ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

D. J. Duke , University of York, UK and A.S. Evans, University of Bradford, UK (Eds)

2nd BCS-FACS Northern Formal Methods Workshop

Proceedings of the 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, 14-15 July 1997

A Tool for Logic Program Refinement

R. Colvin, I. Hayes, R. Nickson and P. Strooper

Published in collaboration with the British Computer Society

A Tool for Logic Program Refinement

Robert Colvin Ian Hayes Ray Nickson Paul Strooper

Software Verification Research Centre
School of Information Technology,
University of Queensland, Brisbane, 4072, Australia

Abstract

The refinement calculus provides a method for transforming specifications to executable code, maintaining the correctness of the code with respect to its specification. In the original refinement calculus, the target language is an imperative programming language, but more recently a refinement calculus for deriving logic programs has been proposed.

Due to the amount of detail involved, the manual refinement of programs is a tedious and time-consuming task, and is therefore an obvious candidate for tool support. Several tools exist for the imperative refinement calculus, and in this paper we describe a prototype tool to support the recently developed refinement calculus for logic programs. The tool was developed using *Ergo*, an interactive theorem prover. To provide tool support for the calculus, its underlying semantic model was defined within *Ergo*, and the laws of the calculus were proven in that framework. We illustrate the tool using a simple example refinement.

1 Introduction

The refinement calculus [1, 8, 9] is a method for systematically deriving programs from formal specifications, in a way that guarantees correct implementations. It is based on :

- a *wide-spectrum language* that includes both specification and executable (code) constructs;
- a *refinement relation* between programs in the wide-spectrum language, which models the notion of correct implementation; and
- a collection of *refinement laws* providing the means to refine specifications to code in a stepwise fashion.

A refinement calculus for logic programs has been developed [6]. The wide-spectrum language includes assertions and general predicates (specification constructs), types and invariants, as well as a subset that corresponds to Horn clauses (code). The refinement relation is defined so that an implementation must produce the same set of solutions as the specification it refines, but it need do so only when the assertions and invariants hold. There are refinement laws for manipulating assertions and predicates, and for introducing code constructs.

Rigorous application of a refinement calculus can be tedious and time-consuming. Many refinement laws are conditional, so proof is needed to ensure they are correctly used. Tool support is called for, and several tools for the imperative refinement calculus exist (see [2] for a review). One such tool is PRT [3], which supports the imperative refinement calculus by managing program derivations, allowing the browsing and automatic application of refinement laws, and providing interactive and automated proof facilities for discharging the obligations of conditional laws.

This paper describes a prototype tool for the refinement of logic programs, based on the refinement calculus presented in [6]. The tool, REFLP, is interactive, with the user driving the refinement process. Like

$\langle P \rangle$	-	a specification
$\{\mathrm{A}\}$	-	an assertion
$(S \vee T)$	-	disjunction
$(S \wedge T)$	-	parallel conjunction
(S, T)	-	sequential conjunction
$(\exists X \bullet S)$	-	existential quantification
$(\forall X \bullet S)$	-	universal quantification
$pc(K)$	-	procedure call

Figure 1: Summary of wide-spectrum language

PRT, REFLP is built on *Ergo* [11], a customisable interactive theorem prover whose basic proof paradigm is window inference [5, 12]. So far REFLP has been applied to two full refinements.

Section 2 is a summary of the refinement calculus for logic programs. It provides a framework for refining programs in a wide-spectrum logic programming language. The language, its semantics, and selected refinement laws from the calculus are described. Section 3 is an overview of the *Ergo* theorem prover and of window inference. Section 4 shows how REFLP is implemented using *Ergo*. Section 5 illustrates the refinement of a simple logic program using REFLP. We discuss our experience with using REFLP and a comparison with refining 'by hand'. Section 6 summarises the paper and presents some ideas for future directions.

2 Refinement Calculus for Logic Programs

This section summarises the logic program refinement calculus (for further details, including a discussion of the relation between our semantics and the more traditional logic programming semantics, see [6]). We first present the wide-spectrum logic programming language and its semantics. We then define a notion of refinement, example refinement laws, and some simple refinement steps using the laws. In Section 4, we show how Ergo can be used to prove the refinement laws correct with respect to the semantics.

2.1 Wide-spectrum Language

The logic program development language combines both logic programming language and specification language constructs into a single wide-spectrum language that blurs the distinction between code and specifications. The new language allows constructs that may not be executable, similar to Morgan's [8] inclusion of specification constructs in Dijkstra's imperative language. This has the benefit of allowing gradual refinement without the need for notational changes during the refinement process. The constructs in the language are specifications, assertions, propositional operators, quantifiers, and procedure calls. A summary of the language is shown in Figure 1.

Specifications A specification $\langle P \rangle$, where P is a predicate, represents a set of instantiations of the free variables of the program that satisfy P. For example, the specification $\langle X = 5 \vee X = 6 \rangle$ represents the set of instantiations $\{5, 6\}$ for X. The specification $\langle false \rangle$ always computes an empty answer set: it is like Prolog's `fail`.

Assertions An assertion $\{A\}$, where A is a predicate, provides a context for a program fragment. For example, some programs may require that an integer parameter X be non-zero, expressed as $\{X \neq 0\}$. Assertions are introduced primarily to allow the assumptions a program fragment makes about the context in which it is used to be stated formally. If these assumptions do not hold, the program fragment may abort. Aborting includes program behaviour such as nontermination and abnormal termination due to exceptions like division by zero (the latter being distinct from a procedure that fails if the divisor is zero), as well as

termination with arbitrary results. We define the (worst possible) program **abort** by

$$\textbf{abort} \mathrel{\widehat{=}} \{false\}$$

Note that **abort** is quite different from the program $\langle false \rangle$, which never aborts, but always fails.

Propositional Operators There are two forms of conjunction: a sequential form (S, T) where S is evaluated before T; and a parallel version $(S \wedge T)$ where S and T are evaluated independently and the intersection of their respective results is formed on completion. The disjunction of two programs $(S \vee T)$ computes the union of the results of the two programs. Note that although the same syntax is used for the parallel conjunction and disjunction operators as logical *and* and *or* in predicate logic, they are defined on *programs*, and not predicates; intuitively the meaning is the same.

