

**SOFTWARE VERIFICATION RESEARCH CENTRE**  
**DEPARTMENT OF COMPUTER SCIENCE**  
**THE UNIVERSITY OF QUEENSLAND**

Queensland 4072  
Australia

**TECHNICAL REPORT**

No. 94-44

**Refinement in Ergo**

David Carrington, Ian Hayes,  
Ray Nickson, Geoffrey Watson  
and Jim Welsh

July 1995

Phone: +61 7 365 1003

Fax: +61 7 365 1533

**Note:** Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

# Refinement in Ergo

David Carrington      Ian Hayes      Ray Nickson  
Geoffrey Watson      Jim Welsh

## Abstract

Refinement is a mathematically-based technique for developing a program from an abstract specification so that the program satisfies the specification. The aim of the Program Refinement Tool project is to develop a generic refinement tool suitable for supporting a methodology for the interactive development of programs based on the refinement calculus. This report summarizes our investigation into how the *Ergo* theorem prover can be used to model the refinement calculus and form the basis of this tool.

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Ergo</b>	<b>4</b>
2.1	Theory Hierarchy . . . . .	4
2.2	Constants and Functions . . . . .	5
2.3	Types . . . . .	7
2.4	Rewrite rules . . . . .	7
2.5	Window inference . . . . .	7
2.6	Tactics and Heuristics . . . . .	9

<b>3</b>	<b>Modelling the Refinement Calculus</b>	<b>9</b>
3.1	Higher Order Logic . . . . .	10
3.2	Lattice Theory of Predicates . . . . .	11
3.3	Lattice Theory of Commands . . . . .	13
3.4	Modelling States . . . . .	13
<b>4</b>	<b>Representing Refinement in Ergo</b>	<b>16</b>
4.1	Types . . . . .	18
4.2	Expressions and Predicates . . . . .	19
4.3	Commands . . . . .	20
4.4	Weakest Preconditions . . . . .	22
4.5	Refinement . . . . .	23
4.6	Denoting Identifiers . . . . .	26
4.7	Handling Modality . . . . .	31
4.8	Tactics and Heuristics . . . . .	34
<b>5</b>	<b>Example: Euclid's Algorithm</b>	<b>36</b>
5.1	Specification . . . . .	37
5.2	Refinement: New Local Variable . . . . .	37
5.3	Refinement: Split Specification . . . . .	39
5.4	Refinement: Loop Initialization . . . . .	39
5.5	Refinement: Introduce DO . . . . .	40
<b>6</b>	<b>Conclusions</b>	<b>42</b>

# 1 Introduction

Refinement is a mathematically-based technique for developing a program from an abstract specification so that the program satisfies the specification. The refinement approach starts with a formal, abstract specification of the required product and transforms this via a sequence of small steps into a form suitable for execution.

The refinement calculus, developed by Back, Morgan and Morris, [Bac78, Bac88, MR87, Mor94, Mor87] provides a uniform method for deriving programs from specifications. The calculus extends a programming language with abstract specification constructs, to give a *wide-spectrum language* [B<sup>+</sup>85, B<sup>+</sup>87] that can express specifications, programs, and every step in between. The semantics of the wide-spectrum language is defined using Dijkstra's weakest preconditions [Dij76]. The weakest precondition is also used to define an ordering ( $\sqsubseteq$ ) on constructs in the wide-spectrum language. If *Spec* is a specification and *Prog* is a program, then  $Spec \sqsubseteq Prog$  if and only if *Prog* is an implementation of *Spec*. More generally, the ordering allows one specification or command (*S*) to be replaced by another (*T*): correctness is maintained as long as  $S \sqsubseteq T$ . Typically, *T* is derived from *S* by applying an instance of a *refinement rule*, which guarantees the appropriate relationship between *S* and *T*, subject to the validity of some *applicability condition*.

The aim of this ARC-funded project, based in the SVRC at the University of Queensland, is to develop a generic refinement tool suitable for supporting a methodology for the interactive development of programs based on the refinement calculus. A particular focus of the project is the close coupling of a generic language-based editor, instantiated to support the refinement calculus, with a back-end tool used to carry out refinement steps and generate and discharge the proof obligations that arise with each rule application. We have selected Ergo [UW94] as an appropriate tool to use for these tasks; this choice is justified in [CHN<sup>+</sup>94a].

This report summarizes our investigation into how Ergo can be used to model the refinement calculus and form the basis of our refinement tool. Section 2 gives an overview of the Ergo theorem prover. Sections 3 and 4 discuss how refinement can be modelled in higher order logic, and how we have chosen to realize part of that model in Ergo. In Section 5 we give an example of the derivation of a simple program in the refinement calculus using our Ergo theories.

## 2 Ergo

We are using Ergo as both the transformation engine and the proof editor in our project. The version of Ergo referred to in this document is release 4.0, based on Qu-Prolog 3.2. New releases of Ergo (4.1) and Qu-Prolog (4.0) are imminent as this document is being written. These new versions will support some of the things (higher-order syntax; typed quantifications; theory interpretation) that are mentioned as lacking in this report.

Ergo [RW93, UW94] is the Software Verification Research Centre's theorem prover. It is implemented in Qu-Prolog [RC93]. Proofs are done by rewriting, using *window inference* [RS93] to manage contexts. Rewrite rules are deduced from *axioms*, *theorems* and *definitions*, which are organized into theories. Users can prove their own theorems from existing axioms and theorems, or can add theories that define new notation and give axioms or definitions to suitably constrain its interpretations.

Ergo was designed with an intention to support a modal logic known as *functional logic* [SRH94]. Ergo's base theory is a subset of intuitionistic logic upon which classical, constructive and modal logics can be built. In particular, Ergo does not depend on classical equality: *substitution of equals for equals* is not essential, though Ergo provides easy mechanisms for making use of such substitution in classical theories. This makes it convenient for us to build a model of the refinement calculus where states are not represented explicitly, but are *implicit parameters* to every predicate and expression (see Section 4.7).

### 2.1 Theory Hierarchy

Ergo's extensible logic is partitioned into manageable units called *theories*. The theories are organized into a hierarchy (actually a rooted, directed acyclic graph): theories in the hierarchy can inherit theories closer to the root, and use terms of those inherited *ancestor* theories to define new concepts.

The root theory of Ergo is called `ergo`: this theory defines the most basic connectives that are needed to support Ergo's built-in inference system. It includes declarations of symbols for logical equivalence ( $\Leftrightarrow$ ) and implication ( $\Rightarrow$ ), together with theorems stating the reflexivity and transitivity of

these relations. It also includes a weak definition of equality ( $=$ ); the only significant properties are those of an equivalence relation.

This root theory is used to build a variety of intuitionistic, modal and classical logics, a classical equality relation, arithmetic and set theory. As far as possible, concepts are separated; so, it is possible to use the arithmetic and set theories with classical predicate logic and equality (to provide a ‘traditional’ first-order reasoning environment); alternatively, they can be used with functional logic (in which classical equality does not hold) to provide a functional set theory. We shall define our own (modal) logic of weakest preconditions on top of a minimal intuitionistic base, then combine it with arithmetic and set theory (but not classical equality) to provide a foundation for our theory of refinement.

## 2.2 Constants and Functions

Ergo theories contain declarations of the objects that the theory introduces, along with definitions, axioms and theorems about them.

A *constant* declaration reserves a name to stand for some object in the universe of discourse:

```
constant c.
```

That object might be a simple or complex object: for example, it might stand for an integer value, or in a higher-order theory, it might stand for a function. The value can be made explicit by coupling the declaration with a definition:

```
constant two === 1+1.
```

This declares the constant `two`, and defines the value to be the same as that denoted by the term `1+1` (where `1` and `+` are already defined in this or some ancestor theory). There is no assumption that distinctly-named constants denote distinct values.

A *function* declaration similarly reserves a name to stand for a function, which denotes an object when applied to some (fixed) number of arguments:

```
function f(A,B,C).
function minus(X,Y) === X + -Y.
```

The first declaration reserves `f` as the name of a 3-ary function; the second reserves `minus` as the name of a 2-ary function and defines `minus(X,Y)` to take the same value as `X + -Y`, for any terms `X` and `Y`. A precedence and associativity can be provided so that the function can be written in prefix, infix or postfix form.

