in [OC00].

4. Semantics of ArcAngel

Tactics are applied to a pair: the first element of this pair is a program to which the tactic is applied; the second element is the set of proof obligations generated to obtain this program. This pair is called *RCell* (*refinement cell*), and is defined as follows:

$$RCell == program \times \mathbb{P}\, predicate$$

The result of a tactic application is a possibly infinite list of *RCell*s that contains all possible outcomes of its application: every program it can generate, together with the corresponding proof obligations (existing obligations and those generated by the tactic application). Different possibilities arise from the use of alternation, and the list can be infinite, since the application of a tactic may run indefinitely. If the application of some tactic fails, then the empty list is returned.

$$Tactic == RCell \pfun \mathrm{pfiseq}\, RCell$$

The type pfiseq *RCell* is that of possibly infinite lists of *RCell*s. We use the model for infinite lists proposed in [Mar95]; this is summarised in the Appendix A. In this model, finite, partial, and infinite lists are considered. A partial list ends in an undefined list, denoted $\bot$. An infinite list is a limit of a directed set of partial lists.

In order to give semantics to named laws and tactics, we need to maintain two appropriate environments.

$$LEnv == name \pfun \mathrm{seq}\, argument \pfun program \pfun RCell$$

$$TEnv == name \pfun \mathrm{seq}\, argument \pfun Tactic$$

A law environment records a set of known laws; it is a partial function whose domain is the set of known laws. For a law environment Γ_L and a given law name n, $\Gamma_L\, n$ is also a partial function: it relates all valid arguments of n to yet another function. For a valid argument a, $\Gamma_L\, n\, a$ relates all the programs to which n can be applied when given arguments a; the result is a refinement cell.

For example, let Γ_L be an environment that records the law *strPost*. For a predicate $post'$ and a program $w : [\, pre, post\,]$ we have

$$\Gamma_L\, strPost\, post'\, (\, w : [\, pre, post\,]\,) = (\, w : [\, pre, post'\,], \{\, post' \Rightarrow post\,\}\,)$$

This means that if we apply *strPost* with argument $post'$ to the program $w : [\, pre, post\,]$, we change its postcondition to $post'$, but we must prove that $post' \Rightarrow post$. We are interested in environments that record at least the laws of Morgan's refinement calculus in [Mor94].

Similarly, a tactic environment is a function that takes a tactic name and a list of arguments, and returns a *tactic*.

4.1. Tactics

We define the semantic function for tactics inductively; it has the type

$$[\![\_]\!] : tactic \to LEnv \to TEnv \to Tactic$$

The basic tactic **law** $n(a)$ is that which applies a simple law to an *RCell*.

$$[\![\ \mathbf{law}\ n(a)\ ]\!]\, \Gamma_L\, \Gamma_T\, (p, pobs) =$$

$$\begin{array}{l}\quad \mathbf{if}\ n \in \mathrm{dom}\, \Gamma_L \land a \in \mathrm{dom}(\Gamma_L\, n) \land p \in \mathrm{dom}(\Gamma_L\, n\, a) \\ \quad \mathbf{then} \\ \qquad \mathbf{let}\ (newp, npobs) = \Gamma_L\, n\, a\, p\ \mathbf{in}\ [(newp, pobs \cup npobs)] \\ \quad \mathbf{else}\ [\,]\end{array}$$

We check to see if the law name n is in the law environment Γ_L, and if the arguments a and program p are appropriate. If these conditions hold, then the tactic succeeds, and returns a list with a new *RCell*. The program is transformed by applying the law to the program p; the new

proof obligations are added to the proof obligations *pobs* of the original *RCell*. Otherwise, the tactic fails with the empty list as result. We use angle brackets to delimit finite lists; possibly infinite lists are delimited by square brackets.

The semantics of **tactic** $n(a)$ is similar to that of the **law** construct. Its definition is

$$[\![\mathbf{tactic}\, n(a)]\!]\, \Gamma_L\, \Gamma_T\, r =$$
$$\mathbf{if}\ n \in \operatorname{dom} \Gamma_T \wedge a \in \operatorname{dom} \Gamma_T\, n$$
$$\mathbf{then}\ \Gamma_T\, n\, a\, r$$
$$\mathbf{else}\ [\,]$$

If the tactic is in the tactic environment, and if the arguments are valid, then the tactic succeeds; it returns the result of applying the tactic to the arguments. Otherwise, the tactic fails.

The tactic **skip** returns its argument unchanged; the tactic **fail** always fails.

$$[\![\mathbf{skip}]\!]\, \Gamma_L\, \Gamma_T\, r = [r]$$

$$[\![\mathbf{fail}]\!]\ \ \Gamma_L\, \Gamma_T\, r = [\,]$$

The sequence operator uses a construction known as the Kleisli composition [Lan91]. It applies its first tactic to its argument, producing a list of cells; it then applies the second tactic to each member of this list; finally, this list-of-lists is flattened to produce the result.

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

For a total function $f : A \to B$, $f* : \operatorname{pfiseq} A \to \operatorname{pfiseq} B$ is the map function that operates on a list by applying f to each of its elements; the operator $\cdot$ is used to compose functions; and ${}^{\infty}\!/$ is the distributed concatenation operation. Formal definitions of these operators and others to follow can be found in Appendix A.

The semantics of the alternation operator is given by concatenation: the possible outcomes of each individual tactic are joined to give the list of possible outcomes of the alternation.

$$[\![t_1 \mid t_2]\!]\, \Gamma_L\, \Gamma_T = {}^{\infty}\!/ \cdot [([\![t_1]\!]\, \Gamma_L\, \Gamma_T), ([\![t_2]\!]\, \Gamma_L\, \Gamma_T)]^{\circ}$$

The function $^{\circ}$ applies a list of functions to a single argument and returns a list containing the results of the applications.

The cut operator applies its tactic to its argument, taking the first result (if it exists) and discarding the rest; if there is no first result, then the cut tactic fails.

$$[\![!\, t]\!]\, \Gamma_L\, \Gamma_T = \mathit{head}' \cdot ([\![t]\!]\, \Gamma_L\, \Gamma_T)$$

The function head' is just like the usual *head* operator on lists, except that it is total: $\mathit{head}'\, [\,] = [\,]$.

The recursion operator μ has a standard definition [DP90] as a least fixed point. For a continuous function f from tactics to tactics, we have that

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

where f^i represents i applications of f.

This definition makes sense if the set of tactics is a complete lattice. First, we define a partial order for lists: the completely undefined list $\bot$ is the bottom element. One list is less than another, $s_1 \sqsubseteq_\infty s_2$, whenever they are equal, or the first is a partial list that forms an initial subsequence of the second.

For tactics, as they are partial functions, we may lift the ordering on lists of *RCell*s to an ordering on tactics. We can say that $t_1 \sqsubseteq_T t_2 \Leftrightarrow (\forall\, r : \mathit{RCell} \bullet t_1\, r \sqsubseteq_\infty t_2\, r)$. As the set of lists of *RCell*s is a complete lattice, the set of tactics is also a complete lattice, using the order above [DP90].