Quantifiers Disjunction is generalised to an existential quantifier $(\exists X \bullet S)$, which computes the union of the results of S for all possible values of X. Similarly, the universal quantifier $(\forall X \bullet S)$ computes the intersection of the results of S for all possible values of X. As for the propositional opertaors, note the distinction between program quantification and proposition quantification.

Types and invariants When specifying a procedure, one would like the ability to specify the types of its parameters. A procedure is only required to work correctly with respect to parameters consistent with those types. Because type constraints are expressed using specifications, it is natural to generalise them to arbitrary predicates, called *invariants*, that can impose constraints relating multiple variables. In the current version of the refinement calculus [6], types and invariants are expressed as specifications, and there are laws that allow the effect of an invariant specification to be assumed as context. In an earlier version [7] there was explicit notation for invariants in the language.

Procedures A procedure definition has the form

$$pc(F) = S.$$

It defines the procedure called pc with a list of formal parameters F and body S. The free variables of S should be a subset of the variables of F. A call on the procedure pc is of the form $pc(K)$ where K is a list of actual parameters. Procedure definition is not part of the language; instead, the definition extends the language so that the call $pc(K)$ is a command of the extended language.

Executability Some primitive procedures are considered part of the executable subset of the programming language, e.g., Herbrand equality. A program in the wide-spectrum language is executable if each of its defined procedures is a disjunction of clauses, each of which is a (possibly existentially quantified) sequential conjunction of calls on primitive and defined procedures.

2.2 Semantics

To define the semantics of the extended language we first define the effect of a program if assertions are ignored; then we define the condition under which programs are guaranteed not to abort. We give the semantics for the basic constructs first; procedures are covered in Section 2.3.

Program effect We define a function, ef, that gives the effect of a program as a characteristic predicate of the results computed by the program, ignoring assertions. The effect function for the basic constructs in our language is detailed in Figure 2(a).

No abort We define a function, ok, that defines under what circumstances a program is guaranteed not to abort. The details of ok for basic constructs are given in Figure 2(b). A specification can never abort; an assertion aborts when its predicate is false; a parallel conjunction or a disjunction aborts if either branch aborts. The sequential conjunction (S, T) aborts either if S aborts, or if S succeeds and T aborts (if S fails, T is not executed at all). Quantified programs must be ok for *all* instantiations of variables.

2.3 Procedures

Suppose we have a non-recursive procedure definition

$$pc(F) = S$$

$$
\begin{array}{rcl}
ef.\langle P\rangle & \widehat{=} & P \\
ef.\{A\} & \widehat{=} & true \\
ef.(S \vee T) & \widehat{=} & ef.S \vee ef.T \\
ef.(S \wedge T) & \widehat{=} & ef.S \wedge ef.T \\
ef.(S,\ T) & \widehat{=} & ef.S \wedge ef.T \\
ef.(\exists X \bullet S) & \widehat{=} & (\exists X \bullet ef.S) \\
ef.(\forall X \bullet S) & \widehat{=} & (\forall X \bullet ef.S)
\end{array}
$$

(a) Effects

$$
\begin{array}{rcl}
ok.\langle P\rangle & \widehat{=} & true \\
ok.\{A\} & \widehat{=} & A \\
ok.(S \vee T) & \widehat{=} & ok.S \wedge ok.T \\
ok.(S \wedge T) & \widehat{=} & ok.S \wedge ok.T \\
ok.(S,\ T) & \widehat{=} & ok.S \wedge (ef.S \Rightarrow ok.T) \\
ok.(\exists X \bullet S) & \widehat{=} & (\forall X \bullet ok.S) \\
ok.(\forall X \bullet S) & \widehat{=} & (\forall X \bullet ok.S)
\end{array}
$$

(b) Non aborting conditions

Figure 2: Semantics of program constructs

where V is the set of variables of F. A call of the form $pc(K)$ is by definition equivalent to $(\exists\ V \bullet \langle F = K\rangle, S)$. The bound variables V are renamed if necessary, to avoid capturing occurrences in K. The effect and non-aborting condition for the procedure call are therefore:

$$
\begin{array}{rcl}
ef.pc(K) & = & (\exists\ V \bullet (F = K) \wedge ef.S) \\
ok.pc(K) & = & (\forall\ V \bullet (F = K) \Rightarrow ok.S)
\end{array}
$$

Thus, a non-recursive procedure behaves like the right-hand side of its definition, with appropriate parameter substitution.

To define the semantics of recursive procedures, we use a sequence of approximations for both ef and ok, and define the actual meaning as the limit of these approximations. The details are explained in [6].

2.4 Refinement

Refinement between programs is given by reducing the circumstances under which abortion is possible — that is, by weakening ok — while maintaining the effect in those cases where abortion is not possible.

$$S \sqsubseteq T \ \widehat{=}\ ok.S \Rightarrow \left(\begin{array}{c} ok.T \\ ef.S \Leftrightarrow ef.T \end{array} \right)$$

Program equivalence ($\sqsupseteq\sqsubseteq$) is defined as refinement in both directions.

$$S \sqsupseteq\sqsubseteq T \ \widehat{=}\ (S \sqsubseteq T \wedge T \sqsubseteq S)$$

Given two procedures pa and pc we say that pa is refined by pc if for all arguments K

$$pa(K) \sqsubseteq pc(K).$$

When reasoning about calls on a procedure, the specification of the procedure can be used, since it is assured that any refinement is a valid implementation of the specification.

In the refinement of the body of a recursive procedure with specification

$$pc(F) = S$$

where F is a list of distinct variables, we may make use of recursive calls on the procedure provided that we can find a well-founded relation '$<$' such that the arguments to all of the recursive calls are less than F according to the well-founded relation. Introduction of a recursive call with actual parameter K can be achieved via the refinement

$$\{K < F\}, S\left[\tfrac{K}{F}\right] \sqsubseteq pc(K)$$

where $S\left[\frac{K}{F}\right]$ represents S with all occurrences of the variables in F replaced by the corresponding arguments in K.

2.5 Refinement Laws

To illustrate the refinement semantics, we show some refinement laws that can be proven correct with respect to the above semantics. Most laws have two parts: the part above the line represents the premisses that must be satisfied before the transformation below the line can be employed. If there are no premisses, the line is omitted and only the part below the line is shown.