After the declarations:

```
function 400 rightassoc '*'(X, Y). % multiplication
function 200 postfix '!'(X).      % factorial
```

the terms `1*1` and `6!` are legal.

An *object-variable* declaration is used to reserve a name for a variable that ranges over objects in the universe of discourse:

```
obj_var(x).
```

The declaration reserves `x` and all instances of `x` subscripted by numbers (`x`, `x_0`, `x_1`, etc.) as object variable names. Object variables are used as the bound variables in quantified terms (below) and as the domain of object-level substitutions, such as `[E/x]*T` which stands for the result of replacing all instances of the object variable `x` in the term `T` by `E`.

A *quantifier* declaration is used to reserve a name for a quantifier, and perhaps provide a definition for it:

```
quantifier(all).
quantifier(ex x P) === not (all x (not P)).
```

The bound variable in a quantifier declaration or an instance of its use must be an object variable.

## 2.3 Types

Ergo is untyped, and does not distinguish terms from formulas. Type theories can be defined axiomatically (or in terms of some more primitive theory, such as set theory) when they are needed. There are conventions for representing type constraints for which Ergo provides support when used with a suitable theory. The two-place functor `:` (colon) is declared (without definition) in Ergo's root theory, and is intended to be used in constructs like `X:T` to mean 'X is of type T'. This notation is supported in constant and function declarations:

```
constant c:type.  
function f(X:t1, Y:t2):t3.
```

The choice of terms to denote types themselves is entirely the user's, and Ergo's inference mechanism does not make any use of type constraints.

## 2.4 Rewrite rules

An axiom or theorem has the general form:

```
Antecedents  $\Rightarrow$   
( Input Reln Output ) :-  
Side Condition.
```

This is interpreted as a conditional rewrite rule that transforms `Input` into `Output` according to the relation `Reln` providing the `Side Condition` (a syntactic constraint on the rest of the rule) holds and the `Antecedents` can be proved. Axioms and theorems are made to fit this pattern in several ways; '`true  $\Rightarrow$` ' can be added to the front of a rule to create the `Antecedents`, '`= true`' can be added to the end to create the `Reln` and `Output`, and an empty `Side Condition` is added to a rule that has none.

## 2.5 Window inference

Window inference is used to manage the proof. To prove a theorem  $T$ , one begins with the obligation  $T \Leftarrow true$ ; this is the *initial window*. Each

window has a *focus*, a *relation*, a list of *hypotheses* and perhaps a *goal*. The initial window has focus  $T$ , relation  $\Leftarrow$ , no hypotheses and goal *true*. The relation constrains the operations that can be done to the focus; the hypotheses (which represent the context) are used to discharge the antecedents of rules, and the goal gives heuristic information about the intended result of transforming the focus.

At any time, the focus of a window can be transformed by applying a rule whose:

- **Input** matches the focus;
- **Side Condition** holds;
- **Reln** is (at least as strong as) the window's relation;
- **Antecedents** can be discharged (perhaps recursively) by appealing to the window's hypotheses.

The appropriately instantiated **Output** becomes the new focus.

A new window can be *opened* by selecting some subterm of the old focus to be the focus of the new window; an *opening rule* gives a mechanism for computing the relation, hypotheses and goal of the new window. When the new window is eventually closed, its focus replaces the subterm that was originally selected in the outer window. For example, if the original focus is  $A \Rightarrow B$ , the hypotheses are  $\Gamma$  and the new focus is  $B$ , the window relation and goal are unchanged and  $A$  is added as an hypothesis. The opening rule used here is:

$$\frac{A, \Gamma \vdash B \text{ R } B'}{\Gamma \vdash (A \Rightarrow B) \text{ R } (A \Rightarrow B')} \quad \text{R is one of } \Rightarrow, \Leftarrow, \Leftrightarrow$$

If the new focus is to be  $A$ , the relation is reversed (so  $\Rightarrow$  becomes  $\Leftarrow$ ,  $\Leftarrow$  becomes  $\Rightarrow$ , and  $\Leftrightarrow$  remains  $\Leftrightarrow$ ), the goal is negated and  $\neg B$  is added as an hypothesis; the opening rule is:

$$\frac{\neg B, \Gamma \vdash A \text{ R } A'}{\Gamma \vdash (A \Rightarrow B) \text{ R}' (A' \Rightarrow B)} \quad \text{R and R' are related as follows: } [\Rightarrow \longrightarrow \Leftarrow, \Leftarrow \longrightarrow \Rightarrow, \Leftrightarrow \longrightarrow \Leftrightarrow].$$

The opening rules are essentially monotonicity axioms about the connectives concerned; the rules just explained are valid because  $\Rightarrow$  is monotonic in its second argument and anti-monotonic in its first.

As well as opening rules, a theory contains **preorder** and **equivalence** declarations, identifying certain binary function symbols as suitable window relations. There are also declarations showing relationships between different window relations (for example, that  $\Leftrightarrow$  is a stronger relation than  $\Rightarrow$ , so transformation rules about  $\Leftrightarrow$  can be used whenever  $\Rightarrow$  is needed).

## 2.6 Tactics and Heuristics

Ergo allows the user to associate a collection of *tactics* with a theory. A tactic is a procedure written in a high-level programming language (Qu-Prolog) that is able to contribute many steps to a proof. Tactics can execute proof commands to apply transformation rules, open and close windows, manipulate hypotheses, etc. The correctness of a proof involving tactics is ensured because tactics have no direct access to the proof data structures: they can manipulate proofs only via the same commands that human users use.

Ergo *heuristics* are sets of Condition-Action pairs. Typically, a heuristic condition is a test on the form of the current window focus, and perhaps its hypotheses: the action is a tactic or proof command that attempts to manipulate the window in some controlled way. For example, there is a heuristic which (when active) will transform a the term  $P \wedge true$  as the focus of a window into  $P$ , using the appropriate rule from a theory of propositional logic. Thus, heuristics are used to automate simple, repetitive proof steps.

## 3 Modelling the Refinement Calculus

Our definition of refinement is based on predicate transformers: in particular, the weakest precondition predicate transformer of Dijkstra [Dij76]. A predicate is a Boolean-valued function, whose particular value depends on a *state*, mapping (program) variables to their values. The weakest precondition  $wp(S, \psi)$  for a command  $S$  and a predicate  $\psi$  (the *postcondition* — a predicate on the final state) is another predicate  $\phi$  (the *precondition* — a

predicate on the initial state) such that if  $S$  is started in a state satisfying  $\phi$ , it is guaranteed to terminate in a state satisfying  $\psi$ . We define  $wp$  by case analysis on the structure of commands; for example:

$$\begin{aligned} wp(\mathbf{skip}, \psi) &= \psi \\ wp(\mathbf{abort}, \psi) &= \mathit{false} \\ wp(S_1 ; S_2, \psi) &= wp(S_1, wp(S_2, \psi)) \\ wp(\tilde{x} := \tilde{e}, \psi) &= \psi[\tilde{x} \backslash \tilde{e}] \end{aligned}$$

### 3.1 Higher Order Logic

Because we are dealing with predicates implicitly parameterized by state, and because we must consider functions from predicates to predicates, a refinement calculus based on weakest preconditions cannot be a straightforward extension of first-order classical logic. In particular, we cannot naïvely treat program variables as logical variables and  $wp$  as a function. We say that the  $wp$  connective is *modal*, because it does not preserve the usual (classical) substitutivity of equality; it is not the case that  $a = b \Rightarrow wp(S, p(a)) = wp(S, p(b))$  (take  $S \triangleq a := a + 1$ , for example). This is because  $a = b$  stands for a function from a state (which gives values to  $a$  and  $b$ ) to a truth value, whereas the corresponding classical formula (in which  $a$  and  $b$  are constants) itself denotes a truth value. In this sense,  $wp$  behaves like a *modal* operator, since it transforms the (implicit) state parameter. Another way to view this is to say that  $wp$  is not monotonic with respect to our ‘=’ (so = is not a true equality relation).

The most theoretically elegant solution to this dilemma is to reason in higher-order logic. The domain of discourse includes the Boolean values  $tt$  and  $ff$ , all the values of our programming types (0, 1, ‘a’, ‘b’, 3.1416, etc.), and may also include the program variables themselves. States map program variables to values; they are modelled either as functions or as tuples of values (the two approaches are compared in Section 3.4). Expressions evaluate to some value in a state; predicates to a Boolean value. Commands are functions mapping predicates to predicates: that is, a command  $S$  is identified with its predicate transformer  $wp(S, \_)$ .