The bottom element is used in the semantics of **abort**, which runs indefinitely.

$$[\![\mathbf{abort}]\!]\, \Gamma_L\, \Gamma_T = \bot$$

The tactics **succs** t and **fails** t are defined as follows. While abortion arises from a tactic that does

not terminate, failure arises from the application of a law or tactic to a program that is not in its domain, or with inappropriate arguments. In this case, the result is the empty list, as no program or proof-obligation is generated.

$$\mathbf{succs}\, t\, r = (\mathbf{if}\ t\, r = \bot\ \mathbf{then\ abort\ else}\ (\mathbf{if}\ t\, r = [\,]\ \mathbf{then\ fail\ else\ skip})\, r)$$
$$\mathbf{fails}\, t\, r = (\mathbf{if}\ t\, r = \bot\ \mathbf{then\ abort\ else}\ (\mathbf{if}\ t\, r = [\,]\ \mathbf{then\ skip\ else\ fail})\, r)$$

If the application of t aborts, so do $\mathbf{succs}\, t$ and $\mathbf{fails}\, t$. If it fails, then $\mathbf{succs}\, t$ fails and $\mathbf{fails}\, t$ skips. Finally, if it succeeds, $\mathbf{succs}\, t$ skips and $\mathbf{fails}\, t$ fails.

4.2. Structural Combinators

The structural combinators apply tactics to components of a program independently (and so can be thought of as in parallel), and then reassemble the results in all possible ways. There is one combinator for each construct in the programming language.

The structural combinator $t_1 \boxed{;} t_2$ applies to a sequential composition $p_1;\ p_2$. Independently, t_1 is applied to p_1 and t_2 is applied to p_2; the resulting alternatives are assembled into pairs, so that the first is an alternative outcome from t_1 and the second is an alternative outcome from t_2; finally, these pairs are combined with the sequential composition tactical.

$$[\![\, t_1 \boxed{;} t_2 \,]\!]\, \Gamma_L\, \Gamma_T\, (p_1;\ p_2, pobs) = \Omega_{;} * \ (\prod(\langle\, ([\![\, t_1 \,]\!]\, \Gamma_L\, \Gamma_T)\, (p_1, pobs), ([\![\, t_2 \,]\!]\, \Gamma_L\, \Gamma_T)\, (p_2, pobs)\,\rangle))$$

The distributed cartesian product operator $\prod$ is defined in Appendix A. The sequential combination function $\Omega_{;}$ sequentially composes the programs and unites the proof obligations.

$$\Omega_{;} : \mathit{RCell} * \nrightarrow \mathit{RCell}$$
$$\Omega_{;}\, \langle (p_1, pobs_1), (p_2, pobs_2) \rangle = (p_1;\ p_2, pobs_1 \cup pobs_2)$$

The symbol $\nrightarrow$ indicates that $\Omega_{;}$ is a partial function since it is defined only for lists with two elements.

The semantics of the structural combinators $\boxed{\mathbf{if}}\_\boxed{\mathbf{fi}}$ and $\boxed{\mathbf{do}}\_\boxed{\mathbf{od}}$ use a few functions which are explained and defined in [Oli00]; we start by giving a simple example. Consider the *RCell*

$$(\ \mathbf{if}\ x \geq 0 \rightarrow x : [\, x \geq 0, x > y \,]\ [\!]\ x < 0 \rightarrow x : [\, x < 0, x < z \,]\ \mathbf{fi}, \{x \geq 1\}\ )$$

and the tactic

$$\boxed{\mathbf{if}}\ \mathbf{law}\ \mathit{weakPre}(\mathit{true})\ |\ \mathbf{law}\ \mathit{assign}(\ x := x + 1\ )\ \boxed{[\!]}\ \mathbf{law}\ \mathit{assign}(\ x := x - 1\ )\ \boxed{\mathbf{fi}}$$

First of all, we extract the commands from each branch of the conditional; the function *extractP* does this for us.

$$\mathit{extractP}(\ x \geq 0 \rightarrow x : [\, x \geq 0, x > y \,]\ [\!]\ x < 0 \rightarrow x : [\, x < 0, x < z \,]\ )$$
$$= \langle\ x : [\, x \geq 0, x > y \,], x : [\, x < 0, x < z \,]\ \rangle$$

The function Φ_r constructs an *RCell* with a given program p and an empty set of proof obligations. We map Φ_r over our list of commands.

$$\Phi_r * (\ \langle x : [\, x \geq 0, x > y \,], x : [\, x < 0, x < z \,] \rangle\ ) =$$
$$\langle (\ x : [\, x \geq 0, x > y \,], \emptyset\ ), (\ x : [\, x < 0, x < z \,], \emptyset\ ) \rangle$$

Now, we need to apply each tactic to its corresponding command.

We use the function *apply* that takes two lists: the first is a list of functions, and the second is a list of elements to which the functions are applied; it returns list of the results applying each

function in the first list to the corresponding element in the second list. In our example we have

$$\begin{array}{l} \mathit{apply}\,\langle\, \mathbf{law}\ \mathit{weakPre}(\mathit{true}) \mid \mathbf{law}\ \mathit{assign}(\, x := y + 1\,), \mathbf{law}\ \mathit{assign}(\, x := z - 1\,)\,\rangle \\ \qquad \langle\, (\, x : [\, x \geq 0, x > y\,], \emptyset\,), (\, x : [\, x < 0, x < z\,], \emptyset\,)\,\rangle \\ = \langle\, \mathbf{law}\ \mathit{weakPre}(\mathit{true}) \mid \mathbf{law}\ \mathit{assign}(\, x := y + 1\,)\ (\, x : [\, x \geq 0, x > y\,], \emptyset\,), \\ \quad \mathbf{law}\ \mathit{assign}(\, x := z - 1\,)\,(\, x : [\, x < 0, x < z\,], \emptyset\,)\,\rangle \\ = \langle\, [\,(\, x : [\, x > y\,], \{x \geq 0 \Rightarrow \mathit{true}\}\,), (\, x := y + 1, \{x \geq 0 \Rightarrow y + 1 > y\}\,)\,], \\ \quad [\,(\, x := z - 1, \{x < 0 \Rightarrow z - 1 < z\}\,)\,]\,\rangle \end{array}$$

Now, we have to take the distributed cartesian product of this list to get all the possible combinations of the cells of the first list with the cells of the second list.

$$\begin{array}{l} \prod\langle\, [\,(\, x : [\, x > y\,], \{x \geq 0 \Rightarrow \mathit{true}\}\,), (\, x := y + 1, \{x \geq 0 \Rightarrow y + 1 > y\}\,)\,], \\ \quad [\,(\, x := z - 1, \{x < 0 \Rightarrow z - 1 < z\}\,)\,]\,\rangle \\ \\ = [\,\langle\,(\, x : [\, x > y\,], \{x \geq 0 \Rightarrow \mathit{true}\}\,), (\, x := z - 1, \{x < 0 \Rightarrow z - 1 < z\}\,)\,\rangle, \\ \quad \langle\,(\, x := y + 1, \{x \geq 0 \Rightarrow y + 1 > y\}\,), (\, x := z - 1, \{x < 0 \Rightarrow z - 1 < z\}\,)\,\rangle\,] \end{array}$$

The function *extractG* makes a list of guards of the guarded command. In the example we have

$$\mathit{extractG}(\, x \geq 0 \rightarrow x : [\, x \geq 0, x = x_0 + 1\,] \,[]\, x < 0 \rightarrow x : [\, x < 0, x = x_0 - 1\,]\,) = \langle x \geq 0, x < 0\rangle$$

We combine the list of guards with each element of the list of *RCell*s we get from the distributed cartesian product above. The function *insertG* takes a list of guards g_i, a list of *RCell*s $(\, p_i, \mathit{pobs}_i\,)$, and returns a pair, where the first element is a guarded command and the second is a set of proof obligations.

The guarded command associates each guard g_i to the program p_i: it is $g_1 \rightarrow p_1 \,[]\, g_2 \rightarrow p_2 \,[]\, ...$ The set of proof obligations is the union of the pobs_i.