Monotonicity sequential first

$$\frac{S \sqsubseteq S'}{S, U \sqsubseteq S', U}$$

This is an example of a monotonicity law. In general, a monotonicity law states that the refinement of a component of a program refines the entire program. In this case, if S refines to S' and S is the first part of a sequential conjunction, then the conjunction is refined by replacing S with S'. These laws are fundamental to the way the refinement calculus works.

Remove assertion

$$\{A\}, S \sqsubseteq S$$

An assertion may be removed from a program. The law has no premisses, so the transformation is always possible.

Distribute assertions over disjunction

$$\{A\}, (S \vee T) \sqsupseteq (\{A\}, S) \vee (\{A\}, T)$$

A group of laws describe the distributive properties of assertions. The above law states that an assertion may be distributed to both parts of a program fragment. Similar distributive laws exist for both sequential and parallel conjunction. Since this law is a refinement equivalence, it may be applied in either direction.

Equivalent under assertion

$$\frac{A \Rightarrow (P \Leftrightarrow Q)}{\{A\}, \langle P \rangle \sqsupseteq \{A\}, \langle Q \rangle}$$

This rule allows a specification P to be rewritten as the specification Q if they are equivalent under an assertion.

2.6 Example

A small example of applying refinement laws follows. Consider a (fragment of a) wide-spectrum program

$$\{Y = 5\}, (\langle X + Y = 10 \rangle \vee \langle X + Y = 20 \rangle)$$

Using the 'Distribute assertion' law presented above, this can be refined (equivalently) to

$$(\{Y = 5\}, \langle X + Y = 10 \rangle) \vee (\{Y = 5\}, \langle X + Y = 20 \rangle)$$

Now the above law 'Equivalent under assertion' can be applied once to each disjunct, to result in

$$(\{Y = 5\}, \langle X = 5 \rangle) \vee (\{Y = 5\}, \langle X = 15 \rangle)$$

Finally, the assertions may be removed by applying 'Remove assertion' twice

$$\langle X = 5 \rangle \vee \langle X = 15 \rangle$$

step no.	proof step	focus	hypotheses
0	Original focus	$A \Rightarrow (A \vee B)$	
1	Open on $A \vee B$	$A \vee B$	A
2	Open on A	A	A and $\neg B$
3	Simplify using hypothesis	$true$	A and $\neg B$
4	Close window	$true \vee B$	A
5	Simplify	$true$	A
6	Close window	$A \Rightarrow true$	
7	Simplify	$true$	

Figure 3: Example proof

3 The Ergo Theorem Prover

The *Ergo* theorem prover [11] is based on the window inference paradigm [5, 12], a proof technique based on term rewriting with access to context.

3.1 Window inference

Proof by window inference proceeds by transforming terms in a way that preserves some window relation. To prove a theorem α, one starts with α as the initial focus, and transforms it by a sequence of steps, each of which preserves the relation $\Leftarrow$ (reverse implication), into the goal $true$:

$$\alpha \Leftarrow \alpha_1 \Leftarrow \cdots \Leftarrow true$$

Because $\Leftarrow$ is a transitive relation, we have proved $true \Rightarrow \alpha$, and hence we have proved α.

During a proof, part of an expression can be *windowed* or *focused* on. This subexpression can be subsequently manipulated/simplified as required, then placed back into the original expression by *closing* the window. Sometimes windowing allows the introduction of *hypotheses*, providing a context in which a subexpression appears. For instance, focusing on the right-hand side of an implication allows the left-hand side to be used as a hypothesis, and focusing on a disjunct gives the negation of the other disjunct as a hypothesis. Window inference rules define how the focus, context and proof relation change during a proof.

For example, consider proving $A \Rightarrow (A \vee B)$ as shown in Figure 3. Step 1 is an application of a *window opening rule*: in order to transform the expression $\alpha \Rightarrow \beta$ under the relation $\Leftarrow$, one can transform β, again using $\Leftarrow$, in a context that includes α. This corresponds to the theorem:

$$\frac{\alpha \Rightarrow (\beta \Leftarrow \beta')}{(\alpha \Rightarrow \beta) \Leftarrow (\alpha \Rightarrow \beta')}$$

This theorem must be proven to show that the opening rule is valid.

The above window opening rule preserves the $\Leftarrow$ relation. Other rules can lead to a change of relation. Consider the following rule:

$$\frac{\alpha \Rightarrow \alpha'}{(\alpha \Rightarrow \beta) \Leftarrow (\alpha' \Rightarrow \beta)}$$

The rule says that, in order to transform $\alpha \Rightarrow \beta$, maintaining $\Leftarrow$ as the relation, one can transform α into α', but the relation must be $\Rightarrow$. Intuitively, to strengthen $\alpha \Rightarrow \beta$ one must weaken α; the function $\Rightarrow$ is anti-monotonic in its first argument.

Window opening in *Ergo* is denoted by a number enclosed in square brackets, as in `[1]`. This indicates that the first part of an expression is to be the new window; for instance, the first part of a binary expression would be the left operand. Opening in this way can be nested, as in `[1,2]`. This simply means the new focus will be the second part of the first part of the original focus.

3.2 Using definitions and rewriting rules

Rules, including theorems, axioms, and definitions, are applied with the *use* function. In its simplest form, *use* takes as parameter the name of a rule to be applied. Consider the theorem

```
theorem imp_and ===
  A => (B and C)
  <=>
  (A => B) and (A => C).
```

It can be used to transform an expression of the form $A \Rightarrow (B \wedge C)$ to $(A \Rightarrow B) \wedge (A \Rightarrow C)$, and vice versa. It can be applied in *Ergo* in the forward direction with the command `use(imp_and)`.

Definitions (such as *ef*, *ok*, etc. in the refinement calculus) can be applied to rewrite an expression. The command `unfold` will unfold an expression using the definition of its outermost operator.

Hypotheses can also be used to transform the current focus. A hypothesis is a predicate that can be assumed *true* in the current context, and can therefore be used in the same way as an axiom or theorem. Often there will be more than one hypothesis in the context: an example of using the second such hypothesis is `use(hyp:2)`.