$Val$	Some set of values (with typical elements $u, v, w$ ).
$Bool \triangleq \{tt, ff\}$	The Boolean (truth) values (typical element $\xi$ ).
$Sta$	States (with typical elements $\sigma, \tau$ ). Two possible definitions for this type are discussed below.
$Expr \triangleq Sta \rightarrow Val$	Expressions map states to values (typical elements $e, f$ ).
$Pred \triangleq Sta \rightarrow Bool$	Predicates map states to truth values (typical elements $\phi, \psi$ ).
$Cmd \triangleq Pred \xrightarrow{m} Pred$	Commands are monotonic predicate transformers, turning predicates on final states to predicates on initial states (typical elements $S, T, U$ ).

### 3.2 Lattice Theory of Predicates

The Booleans  $Bool \triangleq \{tt, ff\}$  are totally ordered by the relation  $\leq$  (Boolean implication), with  $ff \leq tt$ . Every finite total order forms a complete lattice (see e.g. [Bir48, Mor87]), so the Booleans do so; the meet, join and complement functions so induced are Boolean conjunction  $\&$ , disjunction  $|$  and negation  $\sim$  respectively. If we order the states  $Sta$  by the trivial relation of state equality, then the predicates  $Pred \triangleq Sta \rightarrow Bool$  form a class of monotonic (trivially!) functions whose range is a complete lattice, and hence  $Pred$  too is a complete lattice ([Mor87] again), whose order is the relation (in  $Pred \times Pred \rightarrow Bool$ ) obtained by a pairwise lifting of  $\leq$ :

$$\phi \Rightarrow \psi \triangleq \forall \sigma \bullet \phi \sigma \leq \psi \sigma$$

This induces meet and join functions  $\wedge$  and  $\vee$  in  $Pred \times Pred \rightarrow Pred$ , and complementation function  $\neg$  in  $Pred \rightarrow Pred$ , which correspond to the pointwise lifted Boolean connectives:

$$\begin{aligned} \phi \wedge \psi &\equiv \lambda \sigma \bullet \phi \sigma \& \psi \sigma \\ \phi \vee \psi &\equiv \lambda \sigma \bullet \phi \sigma | \psi \sigma \\ \neg \phi &\equiv \lambda \sigma \bullet \sim \phi \sigma \end{aligned}$$

We can also define predicate implication  $\Rightarrow: Pred \times Pred \rightarrow Pred$  in terms of join and negation; it is easy to show that it corresponds to pointwise lifted Boolean implication:

$$\begin{aligned} \phi \Rightarrow \psi &\triangleq \neg\phi \vee \psi \\ &\equiv \lambda\sigma \bullet (\lambda\tau \bullet \sim\phi\tau)\sigma \mid \psi\sigma \\ &\equiv \lambda\sigma \bullet \sim\phi\sigma \mid \psi\sigma \\ &\equiv \lambda\sigma \bullet \phi\sigma \leq \psi\sigma \end{aligned}$$

Similarly, we can lift other Boolean connectives and functions; for example, we can define predicate (state-dependent) constructs for  $\forall$  and  $\exists$  that quantify some of the state variables and free logical variables.

It is useful to define a function in  $Pred \rightarrow Bool$  that takes an arbitrary predicate formula (defined by pointwise-lifting a Boolean formula) and universally quantifies it over all states (the *everywhere* operator of Dijkstra and Scholten [DS90]):

$$[\psi] \triangleq \forall\sigma \bullet \psi\sigma$$

We also define a function in  $Bool \rightarrow Pred$  that lifts a Boolean formula to the corresponding predicate (which ignores its state argument):

$$\bar{\xi} \triangleq \lambda\sigma \bullet \xi$$

Finally, we combine the two into a function  $Pred \rightarrow Pred$  whose result holds in all states (ignoring its actual state argument); we call the operation *necessary* by analogy with the comparable operator in functional and other modal logics:

$$\begin{aligned} nec(\psi) &\triangleq \overline{[\psi]} \\ &\equiv \lambda\sigma \bullet (\forall\tau \bullet \psi\tau) \end{aligned}$$

Now, instead of writing  $\phi \Rightarrow \psi$  when we need state-independent implication on predicates, we can write  $nec(\phi \Rightarrow \psi)$ ; the advantage is that this is a predicate, so can participate in further predicate formulas (perhaps having free logical variables bound by quantification, or being conjoined with some predicate that *does* depend on state); also, *nec* is more general since we can ‘universalize’ any predicate formula that way, whereas  $\Rightarrow$  requires us to cast our formula as an implication.

### 3.3 Lattice Theory of Commands

The commands  $Cmd \triangleq Pred \xrightarrow{m} Pred$  are monotonic functions whose range is a complete lattice, so  $Cmd$  too forms a complete lattice [Mor87, BvW89, GM91]. The order is lifted  $\Rightarrow$ :

$$S \subseteq T \triangleq \forall \psi \bullet S\psi \Rightarrow T\psi$$

This order is our usual notion of *refinement*. The meet function in the lattice is  $\sqcap$  (demonic choice), and the join is  $\sqcup$  (angelic choice). The bottom of the lattice is **abort**, and the top is **magic**. The rest of our usual command constructors can be fitted into the lattice of commands as well.

As we did with other Boolean-valued functions, we lift the refinement order back to a predicate that ignores its state argument:

$$S \sqsubseteq T \triangleq \overline{S \subseteq T}$$

Thus we allow  $S \sqsubseteq T$  to appear freely in predicate formulas, and can avoid ever dealing with raw Boolean values.

### 3.4 Modelling States

We discuss two ways to model states. The two approaches will be called ‘States-as-Functions’ and ‘States-as-Tuples’.

#### States-as-Functions

In the approach of [BvW89], states are modelled as functions from a space of program variables ( $Var$ ) to a space of values ( $Val$ ):

$$Sta \triangleq Var \rightarrow Val$$

We assume a supply of names (identifiers) to denote the program variables  $Var$  (typical elements  $a, b$ ). A name uniquely identifies a variable (that is, different identifiers denote distinct variables), and these visible, persistent names are used in expressions, predicates and commands. *Expressions* then

denote functions from state to value, just as predicates denote functions from state to Boolean values:

$$Expr \triangleq Sta \rightarrow Val$$

For example, the assignment command  $a := a + a$  can be written:

$$a := a^\uparrow \text{++} a^\uparrow$$

Here:

- $a$  is an identifier;
- $\uparrow$  is a function mapping an identifier to the expression representing its value;
- $\text{++}$  is a function mapping a pair of expressions to an expression (obtained by lifting the primitive function  $+$  that adds integer values); and
- $:=$  is a function mapping an identifier and an expression to a command.

Op	Type	Definition
$:=$	$Var \times Expr \rightarrow Cmd$	$a := e \triangleq (\lambda\psi\bullet\psi[a := e])$
$\text{++}$	$Expr \times Expr \rightarrow Expr$	$e \text{++} f \triangleq (\lambda\sigma\bullet e\sigma + f\sigma)$
$\uparrow$	$Var \rightarrow Expr$	$a^\uparrow \triangleq (\lambda\sigma\bullet\sigma a)$
$\text{--[} := \text{--]}$	$Pred \times Var \times Expr \rightarrow Pred$	$\psi[a := e] \triangleq (\lambda\sigma\bullet\psi(\sigma \oplus \{a \mapsto e\sigma\}))$

This approach requires a dereference ( $\uparrow$ ) to turn an identifier  $a$  into the expression  $a^\uparrow$  that gives the value  $\sigma a$  when applied to a state  $\sigma$ ; naturally, we could introduce syntactic sugar to make dereference implicit when an identifier appears in an expression context.