We use the function *mkGC* to apply $\mathit{insertG}\,\langle x \geq 0, x < 0\rangle$ to each element of the cartesian product above.

$$\begin{array}{l} \mathit{mkGC}\,\langle x \geq 0, x < 0\rangle \\ \qquad [\,\langle\,(\, x : [\, x > y\,], \{x \geq 0 \Rightarrow \mathit{true}\}\,), (\, x := z - 1, \{x < 0 \Rightarrow z - 1 < z\}\,)\,\rangle, \\ \qquad\ \langle\,(\, x := y + 1, \{x \geq 0 \Rightarrow y + 1 > y\}\,), (\, x := z - 1, \{x < 0 \Rightarrow z - 1 < z\}\,)\,\rangle\,] \\ = [\,(\, x \geq 0 \rightarrow x : [\, x > y\,] \,[]\, x < 0 \rightarrow x := z - 1, \{x \geq 0 \Rightarrow \mathit{true}, x < 0 \Rightarrow z - 1 < z\}\,), \\ \quad (\, x \geq 0 \rightarrow x := y + 1 \,[]\, x < 0 \rightarrow x := z - 1, \{x \geq 0 \Rightarrow y + 1 > y, x < 0 \Rightarrow z - 1 < z\}\,)\,] \end{array}$$

The last step, to rebuild the *RCell*s, uses the function Ω_{if}. The arguments of this function are the original set *pobs* of proof obligations and the list of *RCell*s $(\, \mathit{gc}_i, \mathit{pobs}_i\,)$ generated in the previous step. The result is the list of *RCell*s $(\, \mathbf{if}\ \ \mathit{gc}_i\ \mathbf{fi}, \mathit{pobs} \cup \mathit{pobs}_i\,)$; each guarded command in the argument list is turned into a conditional, and the set of original proof obligations is added to those generated by the tactic.

In our example we have

$$\begin{array}{l} \Omega_{\mathit{if}}\,\{x \geq 1\} \\ \quad [\,(\, x \geq 0 \rightarrow x : [\, x > y\,] \,[]\, x < 0 \rightarrow x := z - 1, \{x \geq 0 \Rightarrow \mathit{true}, x < 0 \Rightarrow z - 1 < z\}\,), \\ \quad\ (\, x \geq 0 \rightarrow x := y + 1 \,[]\, x < 0 \rightarrow x := z - 1, \{x \geq 0 \Rightarrow y + 1 > y, x < 0 \Rightarrow z - 1 < z\}\,)\,] \\ = [\,(\, \mathbf{if}\ x \geq 0 \rightarrow x : [\, x > y\,] \,[]\, x < 0 \rightarrow x := z - 1\ \mathbf{fi}, \\ \quad\ \{x \geq 0 \Rightarrow \mathit{true}, x < 0 \Rightarrow z - 1 < z, x \geq 1\}\,), \\ \quad (\, \mathbf{if}\ x \geq 0 \rightarrow x := y + 1 \,[]\, x < 0 \rightarrow x := z - 1\ \mathbf{fi}, \\ \quad\ \{x \geq 0 \Rightarrow y + 1 > y, x < 0 \Rightarrow z - 1 < z, x \geq 1\}\,)\,] \end{array}$$

With this example as motivation, we present the definition of the combinator for the conditional.

$$\begin{array}{l} (\, [\![\, \boxed{\mathbf{if}}\, \mathit{tacs}\, \boxed{\mathbf{fi}}\,]\!]\,\Gamma_L\,\Gamma_T\,)(\, \mathbf{if}\ \mathit{gc}\ \mathbf{fi}, \mathit{pobs}\,) = \\ \quad \Omega_{\mathit{if}}\ \mathit{pobs}\,(\mathit{mkGC}\,(\mathit{extractG}\ \mathit{gc})\,(\prod(\mathit{apply}\,((\![\_]\!]\,\Gamma_L\,\Gamma_T) * \mathit{tacs})\,(\Phi_r * (\mathit{extractP}\ \mathit{gc})))))\end{array}$$

First, we extract the commands from the branches *gc* of the conditional (*extractP*). Then, we

construct a list of *RCell*s with these commands as their programs, and an empty set of proof obligations ($\Phi_r *$).

We apply each element of the list *tacs* of tactics to the corresponding element in the list of *RCell*s we constructed ($apply\,((\llbracket\_\rrbracket\,\Gamma_L\,\Gamma_T)\,*\,tacs)$). The next step is to take the distributed cartesian product of the resulting list ($\prod$).

Finally, we rebuild the conditionals with the resulting commands ($mkGC\,(extractG\,gc)$) and add the original proof obligations *pobs* to the new sets of proof obligations ($\Omega_{if}\,pobs$). The similar definition of $\boxed{\textbf{do}}\_\boxed{\textbf{od}}$ may be found in [Oli00].

The structural combinator $\boxed{\textbf{var}}\_\boxed{\rrbracket}$ applies a tactic to the program in a variable block and rebuilds the variable blocks. Its formal definition is

$$(\,\llbracket\,\boxed{\textbf{var}}\,t\,\boxed{\rrbracket}\,\rrbracket\Gamma_L\,\Gamma_T)\,(\,\llbracket\,\textbf{var}\;d\bullet p\,\rrbracket\,,pobs\,)=(var\;d)*(\,\llbracket\,t\,\rrbracket\,\Gamma_L\,\Gamma_T\,(p,pobs)\,)$$

The function *var* rebuilds the variable block after the application of the tactic to its body. It takes as arguments a variable declaration and an *RCell*. The result is a new *RCell* containing a variable block built from the declaration and the program. The proof obligations are not changed.

$$var : Declaration \rightarrow RCell \rightarrow RCell$$

$$var\;d\,(p,pobs)=(\,\llbracket\,\textbf{var}\;d\bullet p\,\rrbracket\,,pobs\,)$$

The structural combinator $\boxed{\textbf{con}}\_\boxed{\rrbracket}$ is defined similarly.

The structural combinator $\boxed{\textbf{pmain}}\_\boxed{\rrbracket}$ is used to apply a tactic to the main program of a procedure block. Its definition is as follows.

$$(\,\llbracket\,\boxed{\textbf{pmain}}\,t\,\boxed{\rrbracket}\,\rrbracket\,\Gamma_L\,\Gamma_T\,)\,(\,\llbracket\,\textbf{proc}\;n\mathrel{\widehat{=}}p_1\bullet p_2\,\rrbracket\,,pobs\,)=(procm\;n\;p_1)*(\,\llbracket\,t\,\rrbracket\,\Gamma_L\,\Gamma_T\,(p_2,pobs)\,)$$

where

$$procm : name \rightarrow program \rightarrow RCell \rightarrow RCell$$

$$procm\;n\;p_1\,(p_2,pobs)=(\,\llbracket\,\textbf{proc}\;n\mathrel{\widehat{=}}p_1\bullet p_2\,\rrbracket\,,pobs\,)$$

The function *procm* is similar to *var*; it rebuilds the procedure block. The definitions of the other combinators that apply to procedure and variant blocks are very much like those shown here.

For parameter passing, we give as an example the semantics of $\boxed{\textbf{val}}\_$. Its definition is

$$\llbracket\,\boxed{\textbf{val}}\,t\,\rrbracket\,\Gamma_L\,\Gamma_T\,(\,(\textbf{val}\;v:T\bullet p)\,(a),pobs\,)=(val\;v\;T\;a)*(\,\llbracket\,t\,\rrbracket\,\Gamma_L\,\Gamma_T\,(p,pobs)\,)$$

where

$$val : name \rightarrow Type \rightarrow args \rightarrow RCell \rightarrow RCell$$

$$val\;v\;T\;a\,(p,pobs)=(\,(\textbf{val}\;v:T\bullet p)(a),pobs\,)$$

This structural combinator applies the tactic to the body of the parameterised command and then rebuilds it, using the function *val*.