Rules, hypotheses and commands like `unfold` can be combined with windowing, as in

```
[2] ---> use(imp_and).
```

This is a shorthand for opening a sub-expression, applying the rule, and finally closing the window again. When using *Ergo* interactively, a further 'shorthand' is available in that the new focus (`[2]`) can be selected by using the mouse. Subsequently the required rule can be applied through the use of mouse-driven menus. In this paper, of necessity we use the textual form to express *Ergo* commands.

Ergo provides generic facilities for automation of proof/refinement steps. The degree of automation is customisable. For example, *Ergo* can be configured to complete propositional aspects of proofs completely automatically. We have chosen not to do so in this paper, for illustrative purposes.

To structure the database of refinement and window inference rules, and to facilitate the searching and retrieval of such rules, they are organised into *theories*. Some of the theories are general-purpose theories that are useful in many applications, while others, such as the theory described in the next section for the refinement calculus for logic programs, are application-specific.

4 The Refinement Tool

In this section, we describe the components of REFLP: the operators and axioms of the *Ergo* theory for our refinement calculus, how these can be used to prove the refinement laws, and the window-inference rules.

4.1 Operators and Axioms

The syntax of the wide-spectrum language must be converted into a textual form suitable for use in *Ergo*, which involves defining the operators of the language. Figure 4 shows some examples of the syntax we use. Remember from section 2.1 that calculus operators and quantifiers must be distinguiished from standard propositional operator and quantifiers, hence they are prepended with `p`, as in `pand` for *and*.

We then define the semantics of the functions *ef* and *ok* as *axioms* in *Ergo*. For example, the axioms defining the meaning of a specification in our language are:

```
axiom def_ef_spec === ef(spec(P)) = P.
axiom def_ok_spec === ok(spec(P)) = true.
```

The axioms are given a name so that they can be referenced from within a proof. The axiom `def_ef_spec` states that the effect of a specification $\langle P \rangle$ is P, and `def_ok_spec` states that a specification never aborts.

Ergo Representation	Calculus Representation
`spec(X = 5)`	$\langle X = 5 \rangle$
`assert(B <=> C)`	$\{B \Leftrightarrow C\}$
`assert(A) sand spec(P)`	$\{A\}, \langle P \rangle$
`spec(P) pand spec(Q)`	$\langle P \rangle \wedge \langle Q \rangle$
`spec(P) por spec(Q)`	$\langle P \rangle \vee \langle Q \rangle$
`pex x spec(P)`	$\exists\, x \bullet \langle P \rangle$
`pall x spec(P)`	$\forall\, x \bullet \langle P \rangle$

Figure 4: Syntax of operators

The axioms for the other language constructs are also straightforward translations of the semantics shown in Figure 2.

Finally, we define the meaning of refinement.

```
define S refsto T  === ok(S) => ok(T) and (ef(S) <=> ef(T)).
define S refines T === T refsto S.
define S refeq T   === (S refsto T) and (T refsto S).
```

The operator `refines`, read 'is a refinement of', is simply refinement in the other direction. Again, these are straightforward translations of the definition of refinement given in Section 2.4.

4.2 Proof of Monotonicity of parallel conjunction

Once we have defined the theory of logic program refinement in *Ergo*, it is possible to formally prove refinement laws in this theory. While previously proofs were done by hand, proofs in *Ergo* are rigorous and show correctness with respect to the framework defined. Of course, the enforced rigour may slow down the proving process by forcing the application of intuitive or simplistic rules explicitly.

To illustrate how theorems are proven using window inference with *Ergo*, we show a proof of the following theorem.

$$\frac{S \sqsubseteq T}{S \wedge U \sqsubseteq T \wedge U}$$

It states that parallel conjunction can be refined by refining its first component. In REFLP syntax, the theorem has the form

```
theorem mono_pand_first ===
  (S refsto T)
  =>
  (S pand U refsto T pand U).
```

The *Ergo* commands to prove this theorem are shown in Figure 5.

The first step in the proof is to open the left-hand side of the implication (the proof obligation) (1). It has the definition of refinement applied to it (2). The focus is then

```
ok(S) => ok(T) and (ef(S) <=> ef(T))
```

Later in the proof, it becomes useful to have the implication split into `(ok(S) => ok(T)) and (ok(S) => ef(S) <=> ef(T))`. This is achieved by applying the law `imp_and` (3); Ergo could easily be configured to perform this step automatically. Once this has been done, the window is closed as no further simplification is possible (4), and the right-hand side of the implication is made the focus (5). This allows the left-hand

step	*Ergo* command	focus	context
0		`(S refsto T) =>` `  (S pand U refsto T pand U)`	
1	`[1]`	`S refsto T`	
2	`  unfold`	`ok(S) => ok(T) and` `  (ef(S) <=> ef(T))`	
3	`  use(imp_and)`	`(ok(S) => ok(T)) and` `  (ok(S) => (ef(S) <=> ef(T)))`	

4	`close`	`(ok(S) => ok(T)) and`
		`  (ok(S) => (ef(S) <=> ef(T))) =>`
		`  (S pand U refsto T pand U)`	

5	`[2]`	`S pand U refsto T pand U`	hyp:1, hyp:2

6	`  unfold`	`ok(S pand U) =>`
		`  ok(T pand U) and`
		`  (ef(S pand U) <=> ef(T pand U))`
			hyp:1, hyp:2

7	`  [1] --->`
	`  unfold`
		`ok(S) and ok(U) =>`
		`  ok(T pand U) and`
		`  (ef(S pand U) <=> ef(T pand U))`
			hyp:1, hyp:2

8	`  [2,1] --->`
	`  unfold`
		`ok(S) and ok(U) =>`
		`  (ok(T) and ok(U)) and`
		`  (ef(S pand U) <=> ef(T pand U))`
			hyp:1, hyp:2

9	`  [2,2,1] --->`
	`  unfold`
		`ok(S) and ok(U)`
		`  (ok(T) and ok(U)) and`
		`  (ef(S) and ef(U) <=>`
		`  ef(T pand U))`
			hyp:1, hyp:2

10	`  [2,2,2] --->`
	`  unfold`
		`ok(S) and ok(U) =>`
		`  (ok(T) and ok(U)) and`
		`  (ef(S) and ef(U) <=>`
		`  ef(T) and ef(U))`
			hyp:1, hyp:2

11	`  [2,1,2]`	`ok(S) and ok(U) =>`
		`  ok(T) and`
		`  (ef(S) and ef(U) <=>`
		`  ef(T) and ef(U))`
			hyp:1, hyp:2

12	`  [2,1] --->`
	`  use(hyp:1)`
		`ok(S) and ok(U) =>`
		`  (ef(S) and ef(U) <=>`
		`  ef(T) and ef(U))`
			hyp:1, hyp:2

13	`  [2,1,1] --->` `  use(hyp:2)`	*true*	hyp:1, hyp:2

Hypotheses used in context:
hyp:1 `ok(S) => ok(T)`
hyp:2 `ok(S) => (ef(S) <=> ef(T))`

Figure 5: Proof of monotonicity of parallel conjunction

side to be used as a hypothesis; because it is a conjunction, it is separated into two hypotheses, labelled hyp:1 and hyp:2.