Since we assume flat universes of identifiers, this approach does not allow us to easily model lexically scoped variables. A variation [BvW90a] supports multiple spaces of identifiers. Expressions and predicates are then indexed by a variable space, so if we have two var-spaces  $A$  and  $B$  where  $A$  is the

set of variables whose identifiers are  $\{a, b\}$  and  $B$  has  $\{a, c\}$ , the predicate  $a = 0_A$  is different from the predicate  $a = 0_B$ , though there are coercion operators that can be used explicitly or assumed in contexts where coercion is needed. Commands map final predicates over one var-space to initial predicates over another, so each command has an *arity* (actually a whole family of arities) expressing the change of var-space. We must then consider whole families of predicate lattices, and whole families of command lattices. For example, the **skip** command maps  $\psi_A$  (a postcondition  $\psi$  over var-space  $A$ ) to itself, so **skip** has arity  $A \rightarrow A$ . There are explicit commands to add and remove variables; thus the command  $\langle +a := u \rangle$  adds a new identifier  $a$  (and gives it the initial value  $u$ ), while  $\langle -a \rangle$  removes  $a$ :

$$\begin{aligned} (\langle +a := u \rangle)\psi_A &\triangleq \psi[a \setminus u]_B \quad \text{where } a \notin A \text{ and } B = A \cup \{a\} \\ (\langle -a \rangle)\psi_A &\triangleq \psi_B \quad \text{where } B = A \setminus \{a\} \end{aligned}$$

Ordinary variable blocks are defined in terms of these commands. Still, the method does not permit arbitrary reuse of identifiers, since the requirement  $a \notin A$  in the first definition above prevents the redeclaration of an identifier  $a$  if it is already in scope. Arguably, such redeclaration is undesirable anyway (since it compels us to consider ‘scopes with holes’).

We also assume a flat universe of values, so it is not obvious how to model typed variables. A possible solution is to augment the var-space index with type information, so a command  $\langle +a:T := u \rangle$  would add an identifier  $a$  together with its type  $T$  (and initial value  $u$ ).

The States-as-Functions approach has been used to describe the predicate-transformer semantics of a programming notation as a theory in the prover HOL [BvW90b], which can then be used to reason about refinement. It seems likely that this HOL theory would be cumbersome to use in practice, because of the need to manage explicit var-spaces and to distinguish variables from their values (by dereferencing). The next approach can be seen as an attempt to simplify this.

### States-as-Tuples

In this approach [vW94], names are not represented at all. Instead, a state is viewed as a tuple of (possibly typed) values:

$$Sta \triangleq Val \times \dots \times Val$$

Ordinary commands can change these values, and scoped variable declaration commands augment and diminish the tuples. Since there are no identifiers, the language does not have expressions; in their place, we use functions from initial state to final state written as explicit tuples of values inside the scope of a  $\lambda$  binder.

For example, if our current state space is a three-tuple whose first component corresponds to the variable  $a$ , the command  $a := a + a$  is written:

$$\mathit{assign} \lambda(u, v, w) \bullet (u + u, v, w)$$

Here, the state function  $\lambda(u, v, w) \bullet (u + u, v, w)$  maps an initial state  $\sigma$  (a 3-tuple of values) to a final state  $\tau$  whose first element is twice the value of the first element of  $\sigma$ , and whose second and third elements are identical to the corresponding elements of  $\sigma$ . The function  $\mathit{assign} : (Sta \rightarrow Sta) \rightarrow Cmd$  maps its state-function argument  $e$  (playing the role of an expression) to a command  $\mathit{assign} e$  that maps a predicate  $\psi$  to another predicate  $\lambda\sigma \bullet \psi(e\sigma)$ .

Without persistent variable names, every expression and predicate must be written as a  $\lambda$ -expression whose formal parameters name the components of the state. State functions such as that used in  $\mathit{assign}$  above must give the final value of all components, even those that are unchanged from the initial state ( $v$  and  $w$  above).

The States-as-Tuples approach has been used to implement a usable tool supporting the refinement calculus in HOL [vW94].

## 4 Representing Refinement in Ergo

One way to build a theory of refinement in Ergo would be to directly model one of the higher-order theories of refinement just described. There are several reasons we would prefer not to do this:

- The need for explicit or implicit dereference in the States-as-Functions approach, and the difficulties with types and scopes, make that approach a clumsy way of reasoning about programs.
- The  $\lambda$ -bound terms in the States-as-Tuples approach are unpleasant to look at, and the need to explicitly constrain unchanged values in state transformers is cumbersome.

- Ergo currently has no support for higher-order logic.
- There is a large existing body of Ergo theories about propositional and predicate calculus, arithmetic etc., which would have to be redeveloped (or reinterpreted) in a lifted-state higher-order framework.

The issues regarding the appearance of constructs (dereference in States-as-Functions and the abundance of  $\lambda$ -expressions in States-as-Tuples) are serious impediments to the acceptability of these notations. One could invent ‘syntactic sugar’ to improve this a great deal, but reasoning would be done in ‘unsugared’ terms, and it seems probable that (in some circumstances at least) the unsugared notation would become visible to the user of a tool supporting such activity. In any case, the kinds of sugaring necessary are not currently available in Ergo. A very significant benefit of the refinement calculus is that it allows the user to abstract away from notions of state (and indeed from predicate transformers), and work purely at the level of program-to-program transformations and first-order logic proofs; the programs can be expressed using the familiar syntax of imperative languages. With the HOL approaches, at least as described here, the familiar notation is not adequate. The difficulties with types and scopes (States-as-Functions) and the framing problem (States-as-Tuples) are also a concern.

A version of Ergo with support for the syntax of higher-order logic (allowing variables and compound terms to appear in functor positions in terms) will soon be released, but there is no immediate proposal to provide higher-order theories, nor support them with tactics and heuristics. An axiomatization of higher-order logic could certainly be built upon Ergo’s intuitionistic base (with or without syntactic support), or, indeed, higher-order logic could be defined in set theory. That would involve work that is not immediately relevant to this project, and we should prefer to avoid it if possible. Even if it were done, it would be necessary to redevelop or reinterpret our definitions and proofs about first-order logic and arithmetic in a framework where predicates and expressions are functions (on states), rather than simple values. Ergo is designed with such reinterpretation in mind [SRH94], but we need the ability to reason at *two* levels: to discharge obligations like ‘ $Pre \Rightarrow Post$ ’, in which identifiers denote values in some state, we must reason at the ‘lifted’ level; but to discharge obligations like ‘ $\tilde{w}$  is a list of fresh identifiers’, in which identifiers denote themselves, we must reason in ordinary first-order logic.

Instead of building a *deep embedding* [BG94] of the higher-order refinement calculus in Ergo, we use a *shallow embedding*, where we treat certain definable concepts axiomatically. Our approach is to build an axiomatic theory of weakest preconditions upon which we define refinement and the refinement calculus. This axiomatic theory is designed with a higher-order logic model in mind, but we will not actually do the modelling of higher-order logic in Ergo, nor have we restricted the notation used in the theory to one that could be furnished with a model in the current Ergo. It is important to have persistent identifiers that can be used in programs in the familiar way, so the States-as-Tuples approach is unlikely to be useful. Our axiomatic development corresponds more to the States-as-Functions model, with identifiers, expressions, predicates and commands denoted explicitly by Ergo terms whose forms are something like the familiar names, terms, formulas and commands we normally write. The Ergo mechanisms for transforming window hypotheses are used to ensure that reasoning is always done in the appropriate context (lifted or otherwise).

In the following, we show essentially the Ergo syntax as it appears in our theory files, but freely use special characters that are not normally available in Ergo's (ASCII) input and output. Generally, Ergo keywords are in **sans serif** font, and declared symbols are in **type-writer** font.

## 4.1 Types

We shall need to refer explicitly to program variables, so we do introduce a type `ident` (Section 4.6 discusses possible definitions for this type):

```
constant ident.
```

This declaration simply names a primitive (undefined) constant `ident`, which we shall treat as a type by using it inside terms of the form `a:ident` interpreted as a typing constraint (see Section 2.3). The primitive constant plays much the same role as a *given set* in Z [Spi92].

Anticipating a possible later modelling in higher-order logic, we shall also introduce primitive constructors (providing precedence and associativity so they can be written in infix form) for the types of partial, total and monotonic functions:

```

function 200 leftassoc  $\leftrightarrow$  (T1,T2) .
function 200 leftassoc  $\rightarrow$  (T1,T2) .
function 200 leftassoc  $\xrightarrow{m}$  (T1,T2) .

```

When supplied type arguments, these constructs can be used on the right-hand side of ‘:’ in declarations. Our initial axiomatization will not make use of this type information; the declarations serve merely as documentation.

We shall avoid talking explicitly about general values and states in our theories. For documentation, we do define types for them:

```

constant val .
constant state === ident  $\leftrightarrow$  val .