The tactic $\textbf{con}\,v\bullet t$ introduces v as a set of variables whose values are taken from an appropriate syntactic class denoted *TERM* below. These variables are used in t as elements of its syntactic class. Their values are angelically chosen so that the tactic succeeds. Its semantics is

$$\llbracket\,\textbf{con}\,v\bullet t\,\rrbracket\,\Gamma_L\,\Gamma_T=\llbracket\,|_{v\in TERM}\,t(v)\,\rrbracket\,\Gamma_L\,\Gamma_T$$

where $|_{v\in TERM}\,t(v)$ is an alternation of all tactics that can be obtained from t, with v ranging over all values in *TERM*. This is an infinite alternation, which can be defined as follows.

$$|\;{}_{i=0}^{\infty}\;f(i)=\mu\,X\bullet F(X_0)$$

where $F(X_i)=f(i)\mid F(X_{i+1})$.

The tactic $\textbf{applies to}\;\;p\;\textbf{do}\;t$ needs to consider all ways in which the given program can match

p. Its semantics uses the function *equals*, that yields a tactic that succeeds only if it is presented with a *RCell* matching the given program:

$$equals : RCell \rightarrow Tactic$$

$$equals\ r\ r = [r]$$
$$equals\ r\ h = [\,], \textbf{if}\ h \neq r$$

The definition of the tactic **applies to** p **do** t is

$$[\![\, \textbf{applies to}\ \ p\ \textbf{do}\ t \,]\!]\, \Gamma_L\, \Gamma_T\, (p_1, pobs) = [\![\, \textbf{con}\, p \bullet equals\, (p, pobs);\ \ t \,]\!]\, \Gamma_L\, \Gamma_T\, (p_1, pobs)$$

For simplicity, we use the meta-program p as a variable itself; the alternative is to consider the individual variables of p. In $\textbf{con}\, p \bullet equals\, (p, pobs);\ \ t$, an instantiation of p is angelically chosen to ensure the success of $equals\, (p, pobs);\ \ t$. The tactic $equals\, (p, pobs)$ succeeds only for those instantiations that match its argument: $(p_1, pobs)$; so $equals\, (p, pobs)$ is a filter for the instantiations of p: only those that match p_1 are considered. If there is none, the tactic fails. If a successful instantiation can be chosen, the instantiated values are used in t, which is applied to $(p_1, pobs)$.

4.3. Tactic Declarations and Programs

A *tacDec* includes a tactic name n in the domain of the tactics environment. This element is mapped to a new function that, for each possible argument $v \in TERM$, gives the semantics of the tactic when the arguments of the tactic are replaced by v. The clauses **proof obligations** and **program generated** do not change the semantics of a *tacDec* given below.

$$[\![\_]\!] : tacDec \rightarrow LEnv \rightarrow TEnv \rightarrow TEnv$$

$$[\![\, \textbf{Tactic}\, n(a)\, t\ \textbf{end}\ ]\!]\, \Gamma_L\, \Gamma_T = \Gamma_T \oplus \{n \mapsto \{\, v \in TERM \bullet v \rightarrow [\![\, t[a \setminus v] \,]\!]\, \Gamma_L\, \Gamma_T \,\}\}$$

where $\oplus$ is the overriding operator.

A *tacProg* is a sequence *tds* of *tacDec* followed by a main tactic t. The semantics is that of t, when evaluated in the environment determined by the tactic declarations and the given laws environment. The definition is as follows.

$$[\![\_]\!] : tacProg \rightarrow LEnv \rightarrow TEnv \rightarrow TEnv$$

$$[\![\, tds\ t \,]\!]\, \Gamma_L\, \Gamma_T = [\![\, t \,]\!]\, \Gamma_L\, (decl\, tds\, \Gamma_L\, \emptyset)$$

This environment is defined by the function *decl*.

$$decl : tacDec* \rightarrow LEnv \rightarrow TEnv \rightarrow TEnv$$

$$decl\, \langle\rangle\, \Gamma_L\, \Gamma_T = \Gamma_T$$
$$decl\, (td_1\ tds)\, \Gamma_L\, \Gamma_T = (\ decl\, tds\, \Gamma_L\, (\ [\![\, td_1 \,]\!]\, \Gamma_L\, \Gamma_T\ )\ )$$

In the case that we have an empty tactic declaration, this function returns the tactics environment given as argument. Otherwise, it uses the tactic environment resulting from the first tactic declaration to evaluate the rest of the declarations.

4.4. Laws of ArcAngel

We have proved most of the laws proposed in [MGW96] for Angel in the context of ArcAngel [Oli00]. Some of these laws are used to reduce tactics to a normal form proposed for a subset of ArcAngel. As examples of these laws, we have associativity laws for tactic sequences and alternations, and distributive laws for alternation over sequence, and vice-versa. We have that **skip** is a neutral element in sequential composition, as **fail** is for alternation; the sequential composition of **fail**

with any tactic is the same as **fail** itself. A few more interesting laws are shown below.

Law $!t_1;\ !t_2 = !(!t_1;\ !t_2)$
Law $\mathbf{succs}\, t;\ t = t$
Law $t_1 \,[;]\, (t_2 \mid t_3) = (t_1 \,[;]\, t_2) \mid (t_1 \,[;]\, t_3)$
Law $(t_1 \mid t_2) \,[;]\, t_3 = (t_1 \,[;]\, t_3) \mid (t_2 \,[;]\, t_3)$

The first law shows that it is unnecessary to avoid backtracking in a sequential composition that already avoids backtracking in each of its parts. The second shows that to verify that t succeeds and then to apply t is the same as applying t directly. Finally, the third and the fourth laws show that the structural combinator $[;]$ distributes over alternation.

We have proved in [Oli00] that tactics written in the subset of ArcAngel, using basic laws, alternation, sequential composition, **skip**, **fail**, and !, have a normal form and that it is unique. This normal form provides a notion of completeness for the set of laws we have proved. This normal form is similar to that presented in [MGW96], and since we proved that the Angel's laws are valid for ArcAngel, we can adopt a similar normal-form reduction strategy.

5. Examples

In this section we give a few examples of ArcAngel programs. The first two, *followingAssign* and *leadingAssign*, implement two derived rules from Morgan's calculus. Their implementations demonstrate the usefulness of ArcAngel's logical constants in pattern-matching. The third program formalises a strategy from the literature for developing a loop.

5.1. Following assignment

In [Mor94, p.32], the derived rule of *following assignment* is presented as a combination of the *assignment* and *sequential composition* laws; in other words, it is rather like a tactic for using the more basic laws of the calculus. Of course, there is a big difference between a tactic and a derived rule. The former is a program that applies rules; proof obligations arise from the law applications. A derived rule is a law itself, that is proved in terms of applications of other laws. Tactics are much more flexible, since they can make choices about the form of the resulting program.

Following assignment splits a specification statement into two pieces, the second of which is implemented by the assignment $x := E$. If we apply $seqComp(\ post[x \setminus E]\ )$ to $w, x : [\ pre, post\ ]$ we get the program

$$w, x : [\ pre, post[x \setminus E]\ ];\ w, x : [\ post[x \setminus E], post\ ]$$

If we now apply $assign(\ x := E\ )$ to the second statement, we get our assignment $x := E$ and the proof obligation $post[x \setminus E] \Rightarrow post[x \setminus E]$. This is a simple tautology, but remains part of the documentation of the tactic, as the proof obligation is raised every time the tactic is applied.