Steps 6–10 unfold the definitions of refinement, *ef* and *ok* as required, leaving the proof in the state

```
ok(S) and ok(U) =>
  (ok(T) and ok(U)) and
  (ef(S) and ef(U) <=> ef(T) and ef(U))
```

Step 11 opens on the boxed `ok(U)`, which is automatically simplified to *true* since it is a hypothesis derived from the left-hand side of an implication. The focus is thus

```
ok(S) and ok(U) =>
  ok(T) and
  (ef(S) and ef(U) <=> ef(T) and ef(U))
```

It is clear from hypothesis 1 that the `ok(T)` on the right-hand side can also be simplified to *true*, since when proving the right-hand side `ok(S)` may be assumed. Thus, hyp:1 is *use*d (12) leaving the proof in the state

```
ok(S) and ok(U) =>
     (ef(S) and ef(U) <=> ef(T) and ef(U))
```

Hypothesis 2 shows that `ef(S) <=> ef(T)` under the assumption `ok(S)`. It can therefore be applied to `ef(S)`, replacing it with `ef(T)` (13), which simplifies the equivalence to *true*, thereby completing the proof.

4.3 Window Inference Rules

Refinement, denoted by $\sqsubseteq$, is a reflexive and transitive relation (a *preorder*). Hence it can be incorporated into the window inference paradigm, and used in a similar way to implication in proofs.

Opening laws exist for the predicate calculus, such as opening on either side of an implication or conjunction. Similar laws are needed for the refinement operators, to allow opening on disjuncts, either side of a refinement, specification and assertion predicates, etc. Before these opening rules can be created, it must be shown that each window in a proof by window inference must be related to the original window by the preorder relation. This means refinement (and refinement equivalence) must be shown to be transitive and reflexive (and symmetric). Reflexivity is required so that this property holds initially; in the case of refinement, the current program (initially S) must be a refinement of the original program (S). Transitivity is required so that successive refinements refine the original specification. Symmetry is only used for equivalence relations, such as refinement equivalence ($\sqsupseteq$ or `refeq`), since this property does not hold for refinement ($\sqsubseteq$ or `refsto`). Symmetry allows a rule to be used in either direction. In addition, since $\sqsupseteq$ is stronger than $\sqsubseteq$, refinement equivalence can be used to show refinement.

For example, the program $\langle X = 1 \Leftrightarrow 1 \rangle \wedge \langle Y = 5 \rangle$ can be refined (equivalently) to $\langle X = 0 \rangle \wedge \langle Y = 5 \rangle$. An opening rule would allow the specification $\langle X = 1 \Leftrightarrow 1 \rangle$ to be focused on and refined separately to the rest of the program. The following window inference rule permits such a step.

```
openrule(S pand T, [1], [], [refsto ---> refsto,
                             refines ---> refines,
                             refeq ---> refeq]).
```

The parameters to the `openrule` have the following meaning.

- The first parameter indicates the expression to which the rule can be applied.
- The second parameter indicates which part of the expression is to become the focus of the new window. In this case, the first part of the expression, S, will be the focus of a new window.

- The third parameter indicates new hypotheses that are added to the context when using this opening rule. In this case there are none (an empty list).
- The last parameter guides the relation of the proof. The relation will be preserved, so long as it is one of `refsto`, `refines`, or `refeq`.

The above window inference rule embodies the following three refinement laws.

$$\frac{S \sqsubseteq S'}{S \wedge U \sqsubseteq S' \wedge U} \qquad \frac{S \sqsupseteq S'}{S \wedge U \sqsupseteq S' \wedge U} \qquad \frac{S \sqsupseteq\sqsubseteq S'}{S \wedge U \sqsupseteq\sqsubseteq S' \wedge U}$$

These laws (which are monotonicity properties) must be proven to show the validity of the window inference rule.

Windowing on the second part of a sequential conjunction permits some context information. For instance, consider the program

$$\{isordered(T)\}, \langle Y \in members(T)\rangle$$

The user wishes to refine the specification $\langle Y \in members(T)\rangle$. When this becomes the new focus, the user should be able to use the hypothesis $isordered(T)$ while refining. This is achieved by the opening rule

```
openrule(assert(A) sand T, [2], [A], [refsto ---> refsto,
                                      refines ---> refines,
                                      refeq ---> refeq]).
```

The second parameter to `openrule` reflects the windowing on T; the third parameter indicates that A becomes a hypothesis. The last parameter is the same as in the previous example. The refinement laws that correspond to this window rule are:

$$\frac{A \Rightarrow (T \sqsubseteq T')}{\{A\}, T \sqsubseteq \{A\}, T'} \qquad \frac{A \Rightarrow (T \sqsupseteq T')}{\{A\}, T \sqsupseteq \{A\}, T'} \qquad \frac{A \Rightarrow (T \sqsupseteq\sqsubseteq T')}{\{A\}, T \sqsupseteq\sqsubseteq \{A\}, T'}$$

These are generalisations of the law 'Equivalent under assertion' (Section 2), and use of the window rule avoids the need for explicit reference to that law.