```

We can then give types for expressions, predicates and commands:

```

constant expr === state  $\rightarrow$  val .
constant pred === state  $\rightarrow$  bool .
constant cmd === pred  $\xrightarrow{m}$  pred .

```

## 4.2 Expressions and Predicates

We want to avoid the need to develop theories about (and tactic support for) predicate logic, set theory and arithmetic in which expressions and predicates are lifted to functions from states. So that we can take advantage of existing theories, we propose to use the Ergo terms **true** and **false** to represent the lifted Boolean constants (predicates)  $\overline{tt}$  and  $\overline{ff}$ , instead of their usual meaning (*tt* and *ff* themselves). We shall use the functors **and**, **or**, **not** etc. (usually denoting Boolean connectives) to denote the corresponding lifted connectives (on predicates), so the existing Ergo axiomatization of (first-order) intuitionistic logic is reinterpreted in the lifted framework. Similarly, we shall allow the terms that usually denote the constants, functions and relations of set theory to denote the corresponding concepts in our lifted universe, thereby making the existing theory hierarchy directly available to us. We believe that this *ad hoc* reinterpretation of the existing theories is sound (really, we are just providing an alternative model, as in Functional

Logic [SRH94]); to demonstrate that it is, we should verify that Ergo's in-built axioms and inference rules, and the axioms in the theories we use, do in fact hold in our intended model. Of particular interest are those axioms pertaining to equality; they ought to be satisfied, since all functions in the existing theories that we make use of are *non-modal* (that is, they admit substitution of equals for equals). We shall see in Section 4.7 how functions that are modal (which we shall need to describe predicate transformers) can be supported in Ergo.

### 4.3 Commands

The syntax of a selection of commands is given now. The means by which their semantics is expressed is defined in the next section.

Some atomic commands:

```
constant skip:cmd.  
constant abort:cmd.  
constant magic:cmd.
```

Sequential composition is expressed with an infix constructor `semi`:<sup>1</sup>

```
function 945 rightassoc semi(S:cmd, T:cmd):cmd.
```

Selections and iterations are expressed by applying `if` and `do` constructors to alternations (`alt`) of guarded commands (`then`):

```
function 946 nonassoc then(B:pred, S:cmd):cmd.  
function 947 rightassoc alt(S:cmd, T:cmd):cmd.  
function if(S:cmd):cmd.  
function do(S:cmd):cmd.
```

---

<sup>1</sup>Later versions of our theory may generalize this to take a list of commands to be sequentially composed; the same goes for alternation (below), and for several operators on expressions and predicates.

The semantics of selection and iteration will (initially, at least) be defined for whole **if**  $b_1 \rightarrow s_1 \parallel \dots \parallel b_n \rightarrow s_n$  **fi** terms, and similarly for **do**  $\dots$  **od**, rather than by piecewise definition if **if**, **do**, **alt**, **then**.

Assignments and specification statements are expressed as constructs involving lists of identifiers; this is supported by a simple theory of lists:

```
function 940 nonassoc :=(Xs:list(ident), Es:list(expr)):cmd.
function spec(Xs:list(ident),  $\phi$ :pred,  $\psi$ :pred):cmd.
```

The awkward syntax for specification statements will be ‘sugared’ in the remainder of this report, and written  $Xs: [\phi / \psi]$ .

We shall (initially) support the syntax of local invariants [MV90], without defining their semantics (which requires an augmented *wp* connective):

```
function linv(J:pred, S:cmd):cmd.
```

In this report we sugar the syntax for invariants, writing  $\llbracket \text{linv } J \bullet S \rrbracket$ .

Finally, blocks with local variables are expressed using the following declaration:

```
function lvar(Xs:list(ident), S:cmd):cmd.
```

We will sugar this too, and write  $\llbracket \text{lvar } Xs \bullet S \rrbracket$ . A block  $\text{tvar}(Xs, T, S)$  (sugared form:  $\llbracket \text{tvar } Xs:T \bullet S \rrbracket$ ) with typed local variables abbreviates the corresponding block with untyped local variables, an initialization and a local invariant:

```
abbreviation  $\llbracket \text{tvar } Xs:T \bullet S \rrbracket$  ===
   $\llbracket \text{lvar } Xs \bullet Xs: [\text{true} / Xs:T] \text{ semi}$ 
     $\llbracket \text{linv } Xs:T \bullet S \rrbracket$ 
   $\rrbracket$ .
```

## 4.4 Weakest Preconditions

Because we do not intend to support higher order logic, we shall not identify commands with their weakest precondition transformers, as is usual in a higher-order approach. Instead, we shall define a two-place functor  $wp$  to represent the function that maps a command and a predicate (postcondition) to a new predicate (precondition):

```
function wp(S:cmd,  $\psi$ :pred):pred.
```

The  $wp$  function is then defined by cases, with one axiom for each command constructor:

```
axiom wpskip      === wp(skip,  $\psi$ ) =  $\psi$ .
axiom wpabort    === wp(abort,  $\psi$ ) = false.
axiom wpmagic    === wp(magic,  $\psi$ ) = true.
axiom wpsemi     === wp(S semi T,  $\psi$ ) = wp(S, wp(T,  $\psi$ )).
axiom wpassign   === wp(Xs := Es,  $\psi$ ) = subs(Xs, Es,  $\psi$ ).
etc.
```

The  $subs$  function used in `wpassign` is explained below, and defined in more detail in Section 4.7.

We shall treat only commands whose predicate transformers are monotonic with respect to implication (hence the use of  $\overset{m}{\rightarrow}$  in the type definition for `cmd`). This means we can state an opening rule [RS93, UW94] that preserves implication, logical equivalence and equality when moving the window focus from a  $wp$  term to its predicate argument. This simplified opening rule discards all the window hypotheses  $\Gamma$ ; we shall see why, and see a way to avoid discarding all of them, in Section 4.7.

$$\frac{\vdash \psi \text{ R } \psi'}{\Gamma \vdash wp(S, \psi) \text{ R } WP(S, \psi')} \quad \text{R is one of } \Rightarrow, \Leftarrow, \Leftrightarrow$$

We also need an opening rule for the command argument of  $wp$ , and indeed for the command constructors themselves. Discussion of these is deferred until the next section.

The *wp* connective is modal (see Section 3.1), so the usual notion of substitution by replacing free occurrences of variables by terms does not work: for example, replacing occurrences of *a* by *b* in  $wp(b := a + 1, a = b)$  then calculating *wp* is not the same as calculating *wp* first then doing the replacement in the result:

<p>WRONG :</p> $wp(b := a + 1, a = b)[a \setminus b]$ $\equiv wp(b := b + 1, b = b)$ $\equiv b + 1 = b + 1$ $\equiv true$	<p>RIGHT :</p> $wp(b := a + 1, a = b)[a \setminus b]$ $\equiv a = a + 1[a \setminus b]$ $\equiv b = b + 1$ $\equiv false$
---	---

In a sense, *wp* acts like a quantifier that binds (some of) the variables in the state so the simple-minded replacement of occurrences does not work. In other ways, *wp* does not behave at all like a quantifier since the result of calculating a *wp* then *does* contain occurrences of program variables that we should like to think of as free.<sup>2</sup> Our proposed solution depends on a declaration that *wp* (along with a few other operators) is *modal*; the semantic notion of substitution (**subs**) that represents state-change corresponds to the syntactic notion of replacement of free occurrences only through non-modal operators.<sup>3</sup> This will be discussed fully in Section 4.7.

## 4.5 Refinement

Now the relation of refinement  $\sqsubseteq$  (pronounced ‘refsto’, and spelt that way in our Ergo theory files), is defined in terms of weakest preconditions:

$$\text{function } 949 \text{ nonassoc } \sqsubseteq(S, T) \quad === \\ \forall \psi : \text{pred} \bullet \text{nec}(wp(S, \psi) \Rightarrow wp(T, \psi)).$$

<sup>2</sup>It is most like a combination of  $\lambda$ -abstraction (binding) with  $\lambda$ -application (leaving free occurrences) — which is exactly how the States-as-Tuples model (Section 3.4) represents it.

<sup>3</sup>In fact, we declare all ordinary (=monotonic) operators as *non-modal*, and define semantic substitution (axiomatically) by syntactic substitution only through non-modal constructs. Doing it this way around avoids questions about closed-world assumptions: does the absence of a *modal* declaration for an operator mean that the operator is non-modal?