Tactic *followingAssign* $(\ x, E\ )$
applies to $w, x : [\ pre, post\ ]$ **do**
law $seqComp(\ post[x \setminus E])$; (**skip** $[;]$ **law** $assign(\ x := E\ )$)
proof obligations
1. $post[x \setminus E] \Rightarrow post[x \setminus E]$
program generated
$w, x : [\ pre, post[x \setminus E]\ ];$
$x := E$
end

There is a further restriction on the use of this tactic: it uses a simple form of the law of sequential

composition that forbids initial variables in postconditions. If applied to a specification statement that does not satisfy this restrictions, *followingAssign* fails.

5.2. Leading assignment

In [Mor94, p.71], the presentation of the derived rule for *leading assignment* is a little more complicated than that of the *following assignment*:

Law 13.1 § *leading assignment* For any expression E,

$$\begin{array}{ll} & w, x : [\, pre[x \setminus E], post[x_0 \setminus E_0] \,] \\ \sqsubseteq & x := E; \\ & w, x : [\, pre, post \,] \end{array}$$

The expression E_0 is that obtained from E by replacing all free occurrences of x and w with x_0 and w_0, respectively. The complication arises from the need to find predicates *pre* and *post*, such that, when appropriate substitutions are made, they match the current goal. The solution in ArcAngel is to use logical constants.

Tactic *leadingAssign* (x, X, E)
applies to $w, x : [\, pre[x \setminus E], post[x_0 \setminus E_0] \,]$ **do**

law *seqCompIV*($x, X, pre \wedge x = E_0$);
con **law** *assignIV*($x := E$); **law** *strPostIV*(*post*); **law** *weakPre*(*pre*) ⟦⟧;
law *removeCon*(X)

proof obligations

1. $(x = x_0) \wedge pre\,[x \setminus E] \Rightarrow (\, pre \wedge x = E[x \setminus x_0])[x \setminus E]$
2. $(\, pre \wedge x = E\,[x \setminus X])[x, w \setminus x_0, w_0] \wedge post \Rightarrow post[x_0 \setminus E_0][x_0 \setminus X]$
3. $(\, pre \wedge x = E\,[x \setminus X]) \Rightarrow pre$

program generated

$x := E;\ w, x : [\, pre, post \,]$

end

If applied to a specification statement $w, x;\ [pre[x \setminus E], post[x_0 \setminus E_0]]$ this tactic first splits it using the law *seqCompIV*. As *seqComp*, this law introduces sequences, but it applies to specification statements that make use of initial variables; it declares a logical constant to record the initial value of x. This gives:

⟦ **con** X •
$x : [pre\,[x \setminus E], pre \wedge x = E\,[x \setminus x_0]\,];\ w, x : [pre \wedge x = E[x \setminus X], post[x_0 \setminus E_0][x_0 \setminus X]\,]$
⟧

The first specification statement is refined to $x := E$, using the *assignIV* law: a law for introducing assignments that considers the presence of initial variables. The precondition and postcondition of the second specification statement are changed to *pre* and *post*, respectively, and finally the logical constant is removed. The law *strPostIV* is the strengthening postcondition law that takes initial variables into account. The proof obligations are generated by the applications of the laws *assignIV*, *strPostIV*, and *weakPre*, respectively; they always hold.

5.3. Tactic *replConsByVar*

In [OC00], several well-known strategies for developing loops [Gri81, Kal90] are formalised in ArcAngel, and we give an example of one of the resulting tactics here. The tactic *replConsByVar*

allows the user to choose an iteration invariant by replacing a constant in the postcondition by a variable.

The tactic *replConsByVar* has five arguments: the declaration of the fresh loop-index variable $newv : T$; the constant to be replaced *cons*; an invariant on the loop indexes *invBound*; the loop initialisation $ivar := ival$; and an integer-valued *variant*.

Tactic *replConsByVar* ($newv : T, cons, invBound, ivar := ival, variant$)
applies to $w : [\, pre, post \,]$ **do**

law *varInt*($newv : T$);
var **law** *strPost*($post[cons \setminus newv] \land newv = cons$);
tactic *takeConjAsInv*($invBound, ivar := ival, variant$); $[\![]\!]$

proof obligations

1. $post[cons \setminus newv] \land newv = cons \Rightarrow post$
2. $post[cons \setminus newv] \land newv = cons \land invBound \Rightarrow post[cons \setminus newv] \land newv = cons$
3. $pre \Rightarrow (\, post[cons \setminus newv] \land invBound \,)[ivar \setminus ival]$

program generated

$[\![$ **var** $newv : T \bullet$
$ivar := ival;$
do $\neg\ newv = cons \rightarrow$
$newv, w : [\, post[cons \setminus newv] \land invBound \land \neg\ newv = cons,$
$post[cons \setminus newv] \land invBound \land 0 \leq variant < variant[newv \setminus newv_0] \,]$
od
$]\!]$

end

This tactic first introduces the new variable, then it strengthens the postcondition to replace the constant by the new variable. Afterwards it calls the tactic *takeConjAsInv* to introduce the iteration. The proof obligation 1 is generated by the law *strPost*, and the others are generated by the tactic *takeConjAsInv*. The proof obligations 1 and 2 always hold. The tactic *takeConjAsInv* implements another, simpler strategy for loop development: taking a conjunct of the postcondition as the main part of the invariant.

Tactic *takeConjAsInv* ($invBound, ivar := ival, variant$)
applies to $w, ivar : [\, pre, invConj \land notGuard \,]$ **do**

law *strPost*($invConj \land notGuard \land invBound$);
law *seqCom*($invConj \land invBound$);
(**law** *assign*($ivar := ival$) [;] **law** *iter*($\langle \neg\ notGuard \rangle, invConj \land invBound, variant$));

proof obligations

1. $invConj \land notGuard \land invBound \Rightarrow invConj \land notGuard$
2. $pre \Rightarrow (\, invConj \land invBound \,)[ivar \setminus ival]$

program generated

$ivar := ival;$
do $\neg\ notGuard \rightarrow$
$w : [\, invConj \land invBound \land \neg\ notGuard,$
$invConj \land invBound \land 0 \leq variant < variant_0 \,]$
od

end

This transforms a specification $w : [pre, invconj \land notGuard]$ into an initialised iteration, and has three arguments: an invariant *invBound* on the loop indexes; the loop initialisation $ivar := ival$; and the integer-valued *variant*.

It first strengthens the postcondition using the argument *invBound*, then introduces a sequential composition. Next, the law *assign* is applied to the first program of the composition to introduce the initialisation, and the law *iter* is applied to the second program in order to introduce the iteration. The first proof obligation is generated by the application of law *strPost*, and the second by the application of law *assign*. The first proof obligation always holds.

Consider the problem described in [WD96, p.314, Example 19.15]: *"We would like to develop an algorithm that converts numbers from a base β to the base* 10. *For an $n+1$ digit number, a solution that requires more than n multiplications is not acceptable."* The problem is to compute the value of a polynomial $\sum_{i=1}^{n} a_i * \beta^{i-1}$ efficiently, where the a_is are the digits in base β. The solution, well-known to numerical analysts, involves Horner's rule: the polynomial has the same value as $H_{1,n}$, where $H_{n,n} = a_n$, and $H_{i,n} = a_i + \beta * H_{i+1,n}$, for $i < n$.

The specification statement $d : [\, d = H_{1,n} \,]$ may be refined to a program that initialises d to $H_{n,n}$, and then repeatedly multiplies d by β and adds the next coefficient, until it finally adds a_1. To develop this program, we use the tactic of replacing a constant by a variable. We first introduce a fresh loop-index variable, j, that ranges between n and 1, requiring that, at any instant, d will have the value $H_{j,n}$; this is our invariant for the loop.