The previous opening rules have maintained the refinement relation (the fourth parameter of the `openrule` command). However, when an assertion or specification is refined, the relation changes from a refinement relation to a logical relation: the manipulation of assertions or specifications happens at the predicate calculus level. For example, to refine the program $\langle X = 1 \Leftrightarrow 1\rangle$, the predicate $X = 1 \Leftrightarrow 1$ becomes the focus: this can be rewritten as $X = 0$ under the logical relation $\Leftrightarrow$.

```
openrule(spec(P), [1], [], [refsto ---> <=>,
                            refines ---> <=>,
                            refeq ---> <=>]).
```

The corresponding refinement laws are:

$$\frac{P \Leftrightarrow P'}{\langle P\rangle \sqsubseteq \langle P'\rangle} \qquad \frac{P \Leftrightarrow P'}{\langle P\rangle \sqsupseteq \langle P'\rangle} \qquad \frac{P \Leftrightarrow P'}{\langle P\rangle \sqsupseteq\sqsubseteq \langle P'\rangle}$$

5 Case Study

To demonstrate the use of REFLP, we present part of the refinement of a procedure to test membership of a tree. This example is taken from [6], where it was refined 'by hand'. The example is too large to present in full, so two extracts from the refinement process are shown. These extracts were chosen to demonstrate the differences when refining by hand and when using REFLP.

For any refinement, there will be some application-specific axioms required before the refinement can begin. The declarations for the example program used in the refinement fragments are shown below, followed by a description of the refinement process.

5.1 Declarations

Consider the procedure *isin*, which determines the elements in a tree (or tests whether an element is in a tree). The tree is strictly ordered. Its specification can be written in the wide-spectrum language as follows.

$$isin(Y, T) = \{isordered(T)\}, \langle Y \in members(T)\rangle$$

This can be read as '$isin(Y, T)$ is the program that, assuming T is an ordered tree, establishes $Y \in members(T)$'. To complete the specification we need the definitions of *isordered* and *members*.

$$\begin{array}{l} members(empty) \mathrel{\widehat{=}} \{\} \\ members(tree(L, X, R)) \mathrel{\widehat{=}} members(L) \cup \{X\} \cup members(R) \\ isordered(T) \mathrel{\widehat{=}} (T = empty) \vee \\ \quad (\exists\, L, X, R \bullet T = tree(L, X, R) \wedge \\ \qquad isordered(L) \wedge isordered(R) \wedge \\ \qquad (\forall\, Y : members(L) \bullet Y < X) \wedge \\ \qquad (\forall\, Y : members(R) \bullet X < Y)) \end{array}$$

Operators such as $\cup, \in, <, \wedge$, etc. are already defined in *Ergo*. However, the operators *isin*, *members*, and *isordered* must be defined; this is done by using axioms as shown below for *isordered* and *members*. They are straightforward syntactic translations.

```
axiom mem_empty ===
        members(empty) = empty_set.

axiom mem_tree ===
        members(tree(L,X,R)) = members(L) union set(X) union members(R).

axiom isord_tree ===
        isordered(T) =
        ((T = empty) or
            (ex [L] ex [X] ex [R]
                  ((T = tree(L, X, R)) and
                  isordered(L) and isordered(R) and
                  (all [Y] (Y in members(L) => Y < X)) and
                  (all [Y] (Y in members(R) => Y > X))))).
```

Finally, the refinement problem can be stated and the refinement process can begin. It is declared as a *theorem* to be proven.

```
theorem isinref ===
      assert(isordered(T)) sand spec(Y in members(T)) refsto _.
```

The underscore on the right-hand side of the refinement is an anonymous variable, corresponding to any program that refines the left-hand side. Using an anonymous variable in this way gives the idea of a constructive refinement process, without a predefined goal. When the target program is known, the right-hand side is instantiated to this program. After the completion of the proof this refinement theorem is saved, perhaps for use as a procedure in a larger program.

5.2 The Refinement

This section looks at two fragments of the refinement of the program *isin*, making a comparison between the process followed in [6] and that followed while using REFLP. The first fragment is the first step of the refinement 'by hand'. It shows that by using a window inference approach some of the laws can be simplified, reducing the details of refinement. The second fragment is from later in the refinement and demonstrates the benefits of representing assertions as hypotheses in REFLP. The refinement is shown in proof script syntax, introduced in Section 3.

First fragment

The original program is

$$\{isordered(T)\}, \langle Y \in members(T)\rangle$$

This can be refined to

$$\{isordered(T)\},$$
$$(\langle T = empty \vee (\exists L, X, R \bullet T = tree(L, X, R))\rangle \wedge \langle Y \in members(T)\rangle)$$

by using the law 'augment weakened assertion', whose definition is

$$\frac{A \Rightarrow B}{\{A\}, S \sqcap\!\sqcap \{A\}, (\langle B\rangle \wedge S)}$$

This law allows a new specification to be added as a parallel conjunct to an already existing program, if the specification follows from an assertion. In this case, the specification $\langle T = empty \vee (\exists L, X, R \bullet T = tree(L, X, R))\rangle$ follows from $\{isordered(T)\}$ (see the definition of *isordered* above). The obligation is not formally discharged in [6], and it is left to the refiner to ensure the application of the law is valid.

This law can be modified when a window inference approach is taken. The assertion $\{A\}$ is on the left of a sequential conjunction; we may assume it in the program S. As was shown in Section 4.3, when opening on S, A becomes part of the context by the open rule definition. Using this idea of A being a hypothesis, we get the equivalent law

$$\frac{A \vdash (A \Rightarrow B)}{A \vdash S \sqcap\!\sqcap \langle B\rangle \wedge S}$$

The obligation can be simplified to B under the hypothesis A. Since the hypothesis of the conclusion, A, is the same as the hypothesis of the premiss, and is not used elsewhere, it can be left implicit. This is because it can be assumed as part of the context. This leaves the rule

$$\frac{B}{S \sqcap\!\sqcap \langle B\rangle \wedge S}$$

As can be seen, the obligation is much simpler and therefore easier to discharge. This is particularly useful when using REFLP, as REFLP forces all obligations to be formally proven. Its definition in *Ergo* syntax is

```
theorem augment_weakened_assertion ===
    B =>
    S refeq (spec(B) pand S).
```

Returning to the refinement in REFLP, we start with the expression

```
assert(isordered(T)) sand spec(Y in members(T)) refsto _.
```

```
[1, 2].
    hyp:1.
      use(isord_tree).
      close.
    use(augment_weakened_assertion, c(hyp:2)).
```

Figure 6: Proof script for the first refinement fragment.