In fact, since we are denoting predicates by the same terms as Boolean expressions, this definition is not adequate, since the quantification needs to range over just those predicates expressed using only non-modal (=monotonic) constructors. Our actual definition is given in Section 4.7.

This relation  $\sqsubseteq$  is reflexive and transitive (a preorder); we prove this (by unfolding the definition and reasoning about *wp* — proof omitted) and declare  $\sqsubseteq$  to Ergo as an appropriate window relation:

```

theorem reflexrefsto === S ⊆ S.
theorem transrefsto === S ⊆ T and T ⊆ U ⇒ S ⊆ U.
is_preorder(⊆).

```

For convenience, we also declare the reversed refinement relation  $\supseteq$  ('refines') and the refinement equality  $\sqsubseteq$  ('refeq'). The former is a preorder and the latter is an equivalence (that is, it is reflexive, transitive and symmetric), and  $\sqsubseteq$  is a stronger relation than  $\sqsubseteq$  and  $\supseteq$ :

```

function 949 nonassoc ⊇(S, T) === T ⊆ S.
function 949 nonassoc ⊆(S, T) === S ⊆ T and T ⊆ S.
theorem reflexrefines === S ⊇ S.
theorem transrefines === S ⊇ T and T ⊇ U ⇒ S ⊇ U.
is_preorder(⊇).
theorem reflexrefeq === S ⊆ S.
theorem transrefeq === S ⊆ T and T ⊆ U ⇒ S ⊆ U.
theorem symrefeq === S ⊆ T ⇒ T ⊆ S.
is_equivalence(⊆).
theorem refeqimprefsto === S ⊆ T ⇒ S ⊇ T.
is_stronger_than(⊆, ⊇).
theorem refeqimprefines === S ⊆ T ⇒ S ⊇ T.
is_stronger_than(⊆, ⊇).

```

We can now give an opening rule for the command argument of *wp*, and for both arguments of  $\sqsubseteq$ ,  $\supseteq$  and  $\sqsubseteq$ . Again, these opening rules are simplified in that they discard hypotheses  $\Gamma$ , some of which can in theory be retained, as discussed in Section 4.7.

First argument of  $wp$ :

$$\frac{\vdash S \text{ R}' S'}{\Gamma \vdash wp(S, \psi) \text{ R WP}(S', \psi)} \quad \text{R determines R' according to: } [\Rightarrow \mapsto \sqsubseteq, \Leftarrow \mapsto \sqsupset, \Leftrightarrow \mapsto \sqsubseteq]$$

First and second arguments of  $\sqsubseteq$ :

$$\frac{\vdash S \text{ R}' S'}{\Gamma \vdash (S \sqsubseteq T) \text{ R} (S' \sqsubseteq T)} \quad \text{R determines R' according to: } [\Rightarrow \mapsto \sqsupset, \Leftarrow \mapsto \sqsubseteq, \Leftrightarrow \mapsto \sqsubseteq]$$

$$\frac{\vdash T \text{ R}' T'}{\Gamma \vdash (S \sqsubseteq T) \text{ R} (S \sqsubseteq T')} \quad \text{R determines R' according to: } [\Rightarrow \mapsto \sqsubseteq, \Leftarrow \mapsto \sqsupset, \Leftrightarrow \mapsto \sqsubseteq]$$

The opening rules for  $\sqsupset$  are the opposites (in an obvious way) of those for  $\sqsubseteq$ . To maintain  $\sqsupset$ , we must transform either argument only by  $\sqsupset$ , whether the original relation is  $\Rightarrow$ ,  $\Leftarrow$ , or  $\Leftrightarrow$ . There is no need to give the transformations for  $\Rightarrow$  and  $\Leftarrow$  explicitly, as the rule for  $\Leftrightarrow$  will be used, since that relation is stronger:

$$\frac{\vdash S \sqsupset S'}{\Gamma \vdash (S \sqsupset T) \Leftrightarrow (S' \sqsupset T)}$$

$$\frac{\vdash T \sqsupset T'}{\Gamma \vdash (S \sqsupset T) \Leftrightarrow (S \sqsupset T')}$$

To support the refinement of components, we also need opening rules for command constructors. Each constructor is monotonic with respect to the refinement relation, so the opening rules for them keep the same relation. Once again, because of the state-dependence of predicates, the rules discard all hypotheses. The rules for **if** and **do** open to a command in a branch in a single step, as the usual definition for **if** and **do** applied to a command other than a guarded command set is not  $\sqsubseteq$ -monotonic [Nel89]. In a future version, we shall generalize these rules to apply to **if** and **do** commands with

arbitrary numbers of guarded commands.

$$\frac{\vdash S \text{ R } S'}{\Gamma \vdash (S ; T) \text{ R } (S' ; T)} \text{ R is one of } [\sqsubseteq, \supseteq, \sqsupseteq]$$

$$\frac{\vdash T \text{ R } T'}{\Gamma \vdash (S ; T) \text{ R } (S ; T')} \text{ R is one of } [\sqsubseteq, \supseteq, \sqsupseteq]$$

$$\frac{\vdash S \text{ R } S'}{\Gamma \vdash \left( \begin{array}{c} \text{if } B \text{ then } S \text{ alt } C \text{ then } T \text{ fi} \\ \text{R} \\ \text{if } B \text{ then } S' \text{ alt } C \text{ then } T \text{ fi} \end{array} \right)} \text{ R is one of } [\sqsubseteq, \supseteq, \sqsupseteq]$$

$$\frac{\vdash T \text{ R } T'}{\Gamma \vdash \left( \begin{array}{c} \text{if } B \text{ then } S \text{ alt } C \text{ then } T \text{ fi} \\ \text{R} \\ \text{if } B \text{ then } S \text{ alt } C \text{ then } T' \text{ fi} \end{array} \right)} \text{ R is one of } [\sqsubseteq, \supseteq, \sqsupseteq]$$

etc.

## 4.6 Denoting Identifiers

Perhaps the most significant choice to be made is of the Ergo terms that will be used to denote identifiers. The requirements include:

1. Persistence: an identifier should always denote the same variable, at least throughout that variable's scope.
2. Distinctness: it should be possible to tell whether two identifiers denote distinct variables (though the variables may or may not have the same values), or whether some program variable occurs in some term, for example when checking whether a proposed local variable is fresh.

3. Substitution and quantification: some predicate transformer definitions and some refinement rules (in particular, those involving assignment commands) require the calculation of formulas (denoting predicates) that involve systematic substitution of expressions for identifiers. Other rules and definitions (local variables, specification statements) require that some predicate is universally or existentially quantified over some program variables (so the free identifiers become bound logical variables).

We have tried three different approaches to representing identifiers in Ergo, and at the same time come to an understanding of their nature. We summarize these approaches, and their advantages and disadvantages, here; the remaining sections of the report detail the final approach, which we think will be successful. The persistence requirement rules out approaches like States-as-Tuples in which program variables are anonymous or denoted by bound logical variables. We could produce a sound theory of refinement by directly supporting the States-as-Functions approach in higher-order logic, with explicit states mapping program variables to values. For the reasons discussed at the start of Section 4, we do not want to do that. Instead, we shall try to subsume the  $Var \rightarrow Val$  mapping by one of the mappings in the semantics of Qu-Prolog.

### Identifiers as Object-var Variables

The first approach tried was perhaps the simplest: using Ergo object-var variables as program identifiers. This means in effect that we are treating program variables as free logical variables in predicates and expressions, and using the *valuation* mapping (from logical variables to values) to represent states. A significant advantage of such an approach is that Ergo's syntax for substitution and quantification (each of which binds object-var variables) is available, so the axiom defining  $wp$  for a single assignment command  $x := E$  (using the object-var variable  $x$  to stand for the identifier) is just:

$$\text{axiom } wpassign \quad === \quad wp(x := E, R) = [E/x]*R$$

where  $[E/x]R$  is Qu-Prolog's (and Ergo's) own notation for syntactic substitution. Similarly, we can write an axiom defining  $wp$  for a block with a single local variable as:

$$\text{axiom wpvar} \quad === \quad \text{wp}(\llbracket \text{lvar } x \bullet S \rrbracket, \psi) = \forall x \bullet \text{wp}(S, \psi).$$

Sad to say, this (conceptually simplest) approach is not feasible. Because functions like  $wp$  are modal, the intended meaning of substitution does not always correspond to Ergo’s treatment of it — we would get behaviour like the ‘WRONG’ column in the example on page 23. This could be worked around by adding applicability conditions to axioms like `wpassign` that constrain  $\psi$  to be non-modal. An alternative would be to define a new function `subs: ident × expr × pred → pred`, whose semantics are expressed using Qu-Prolog substitution only for predicate arguments that involve no non-modal operators. The first option suffers because we do not have a complete definition of  $wp$ , and hence cannot do induction on the structure of commands (to prove that refinement is transitive, for example). The second loses us the advantage of directly exploiting Ergo substitution and quantification in our definitions and axioms, though we shall see that that is necessary anyway in the other approaches.