Our specification is refined to code by the tactic

$$\begin{array}{l} \mathit{replConsByVar}(\, j : \mathbb{Z}, 1, 1 \leq j \leq n, (j, d := n, a_n), j - 1 \,); \\ \boxed{\textbf{var}}\, \textbf{skip}\, ; \boxed{\textbf{do}}\, \mathit{assign}(\, j, d := j - 1, a_{j-1} + \beta * d \,) \boxed{\textbf{od}} \end{array}$$

It generates the following proof obligations:

1. $d = H_{j,n} \wedge j = 1 \Rightarrow d = H_{1,n}$
2. $d = H_{j,n} \wedge j = 1 \wedge 1 \leq j \leq n \Rightarrow d = H_{j,n} \wedge j = 1$
3. $\mathit{true} \Rightarrow (\, d = H_{j,n} \wedge 1 \leq j \leq n \,)[j, d \setminus n, a_n]$

and the following program.

$$\begin{array}{l} [\![\ \textbf{var}\ j : \mathbb{Z} \bullet \\ \quad j, d := n, a_n; \\ \quad \textbf{do}\ \neg\ j = 1 \rightarrow j, d := j - 1, a_{j-1} + \beta * d\ \textbf{od} \\ ]\!] \end{array}$$

The first two proof-obligations are trivial, and the last one follows from the definition of $H_{n,n}$.

6. Related work

Gardiner and Vickers developed the refinement tool `Red`, reported in [Vic90]. It is implemented in Prolog and maintains three interactive windows: one contains the state of the program text being developed; the second contains the proof obligations; and the third contains a tree of the refinement operations performed to obtain the program. In [Vic94], the authors describe a language for representing the refinement tree. The key idea is that, for every construct of the target programming language, there is a corresponding construct in the transformation language. This work is the origin of Angel's *structural combinators* [MGW93]. The language presented, however, is not a language to describe tactics; it is, as already said, a language to describe refinement. There are no constructs for alternation or recursion.

Grundy [Gru92] proposes a refinement tool based on *window inference*. The tool runs on top of the HOL theorem prover; because the HOL system is programmable, users can add their own commands to automate refinements that they frequently repeat, although this is not the main focus of the work. Tactics are written using simple tacticals, such as `THEN`, `ORELSE`, and `REPEAT`.

Later, Grundy and his colleagues [BGL+97] extended that work to produce a more user-friendly tool with a graphical interface. The authors mention that ML can be used to package transformations, but point out themselves that this brings the difficulty of requiring knowledge of HOL and ML. While ML does have a formal semantics, it is difficult to reason about tactics written in such a general language.

Groves, Nickson, and Utting [GNU92, Nic94] describe a refinement tool, placing special emphasis on tactics. The tool is implemented in Prolog, whose programming capabilities are available to the user. Nickson's thesis [Nic94] contains a collection of examples showing how various common development patterns (like introducing various kinds of loops) may be encoded as tactics. These tactics are expressed directly in Prolog, rather than a special tactic language. This means that tactics have all the normal Prolog control mechanisms and can do arbitrary computations in deciding what steps to take, and constructing new components, but all modifications to the program being constructed are done by applying refinement rules. In their tool, the rule for leading assignment is programmed as

```
-- Leading Assignment
tactic leading_assignment(V := E)[get_assignment(V := E)]
   when free_of_var(V, Pre):
   Vars: [Pre, Post] <<
      (
         refine Vars: [Pre, Post]
            using split_spec(Pre and V = E)
            giving ( AssSpec; Rest ),
         refine AssSpec using intro_assign(V := E)
      ).
```

If the parameter to the tactic `V := E` is not supplied, the user is prompted to supply it interactively. The subject of this tactic will match any specification statement that does not mention the target of the assignment in its precondition. The body of the tactic is a sequence of two refinement steps. The first uses a rule of sequential composition to split the specification statement into two parts: `AssSpec`, the assignment specification, and the `Rest` of the specification. The `giving` clause is used to associate names to components of programs resulting from law applications and allow subsequent law applications to these components; this is a role similar to that of ArcAngel's. The second step of `leading_assignment` applies a rule of assignment introduction to `AssSpec`; as there is nothing further to refine from this sub-program, there is no `giving` clause.

Van de Snepscheut [vdS94] describes a notation for formula manipulation and an associated editor that together provide support for the development of programs by stepwise refinement (`Proxac`). Although the idea seems not to have been taken further, van de Snepscheut suggested viewing the program transformations as the commands in a programming language for formula manipulation including function composition, conditionals, and a fixed-point operator.

Queensland University's program refinement tool, PRT [CHN+98], originally had a tactic language inherited from Ergo, on which it was based. Later, a tactic language based on Angel was added; this language, Gumtree, is described in [MNU97b, MNU97a].

Back and von Wright [BW90] show how to use HOL to prove refinement rules correct and how to apply such rules to formalise program refinement. In [WHLL93], von Wright and his colleagues continue this work by describing how the HOL theorem prover may be used to apply the rules from a refinement calculus. They formalise methods for data refinement, proving the validity of refinement rules within their theory. The main aim of the work is to use HOL to generate the verification conditions for refinement steps. Simple tactics for refinement are programmed in ML.

A survey of various early refinement tools may be found in [CHN+94].

7. Conclusions

We have presented ArcAngel, a refinement-tactic language. Using this language, it is possible to specify commonly used strategies of program development. Tactics can be used as transformation rules, which shortens developments and improves their readability.

We defined the semantics of ArcAngel based on Angel semantics. The difference is that Angel is a very general language and its goals have no defined structure. For us, a goal is called an *RCell*, which is a pair with a program as its first element and a set of proof obligations as its second

element. The application of a tactic to an *RCell* returns a list of *RCell*s containing the possible output programs with their corresponding set of proof obligations.

We have shown the soundness of algebraic laws for reasoning about ArcAngel tactics. We covered most of the laws that have been proposed for Angel. For the vast majority of them, proofs are not available in the literature and are provided in [Oli00] in the context of ArcAngel. Since these laws are valid, the strategy proposed to reduce finite Angel tactics to a normal form can be applied to ArcAngel tactics. In [Oli00] we have expanded the proofs of the lemmas and theorems involved, exemplifying some of the procedures.

Generalising the normal form to encompass the rest of the language is not trivial and is still to be done. Basically, there are three points to conclude this task. We must include the basic tactic **tactic** n. The idea is to introduce the body of n wherever this tactic is applied. The other points consist of including recursion and the structural combinators. We have to extend the normal form to handle these. In the case of structural combinators, we believe we should allow their occurrence wherever a basic law is allowed in the presented normal form, and require that the component tactics in the structural combinators are already in the normal form.

As well as carrying on further work on the proof of laws and on defining a normal form for the whole of ArcAngel, we intend to implement a tool to support the definition and application of ArcAngel tactics. This tool is going to be provided as an extension to Refine [CRC99], a refinement tool developed by our group.

Our work, however, has shown that there are very interesting issues in specialising Angel to the refinement calculus, and this is independent of any particular tool. These issues are worthy of further investigation in their own right.

Acknowledgements

We are grateful to Lindsay Groves for very detailed comments and insights. We also thank Andrew Martin, and Ray Nickson for valuable discussions about refinement, mechanisation, and tactics.

References

[Bac78] R. J. R. Back. *On The Correctness of Refinement Steps in Program Development.* PhD thesis, Department of Computer Science, University of Helsinki, 1978. Report A-1978-4.