The proof script for this fragment is shown in Figure 6. The command `[1, 2]` focuses on `spec(Y in members(T))`, and gives `isordered(T)` as a hypothesis (due to the definition of the openrule being used). This hypothesis is not of use as it stands, so it is modified by using the definition of *isordered*. This is done by making the hypothesis the focus with `hyp:1`, applying the definition (`use(isord_tree)`) and closing. The focus returns to `spec(Y in members(T))`, however there are now two hypotheses, namely:

```
hyp:1: isordered(T)
hyp:2: T = empty or
         ex [L] ex [X] ex [R]
               (T = tree(L, X, R) and
                 isordered(L) and isordered(R) and
                     all [Y] (Y in members(L) => Y < X)
                       and
                     all [Y] (Y in members(R) => X < Y) )
```

Now we can use the theorem `augment_weakened_assertion` as described above, using the second hypothesis as the new specification. Note that hypothesis 2 does not exactly match the added specification in [6]. It is possible to manipulate the hypothesis to reduce it to just `(T = empty) or (ex [L] ex [X] ex [R] T = tree(L, X, R))`, since the other conjuncts are *true* from the context. For simplicity this is not shown here.

The *Ergo* syntax to apply this law with hypothesis 2 as the new specification is `use(augment_weakened_assertion, c(hyp:2))`. The `c(hyp:2)` constraint causes the obligation to be discharged using the existing hypothesis 2. The hypothesis is instantiated into the program and the focus is as required.

```
    spec(T = empty or
             ex [L] ex [X] ex [R]
                   (T = tree(L, X, R) and
                     isordered(L) and isordered(R) and
                         all [Y] (Y in members(L) => Y < X) and
                          all [Y] (Y in members(R) => X < Y)))
         pand
        spec(Y in members(T))
```

Second fragment

We now skip several steps. The focus 'by hand' is

$$\{isordered(T)\},$$
$$(\exists\, L, X, R \bullet \langle T = tree(L, X, R)\rangle,$$
$$\quad \{T = tree(L, X, R) \wedge isordered(T)\},$$
$$\quad \langle Y \in members(L) \vee Y = X \vee Y \in members(R)\rangle)$$

```
[2, 2, 2, 2].
        use(separate_specs_with_or).
        [1] ---> use(separate_specs_with_or).
close.
```

Figure 7: Proof script for the second refinement fragment.

In summary, the case where T is *empty* has been removed (since $isin(Y, empty)$ will always fail) and the definition of members has been applied to the original specification. The assertion $\{T = tree(L, X, R) \wedge isordered(T)\}$ was introduced in the interim to help later in the refinement.

The next step is to split the specification $\langle Y \in members(L) \vee ... \rangle$ into a disjunction of three specifications, and to distribute the assertion over each. This requires the application of four laws when refining by hand, including 'remove assertion' and two uses of 'distribute assertion over disjunction' (c.f. Section 2.5). The fourth law separates a specification whose defining predicate is a disjunction, i.e.

$$\langle A \vee B \rangle \sqsupseteq \langle A \rangle \vee \langle B \rangle$$

Together these result in

$$\begin{array}{l}
\{isordered(T)\}, \\
(\exists\, L, X, R \bullet \langle T = tree(L, X, R) \rangle, \\
\quad (\{T = tree(L, X, R) \wedge isordered(T)\}, \langle Y \in members(L) \rangle \\
\quad \vee \{T = tree(L, X, R) \wedge isordered(T)\}, \langle Y = X \rangle \\
\quad \vee \{T = tree(L, X, R) \wedge isordered(T)\}, \langle Y \in members(R) \rangle))
\end{array}$$

While the application of these laws is trivial, passing assertions around clutter the refinement. As discussed above during the first fragment, there is no need for these assertions to be passed around in REFLP, since they follow from the original assertion $\{isordered(T)\}$. Thus any time a sub-program becomes the focus it will automatically have access to these assertions in the form of hypotheses, as determined by the opening rules.

Returning to the refinement in REFLP, the focus is

```
assert(isordered(T)) sand
pex [L] pex [X] pex [R] spec(T = tree(L, X, R)) sand
             spec(Y in members(L) or Y = X or Y in members(R))
```

The proof script fragment to perform this refinement is given in Figure 7. The opening of `[2, 2, 2, 2]` focuses on

```
spec(Y in members(L) or Y = X or Y in members(R))
```

Note that this windowing is assumed when refining by hand. Now the specification can be separated into two by `use(separate_specs_with_or)`. (This REFLP theorem is a straight forward syntactic translation of the law presented above.)

```
spec(Y in members(L) or Y = X) por spec(Y in members(R))
```

The first specification is separated once more:

```
spec(Y in members(L)) por spec(Y = X) por spec(Y in members(R))
```

This window is then closed, and the focus is as required.

```
assert(isordered(T)) sand
pex [L] pex [X] pex [R] spec(T = tree(L, X, R)) sand
            (spec(Y in members(L))
         por spec(Y = X)
         por spec(Y in members(R)))
```

For readability purposes, the opening and closing in the above example was not combined with the rule applications. Taking this into account, using REFLP only two rule applications were required as opposed to four mentioned in [6]. This reduction of tedium speeds up the process and reduces the complexity of the refinement, placing the load on REFLP to manage and maintain the hypotheses correctly. Keeping hypotheses separate also makes the focus more concise and therefore easier to read.

5.3 Result

The remaining steps are not examined here, but the final result is

```
pex [L] pex [X] pex [R] (spec(T = tree(L,X,R)) sand spec(Y < X) sand isin(Y, L)
     por
pex [L] pex [X] pex [R] (spec(T = tree(L,X,R)) sand spec(Y = X)
     por
pex [L] pex [X] pex [R] (spec(T = tree(L,X,R)) sand spec(X < Y) sand isin(Y, R)
```

This corresponds to the executable form described in Section 2.1. Although not part of REFLP at this stage, the translation from such constructs to logic code is systematic [4]. Code derived from this program is

```
isin(Y, tree(L,X,R)) :- Y < X,  isin(Y,L).
isin(Y, tree(L,Y,R)).
isin(Y, tree(L,X,R)) :- Y > X,  isin(Y,R).
```

5.4 Discussion

The most important differences between refinement by hand and using REFLP are that refinement by hand is error-prone and tedious, whereas refinement with REFLP is rigorous and REFLP takes care of many of the tedious details. However, refinement with REFLP requires more steps because even the most trivial and obvious results must be verified. We briefly expand on these points with the experience we have gained with the refinement calculus for logic programs.