More seriously, Ergo will introduce constraints about the distinctness of object-var variables and their occurrence in terms that do not necessarily correspond to the constraints we need on program variables. This is desirable in that it gives us our distinctness requirement for identifiers automatically, but it is disastrous when combined with uses of object-var variables that range over semantic objects other than program variables. For example, we would like to express the definition of refinement as:

$$S \sqsubseteq T \quad \triangleq \quad \forall \psi \bullet \text{wp}(S, \psi) \Rightarrow \text{wp}(T, \psi).$$

In the Ergo representation of this definition,  $S$  and  $T$  would be metavariables, and  $\psi$  would be an object variable (intended to range over predicates). Ergo would automatically generate the constraints ‘ `$\psi$  not_free_in  $S$` ’ and ‘ `$\psi$  not_free_in  $T$` ’, since one would not normally want such a definition to capture occurrences of  $\psi$  in  $S$  or  $T$  (the `not_free_in` constraints cause Ergo to  $\alpha$ -convert the bound  $\psi$  to avoid that capture, when the definition is used to unfold some instance of  $S \sqsubseteq T$ ). However, our  $S$  and  $T$  might contain a free object-var variable  $x$  (representing a program identifier), and the `not_free_in` constraint extends to that too, setting  $\psi$  and  $x$  apart. The effect is to exclude from the range of  $\psi$  all those predicates involving  $x$  — which is clearly inappropriate! With such a definition, it is easy to prove the invalid `skip  $\sqsubseteq$  x := 1`, for example. Using this representation, we were able

to define an Ergo theory containing refinement laws that could be used to do correct program developments, but the theory in which those refinement laws were supposed to be proved was unsound.

## Identifiers as Metavariables

In this approach, Prolog’s meta-level mapping from metavariables to terms is used in place of explicit states mapping program variables to values. This experiment yielded usable theories of weakest preconditions and refinement in which we had reasonable confidence in proofs as long as certain conventions about the allowable treatment of metavariables were adhered to (for example, a metavariable standing for a program variable should never be instantiated!).

The most obvious cost in moving from the first approach to this one is that Ergo substitution and quantification are no longer directly available for binding occurrences of identifiers in axioms like `wpassign` and `wpvar`, even in predicate formulas that involve no modal operators. This meant that we had to define a function `subs` by a meta-level replacement of metavariables (standing for identifiers) by terms (standing for expressions), so duplicating in Prolog code the very functionality that Qu-Prolog’s quantified terms are supposed to provide. The danger in doing this is that one might introduce errors; obvious traps to avoid are accidental name capture and failure to handle renamings (multiple substitutions involving permutation of some names) correctly. These pitfalls are harder to avoid in Qu-Prolog, with object-var variables to consider, than in traditional Prologs.

The requirement to be able to distinguish identifiers is solved by an axiom (again defined at the meta-level, not in terms of an underlying theory) that says that different metavariables (Prolog ‘==’) denote distinct program variables. In combination with our desire to do schematic proofs (one of Ergo’s strengths), in which metavariables are used to stand for arbitrary terms, the decision whether a program variable occurs in some term is more difficult to express. Again, we are trying to invent at the meta-level an operation (`not_free_in`) that Qu-Prolog provides us as a primitive.

The most serious fault with this way of expressing identifiers directly relates to the fact that they have been lifted right out of our universe of discourse, so cannot be referred to by axioms. For example, we cannot write a  $\forall$  term

in which the bound variable ranges over all identifiers, and, since predicates and expressions involve program variables, we cannot give any meaning to quantifiers ranging over these objects either. This suggests that, in moving identifiers from the Qu-Prolog object-var level to the meta level, we have gone in the wrong direction, and would be better to model them directly as objects in the logic.

### Identifiers as constants

In this model, program variables are explicit objects in our universe of discourse, so the identifiers that denote them are expressed with Ergo constants, and the  $Var \rightarrow Val$  state mapping is subsumed by the interpretation of constants (function symbols of arity zero) as elements of the universe of discourse. This is the closest approximation to the States-as-Functions model we have tried. With hindsight, it is the most obvious approximation to make; however, the complexity caused by the need to define meta-level substitution and distinctness led us to try the other approaches first.

We must now explicitly declare every identifier (as a constant of type ‘`ident`’) before we use it; in the previous approaches, there was a potentially infinite supply of identifiers.<sup>4</sup> We have set up tactics that automatically furnish such declarations when new names are used in contexts where an identifier is expected. Distinctness of variables is defined by an axiom that says that identifiers whose names are different indeed denote different variables; the comparison can be done using the Prolog *dif* predicate, which is true when its arguments are provably distinct:

```
function 200 nonassoc distinct_from (X:ident, Y:ident):pred.  
axiom distinct_idents ===  
  X:ident  $\wedge$  Y:ident  $\Rightarrow$  X distinct_from Y :-  
  dif(X, Y).
```

We can extend this definition to express a requirement that some identifier does not occur free in some term; we define `X meta_nfi T` to be true when the identifier `X` is distinct from every free identifier in term `T`. We can also

---

<sup>4</sup>In fact, we can recover that infinite supply in the next release of Ergo using *parameterized constants*.

define predicates like `meta_subset(V, W)` which is true if the identifiers in `V` are a subset of those in `W` (which is not the same as `subset(W, V)`, where we talk about sets of *values*).

Rather than requiring explicit dereference of identifiers when their values are needed, we give them a dual interpretation; in some contexts, an identifier is an object (of type `ident`) that stands for a program variable; in other contexts, it is a constant of some programming type (`int`, `float` etc.) whose value is that mapped to the identifier by the ‘current state’. The two roles of identifiers are kept apart by defining functions and axioms so that they refer to only one interpretation, and giving opening rules that preserve the appropriate properties. For example, the condition ‘`a distinct_from b`’ can be proved using the axiom `distinct_idents` just given (assuming `a` and `b` are declared as having type `ident` in the current theory); whereas the condition ‘`a > b`’ would be proved by appealing to hypotheses about `a` and `b` in the current window, and to axioms and theorems about `>` in the current theory (the axioms and theorems about `>` would presumably refer to the programming types of `a` and `b` in the current window).

## 4.7 Handling Modality

Most of our reasoning is done in an environment where identifiers are implicitly dereferenced in some state (or states). Thus, in the precondition of a specification statement, an instance of an identifier `a` in an expression `a + 1` stands for the value of `a` in some implicit pre-state  $\sigma_0$ ; it is that value that is to be added to one, not the variable that `a` denotes. If that expression appeared in an invariant, `a` is again implicitly dereferenced with respect to a state  $\sigma$ ; this time,  $\sigma$  itself is implicitly universally quantified (that is, the invariant is taken to hold for all states  $\sigma$ ). To reason about these kinds of terms easily, we want to work in an environment where the usual connectives, predicate and function symbols of first-order logic, set theory, etc. are interpreted in a state-relative domain. This *implicit parameters* approach is the basis of *functional logic* [SRH94], in which the reinterpretation of symbols, axioms and theorems from classical logic is formally justified. The concept is similar to modal logic (where the implicit parameter concerns ‘possible worlds’) and temporal logic (where time is the implicit parameter).