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

[BGL+97] M. J. Butler, J. Grundy, T. Långbacka, R. Rukšėnas, and J. Wright. The Refinement Calculator: Proof Support for Program Refinement. In *Proceedings of FMP'97 – Formal Methods Pacific*, Discrete Mathematics and Theoretical Computer Science. Springer-Verlag, 1997.

[BvW98] R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, 1998.

[BW90] R. J. R. Back and J. Wright. Refinement Concepts Formalised in Higher Order Logic. *Formal Aspects of Computing*, 2:247–274, 1990.

[CHN+94] David Carrington, Ian Hayes, Ray Nickson, Geoffrey Watson, and Jim Welsh. A review of existing refinement tools. Technical Report 94-8, Software Verification Research Centre, The University of Queensland, 1994.

[CHN+98] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A program refinement tool. *Formal Aspects of Computing*, 10(2):97–124, 1998.

[CM81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[CRC99] S. L. Coutinho, T. P. C. Reis, and A. L. C. Cavalcanti. Uma Ferramenta Educacional de Refinamentos. In *XIII Simpósio Brasileiro de Engenharia de Software*, pages 61 – 64, Florianópolis - SC, 1999. Sessão de Ferramentas.

[CSW97] A. L. C. Cavalcanti, A. Sampaio, and J. C. P. Woodcock. Procedures, Parameters, and Substitution in the Refinement Calculus. Technical Report TR-5-97, Oxford University Computing Laboratory, Oxford - UK, February 1997.

[CSW98] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Procedures and Recursion in the Refinement Calculus. *Journal of the Brazilian Computer Society*, 5(1):1–15, 1998.

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

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
[GNU92] L. Groves, R. Nickson, and M. Utting. A Tactic Driven Refinement Tool. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, pages 272 – 297. Springer-Verlag, 1992.
[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
[Gru92] J. Grundy. A Window Inference Tool for Refinement. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, pages 230 – 254. Springer-Verlag, 1992.
[Kal90] A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice-Hall, 1990.
[Lan91] Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer, 1991.
[Mar95] A. Martin. Infinite Lists for Specifying Functional Programs in Z. Technical report, University of Queensland, Queensland - Australia, March 1995.
[Mar96] A. P. Martin. *Machine-Assisted Theorem Proving for Software Engineering*. PhD thesis, Oxford University Computing Laboratory, Oxford, UK, 1996. Technical Monograph TM-PRG-121.
[MGW93] A. P. Martin, P. H. B. Gardiner, and J. C. P. Woodcock. Tactic Semantics and Reasoning. Technical report, Oxford University Computer Laboratory, Oxford - UK, December 1993. Draft version.
[MGW96] A. P. Martin, P. H. B. Gardiner, and J. C. P. Woodcock. A Tactical Calculus. *Formal Aspects of Computing*, 8(4):479–489, 1996.
[MNU97a] Andrew Martin, Ray Nickson, and Mark Utting. Improving Angel's parallel operator: Gumtree's approach. Technical Report 97-15, Software Verification Centre, School of Information Technology, The University of Queensland, December 1997.
[MNU97b] Andrew Martin, Ray Nickson, and Mark Utting. A tactic language for Ergo. Technical Report 97-16, Software Verification Centre, Department of Computer Science, The University of Queensland, February 1997.
[Mor87] J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287 – 306, 1987.
[Mor94] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.
[Nic94] Raymond George Nickson. *Tool Support for the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 1994.
[OC00] M. V. M. Oliveira and A. L. C. Cavalcanti. Tactics of Refinement. In *XIV Simpósio Brasileiro de Engenharia de Software*, pages 117 – 132, 2000.
[Oli00] M. V. M. Oliveira. Tactics of Refinement. Technical report, Centro de Informática - Universidade Federal de Pernambuco, Pernambuco - Brazil, December 2000. At http://www.cin.ufpe.br/~lmf.
[vdS94] Jan L. A. van de Snepscheut. Mechanised support for stepwise refinement. In Jürg Gutknecht, editor, *Programming Languages and System Archtectures*, volume 782 of *Lecture Notes in Computer Science*, pages 35–48. Springer, March 1994. Zurich, Switzerland.
[Vic90] T. Vickers. An Overview of a Refinement Editor. In *5th Australian Software Engineering Conference*, pages 39–44, Sydney - Australia, May 1990.
[Vic94] Trevor Vickers. A language of refinements. Technical Report TR-CS-94-05, Computer Science Department, Australian National University, 1994.
[WD96] Jim Woodcock and Jim Davies. *Using Z – Specification, Refinement, and Proof*. Prentice-Hall, 1996.
[WHLL93] J. Wright, J. Hekanaho, P. Luostarinen, and T. Långbacka. Mechanizing Some Advanced Refinement Concepts. *Formal Methods in System Design*, 3:49–81, 1993.

A. Infinite Lists

We present the model for infinite lists adopted here from [Mar95]. The set of the finite and partial sequences of members of X is defined as

$$\text{pfseq}\, X ::= \mathsf{partial}\langle\!\langle \text{seq}\, X \rangle\!\rangle \mid \mathsf{finite}\langle\!\langle \text{seq}\, X \rangle\!\rangle$$

We define an order $\sqsubseteq$ on these pairs such that for $a, b : \text{pfseq}\, X$, if a is finite, then $a \sqsubseteq b$ if, and only if, b is also finite and equal to a. If a is partial, then $a \sqsubseteq b$ if, and only if, a is a prefix of b.

$$\_ \sqsubseteq \_ : \text{pfseq}\, X \leftrightarrow \text{pfseq}\, X$$

$$\begin{array}{l}\forall\, gs, hs : \text{seq}\, X \bullet \\ \quad \mathsf{finite}\, gs \sqsubseteq \mathsf{finite}\, hs \Leftrightarrow gs = hs \\ \quad \mathsf{finite}\, gs \sqsubseteq \mathsf{partial}\, hs \Leftrightarrow \mathit{false} \\ \quad \mathsf{partial}\, gs \sqsubseteq \mathsf{finite}\, hs \Leftrightarrow gs\ \mathsf{prefix}\ hs \\ \quad \mathsf{partial}\, gs \sqsubseteq \mathsf{partial}\, hs \Leftrightarrow gs\ \mathsf{prefix}\ hs\end{array}$$

A chain of sequences is a set whose elements are pairwise related.

$$chain : \mathbb{P}(\mathbb{P}(\text{pfseq}\, X))$$
$$\forall\, c : \mathbb{P}(\text{pfseq}\, X) \bullet c \in chain \Leftrightarrow (\forall\, x, y : c \bullet x \sqsubseteq y \lor y \sqsubseteq x\,)$$

The set pchain contains all downward closed chains.

$$pchain : \mathbb{P}\, chain[X]$$
$$\forall\, c : chain[X] \bullet c \in pchain \Leftrightarrow (\forall\, x : c;\ y : \text{pfseq}\, X \mid y \sqsubseteq x \bullet y \in c\,)$$

The set pfiseq contains partial, finite, and infinite list of elements of X, which are prefixed-closed chains of elements in $\text{pfseq}\, X$.

$$\text{pfiseq}\, X == pchain[X]$$

The idea is that $\bot = \mathsf{partial}\,\langle\rangle$, the empty list $[\,] = \mathsf{finite}\,\langle\rangle$, the finite list $[e_1, e_2, \ldots, e_n]$ is represented by the set containing $\mathsf{finite}\,\langle e_1, e_2, \ldots, e_n\rangle$ and all approximations to it. An infinite list is represented by an infinite set of partial approximations to it. The infinite list itself is the least upper bound of such a set.