The refinement of `isin` was originally done by hand [6]. In the final version of this refinement there are 13 steps and in these 13 steps there are 44 applications of refinement laws. What constitutes a step is something that is decided by the person doing the refinement and typically involves the application of several refinement laws. However, earlier versions of the refinement contained several errors and we discovered that we were using more refinement laws in each step than we originally thought.

For comparison, Table 1 summarises the number of refinement steps for refinement by hand and using REFLP for `isin`. For the refinement with REFLP, a step is either a rule application, or opening/closing a window, i.e., one line from the examples above. At first sight, the difference in the number of steps between refinement by hand (44) and refinement with REFLP (80) appears daunting. However, many of these steps (31) involve opening and closing windows, which is achieved by a simple click of a mouse button. In fact, most of the REFLP commands are executed using a mouse and are thus not as tedious as the *Ergo* scripts in this paper suggest. In addition, 18 steps are required to discharge proof obligations, which in this example are straightforward and therefore were not done in the refinement by hand. Finally, 8 steps involve the application of case-specific axioms such as the definitions of *isordered* and *members*. This leaves 23 steps that actually involve the application of a refinement law, which compares favourably with the 36

	steps
refinement by hand (total)	44
case-specific definitions	8
refinement laws	36
refinement with REFLP (total)	80
opening/closing rules	31
proof obligations	18
case-specific axioms	8
refinement laws	23

Table 1: Number of refinement steps for `isin` refinement

law applications in the refinement by hand. Part of the difference stems from the fact that in REFLP some of the refinement laws are encoded as window inference rules.

Another advantage of REFLP is its hypothesis management. In the refinement by hand, the use of assertions leads to cluttered program fragments and the application of many laws that distribute and manipulate assertions. REFLP maintains assertions implicitly as hypotheses, and allows them to be accessed when needed (as shown in the first step of the `isin` refinement).

6 Concluding Remarks

In this paper, we have presented REFLP, a refinement tool for logic programs that is based on a recently developed logic programming refinement calculus. The syntax and semantics of the refinement calculus have been embedded in the tool and many refinement laws have been proven correct with respect to this semantics. Since REFLP is based on the *Ergo* theorem prover, it uses window inference and the refinement laws have been encoded as window inference rules.

So far, REFLP has been applied to two examples. Experience with these examples has shown that although REFLP requires more steps than a typical refinement by hand, most of these steps are straightforward and in fact fewer applications of refinement laws are needed. The major advantages of REFLP are that it keeps track of most of the tedious detail involved in refinement, and that it is rigorous so that the results of the refinement are guaranteed within the framework presented.

An area for future work is the comparison of the traditional refinement calculus to the logic refinement calculus; for example, by comparing REFLP to a similar tool supporting traditional refinement. One of the motivations for the refinement calculus for logic programs is that in logic programming there is a smaller conceptual gap between specification and implementation, and that hence the refinement process is simplified. We have already found that the development of REFLP was easier than the development of PRT, a similar tool for the traditional refinement calculus. REFLP was built on top of *Ergo* in a straightforward manner, whereas for PRT sophisticated mechanisms were needed to support state-dependent reasoning [10].

In the long term, the *Ergo* interface must be customised to suit the refinement calculus if it is to become feasible to use REFLP for larger examples. To increase the tool's usability, a library of theorems supporting common programming constructs such as integers, lists, etc. must be built. Also, work has been done on developing tactics that automatically discharge proof obligations, to reduce the amount of interaction needed. In the short term, more case studies are required, as it is certain that there are more theorems, opening rules, etc. which will be consolidated into refinement theories for general use. The tool is also lacking general definitions for procedure calls and recursion, which have been incorporated into the calculus since the commencement of the project. The calculus is new, and any subsequent additions to it must also be captured by the tool.

References

[1] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[2] David Carrington, Ian Hayes, and Ray Nickson et al. A review of existing refinement tools. Technical Report 94-8, Software Verification Research Centre, The University of Queensland, 1994.

[3] David Carrington, Ian Hayes, and Ray Nickson et al. A tool for developing correct programs by refinement. In He Jifeng, editor, *Proc. BCS 7th Refinement Workshop, Bath, UK*, Electronic Workshops in Computing, pages 1–17. Springer, 1996. URL http://www.springer.co.uk/eWiC/Workshops/7RW.html. Also available as Technical Report UQ-SVRC-95-49, Software Verification Research Centre, University of Queensland.

[4] Yves Deville. *Logic Programming: Systematic Program Development*. Addison Wesley, 1994.

[5] Jim Grundy. A window inference tool for refinement. In Cliff B. Jones, Roger C. Shaw, and Tim Denvir, editors, *Fifth Refinement Workshop*, Workshops in Computing, pages 230–254. BCS FACS, Springer-Verlag, 1992.

[6] Ian Hayes, Ray Nickson, and Paul Strooper. Refining specifications to logic programs. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of the 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 1996*, volume 1207 of *Lecture Notes in Computer Science*, pages 1–19. Springer Verlag, 1997.

[7] Ian Hayes and Paul Strooper. Refining specifications to logic programs. In I.J. Hayes, editor, *Proc. 5th Australasian Refinement Workshop*, pages 1–13. Software Verification Research Centre, The University of Queensland, April 1996. The proceedings are available electronically via the world wide web URL http://www.cs.uq.edu.au/conferences/arw96/home.html.

[8] Carroll Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

[9] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.

[10] Ray Nickson and Ian Hayes. Supporting contexts in program refinement. Technical Report 96-29, Software Verification Research Centre, Department of Computer Science, The University of Queensland, 1996. Accepted for publication in Science of Computer Programming.

[11] Ray Nickson, Owen Traynor, and Mark Utting. Cogito Ergo Sum: Providing structured theorem prover support for specification formalisms. In Kotagiri Ramamohanarao, editor, *Proceedings of the Nineteenth Australasian Computer Science Conference (ACSC'96)*, volume 18(1) of *Australian Computer Science Communications*, pages 149–158, 1996.

[12] Peter Robinson and John Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1):47–61, 1993.