The definitions of the functions used in this paper are as follows.

1. The map function maps a function f to each element of a possibly infinite list.

$$pfmap : (X \to Y) \to \text{pfseq}\, X \to \text{pfseq}\, Y$$
$$\forall\, xs : \text{seq}\, X;\ f : X \to Y \bullet$$
$$pfmap\, f\, (\mathsf{finite}\, xs) = \mathsf{finite}\, (f \circ xs) \land$$
$$pfmap\, f\, (\mathsf{partial}\, xs) = \mathsf{partial}\, (f \circ xs)$$

$$* : (X \to Y) \to \text{pfiseq}\, X \to \text{pfiseq}\, Y$$
$$\forall\, c : \text{pfiseq}\, X;\ f : X \to Y \bullet$$
$$f * \bot = \bot$$
$$f * c = \{\, x : c \bullet pfmap\, f\, x\,\}$$

   The function *pfmap* maps the function f to the second element of x.

2. The distributed concatenation returns the concatenation of all the elements of a possibly infinite list of possibly infinite lists.

$$\overset{\infty}{/} : \text{pfiseq}(\text{pfiseq}\, X) \to \text{pfiseq}\, X$$
$$\forall\, s : \text{pfiseq}(\text{pfiseq}\, X) \bullet$$
$$\overset{\infty}{/}\, s = \bigsqcup\nolimits_{\infty} \{\, c : s \bullet \overset{\infty}{\frown}/\, c\,\}$$

   It uses the function $\overset{\infty}{\frown}/$, which is the distributed concatenation for $\text{pfseq}(\text{pfiseq}\, X)$. The function *cat* is the standard concatenation function for X^*.

$$\overset{\infty}{\frown}/ : \text{pfseq}(\text{pfiseq}\, X) \to \text{pfiseq}\, X$$
$$\overset{\infty}{\frown}/(f, \langle\rangle) = \emptyset$$
$$\overset{\infty}{\frown}/(p, \langle\rangle) = (p, \langle\rangle)$$
$$\overset{\infty}{\frown}/(f, \langle g\rangle) = [g]$$
$$\overset{\infty}{\frown}/(p, \langle g\rangle) = [g] \frown \{(p, \langle\rangle)\}$$
$$\overset{\infty}{\frown}/(f, gs \frown hs) = (\overset{\infty}{\frown}/(f, gs)) \overset{\infty}{\frown} (\overset{\infty}{\frown}/(f, hs))$$
$$\overset{\infty}{\frown}/(f, gs \frown hs) = (\overset{\infty}{\frown}/(f, gs)) \overset{\infty}{\frown} (\overset{\infty}{\frown}/(p, hs))$$

   The function $\overset{\infty}{\frown}$ is the concatenation function for possibly infinite lists. Its definition is

$$\_ \overset{\infty}{\frown} \_ : \text{pfiseq}\, X \times \text{pfiseq}\, X \to \text{pfiseq}\, X$$
$$\forall\, a, b : \text{pfiseq}\, X \bullet$$
$$a \overset{\infty}{\frown} b = \{\, x : a;\ y : b \bullet x \frown y\,\}$$

where the function $^\frown$ is the concatenation function for pfseq X defined as

$$\_ \,^\frown\, \_ :: \text{pfseq}\, X \times \text{pfseq}\, X \to \text{pfseq}\, X$$

$$\begin{array}{l}\forall\, gs, hs : \text{seq}\, X;\ s : \text{pfseq}\, X \bullet \\ \quad \mathsf{finite}\, gs \,^\frown\, \mathsf{finite}\, hs = \mathsf{finite}\, (gs \frown hs) \\ \quad \mathsf{finite}\, gs \,^\frown\, \mathsf{partial}\, hs = \mathsf{partial}(gs \frown hs) \\ \quad \mathsf{partial}\, gs \,^\frown\, s = \mathsf{partial}\, gs\end{array}$$

3. The function $head'$ returns a list containing the first element of a possibly infinite list.

$$head' : \text{pfiseq}\, X \to \text{pfiseq}\, X$$

$$head'\, xs = take\, 1\, xs$$

It uses the function *take* that returns a list containing the first n elements of a possibly infinite list.

$$take : \mathbb{N} \to \text{pfiseq}\, X \to \text{pfiseq}\, X$$

$$\begin{array}{l}take\, n\, \bot = \bot \\ take\, 0\, xs = [\,] \\ take\, n\, [\,] = [\,] \\ take\, n\, xs = [head\, xs] \frown (take\, (n-1)\, (tail\, xs))\end{array}$$

For a list $(x : xs)$, the function *head* returns x and the function *tail* returns xs. For a pair (a, b), the function *first* returns a and the function *second* returns b.

4. The function $^\circ$ applies a possibly infinite list of functions to a single argument.

$$\_^\circ : \text{pfiseq}(X \nrightarrow Y) \to X \to \text{pfiseq}\, Y$$

$$\begin{array}{l}\bot^\circ\, x = [\,] \\ [\,]^\circ\, x = [\,] \\ [f]^\circ\, x = [f\, x] \\ (fs \frown gs)^\circ\, x = fs^\circ\, x \frown gs^\circ\, x\end{array}$$

5. The function $\prod$ is the distributed cartesian product for possibly infinite lists.

$$\prod : \text{seq}(\text{pfiseq}\, X) \to \text{pfiseq}(\text{seq}\, X)$$

$$\begin{array}{l}\prod\langle xs\rangle = e2l * xs \\ \prod(xs : xss) = [(a : as) \mid a \leftarrow xs, as \leftarrow \prod xss] \\ \text{where} \\ e2l\, x = [x]\end{array}$$

B. Refinement Laws

Law *strPost*($post_2$).
$w : [pre, post_1] \sqsubseteq w : [pre, post_2]$
provided $post_2 \Rightarrow post_1$

Law *strPostIV*($post_2$).
$w : [pre, post_1] \sqsubseteq w : [pre, post_2]$
provided $pre[w \setminus w_0] \wedge post_2 \Rightarrow post_1$

Law *weakPre*(pre_2).
$w : [pre_1, post] \sqsubseteq w : [pre_2, post]$
provided $pre_1 \Rightarrow pre_2$

Law *assign*(w, E).
$w : [pre, post] \sqsubseteq w := E$
provided $pre \Rightarrow post[w \setminus E]$

Law *assignIV*(w, E).
$w, x : [pre, post] \sqsubseteq w := E$
provided $(w = w_0) \land pre \Rightarrow post[w \setminus E]$

Law *fassign*(x, E).
$w, x : [pre, post] \sqsubseteq w, x : [pre, post[x \setminus E]];\ x := E$

Law *seqComp*(mid).
$w : [pre, post] \sqsubseteq w : [pre, mid];\ w : [mid, post]$
provided *mid* and *post* have no free initial variables.

Law *seqCompIV*(x, X, mid).
$w, x : [pre, post] \sqsubseteq [\![\mathbf{con}\ X \bullet x : [pre, mid];\ w, x : [mid[x_0 \setminus X], post[x_0 \setminus X]]$
provided *mid* does not contain other initial variables beyond x_0.

Law *varInt*$(n : T)$.
$w : [pre, post] \sqsubseteq [\![\mathbf{var}\ x : T \bullet w, x : [pre, post]]\!]$
provided x does not occurs in w, *pre*, and *post*.

Law *contractFrame*(x).
$w, x : [pre, post] \sqsubseteq w : [pre, post[x_0 \setminus x]]$

Law *removeCon*(x).
$[\![\mathbf{con}\ c : T \bullet p]\!] \sqsubseteq p$
provided c does not occur in p.