One difficulty with this approach is the treatment of equality. It is convenient to use the equality symbol `=` to represent equality in some state. When we

write  $a = b$  in the precondition of some specification statement, we mean that any mechanism that implements that specification may assume that the values of  $a$  and  $b$  in the initial state are the same; we do not mean that the set of all states is to be constrained so that  $a = b$  in every state (in particular, it is quite possible that  $a \neq b$  in the final state). This means that we can not always substitute equals for equals, but must have regard for the implicit state parameter when doing so. It is useful to define two forms of equality. The *weak* equality is the one just discussed, which holds only in a few states; for this, we use the traditional symbol  $=$  (since that is the kind of equality normally used in specifications and programs). We also define *strong* equality, which we can denote by  $==$ ; we can say  $\alpha == \beta$  only when expressions  $\alpha$  and  $\beta$  denote exactly the same object (and hence have the same value in all states). This strong equality *is* substitutive, like the classical equality.

It is useful to be able to use weak equality for reasoning when we know that the implicit state parameter does not change. For example, we can always replace a predicate  $a = 1 \wedge b = a$  by  $a = 1 \wedge b = 1$ , since there can be no change of implicit state between the conjuncts. To capture this, we must identify which constructs in our logic and programming theory involve change of state and which do not: we do this by identifying the *modal* (state-changing) and *non-modal* operators. The most important modal operator is *wp*; its modality was demonstrated in Section 4.4. We extend the concept of modality to cover general terms: a term is (non-)modal if and only if every operator within it is (non-)modal.

Now we have two versions of substitution of equals for equals: one (for strong equality) is unconditional; the other (for weak equality) is conditional on the non-modality of the operators involved.

$$\begin{aligned} x == y &\Rightarrow C[x] == C[y] && \text{for any term } C[x]. \\ x = y &\Rightarrow C[x] = C[y] && \text{providing } C[x] \text{ is a non-modal term.} \end{aligned}$$

Here,  $C[x]$  is some term including free  $x$ , and  $C[y]$  is that same term with all free occurrences of  $x$  replaced by  $y$ .

It is also useful to define a weak notion of substitution to correspond with our weak equality, so that we can recover an unconditional substitution of equals for equals rule. We define the three-place **subs** operator so that

$\mathbf{subs}(a, e, T)$  is the same as  $T[a \setminus e]$  (syntactic substitution) when  $T$  is non-modal.<sup>5</sup> There are no rules to evaluate  $\mathbf{subs}(a, e, T)$  when  $T$  is modal. The operator  $\mathbf{subs}$  itself is modal, since it involves a state-shift: if the implicit state parameter of  $\mathbf{subs}(a, e, T)$  is  $\sigma$ , then the implicit state parameter of  $T$  itself is  $\sigma \oplus \{a \mapsto e\sigma\}$ . This notation is extended in an obvious way to permit multiple (weak) substitution.

Note that our weak substitution captures a program variable, not a logical (object) variable. This is significant because ordinary Qu-Prolog (and Ergo) syntactic substitution cannot be used to replace anything other than object variables, so weak substitution is the only one available e.g. for defining  $wp$  of an assignment command:

`axiom wpassign == wp(A := E,  $\psi$ ) = subs(A, E,  $\psi$ ).`

(here  $A$  is a metavariable that will be bound to a program identifier (or a list thereof) when the axiom is used). In Ergo, a weak substitution  $\mathbf{subs}(a, e, T)$  must thus be defined by meta-level manipulation of terms even when  $T$  is non-modal. We also define weak quantifiers that can bind program variables; `qall` is the weak analogue of  $\forall$ , and `qex` is the weak analogue of  $\exists$ . These weak quantifiers are defined using ordinary quantifiers and weak substitution, and are also modal operators.

Modality, distinctness, substitution, etc. are defined over terms and formulas, not as functions of values. Thus, as we have seen, the formula  $wp(a := 0, b = a)$  is modal, even though it denotes the same Boolean-valued function as the formula  $b = 0$ . In Ergo, we define `modal`, `distinct_from`, `subs` etc. as functions. By careful control over how the functions and terms involving them are used we can avoid problems with representing these meta-level concepts using the notation of ordinary (object-level) functions. Providing a better formalization of modality and related concepts is the subject of ongoing research.

To enable window reasoning about modality, our window opening rules for modal operators manipulate the hypothesis list, discarding any hypotheses that might be affected by the state-shift produced by the operator.<sup>6</sup> For

---

<sup>5</sup>As usual,  $a$  is a program identifier and  $e$  is an expression.

<sup>6</sup>In Sections 4.4 and 4.5, the opening rules discarded *all* hypotheses, avoiding this issue completely.

example, if the current focus is  $\text{wp}(\mathbf{a}:=1, \text{Post})$  and there is a hypothesis  $\mathbf{a}=\mathbf{b}$ , that hypothesis must be deleted when the focus moves to *Post*, whereas a hypothesis  $\mathbf{b}=\mathbf{c}$  is not affected. The opening rule for *wp* is actually:

$$\frac{\Gamma' \vdash \psi \text{ R } \psi'}{\Gamma \vdash \text{wp}(S, \psi) \text{ R } \text{WP}(S, \psi')} \quad \begin{array}{l} \text{R is one of } [\Rightarrow, \Leftarrow, \Leftrightarrow] \\ \text{Each hypothesis } H \text{ in } \Gamma \text{ becomes} \\ \text{ } sp(H, S) \text{ in } \Gamma'. \end{array}$$

That rule uses the defined function *sp* (itself a modal operator); for a predicate *P* and a command *S*, *sp(P, S)* is the strongest postcondition *S* can establish if started in a state satisfying precondition *P*. Our Ergo theory defines *sp* for a limited range of commands only,<sup>7</sup> including assignments and specification statements.

Equipped with connectives for describing modality, we can present our actual definition of refinement (compare the one in Section 4.5):

$$\begin{array}{l} \text{function } \text{949 nonassoc } \sqsubseteq(S, T) \quad === \\ \quad \forall \psi : \text{pred} \bullet \text{nonmodal}(\psi) \Rightarrow \\ \quad \quad \text{nec}(\text{wp}(S, \psi) \Rightarrow \text{wp}(T, \psi)). \end{array}$$

In this definition, *ψ* must be taken to range over *formulas*, not the functions they denote, so we can restrict attention to nonmodal formulas.

## 4.8 Tactics and Heuristics

It is not sufficient to provide the definitions of the syntax and semantics of our language and refinement relation. To construct a useful refinement tool, we shall need a set of refinement rules (proved as theorems in the above refinement theory); we shall also need tactics and heuristics for handling the mundane tasks that arise when doing refinements in such a low-level theory. At present, we have simple tactics for stating a refinement rule (and perhaps initiating its proof), for posing a refinement problem and for applying rules with their parameters instantiated. Continuing the development of this infrastructure will be our main task in the near future.

---

<sup>7</sup>In fact, for nondeterministic *S*, unique strongest postconditions do not always exist.

## Stating rules

The `refinement_rule` command invokes a tactic that generates a theorem corresponding to some refinement rule, and initiates its proof. The form of the command is:

```
refinement_rule name (parameters) ===
  applicability conditions
  ⇒
  Subject ⊆ Result.
```

For example, the rule that introduces an assignment command is:

```
refinement_rule assI(Xs, Es) ===
  meta_subset(Xs, W) and
  nec(Pre ⇒ subs(Xs, Es, Post))
  ⇒
  W:[Pre / Post] ⊆ Xs := Es.
```

The name is `assI`; it has two parameters (`Xs`: a list of identifiers, and `Es`, a list of expressions); it has two applicability conditions, an arbitrary specification statement as subject and a assignment command as result. When the above command is issued (while building the refinement theory), Ergo will start a proof that the rule is valid. The `refinement_rule` keyword can be replaced by `postulated_refinement_rule`, in which case Ergo takes the rule as a postulate and does not require a proof.

## Stating refinement problems

A specification is presented to the tool using the `refinement` command:

```
refinement name === Spec.
```

The command invokes a tactic that first declares as a constant of type `ident` every name that appears in `Spec` where an identifier is expected (in a frame, after the `lvar` or `tvar` keyword, etc.). The tactic then starts Ergo proving a theorem that `Specification` is refined by `Code` (an Ergo metavariable that will be instantiated to the code we produce), as if we had written:

`theorem name == Spec  $\sqsubseteq$  Code.`

The initial window has the *Spec* as focus, has relation  $\sqsubseteq$ , and no hypotheses.

### Applying refinement rules

Within a refinement, rules are applied with the tactic `apply`. This tactic has two forms:

`apply(name(parameters))` will instantiate the rule's parameters directly, whereas

`apply(name, result)` will select a suitable instantiation to achieve the specified result of the application.

The tactic first declares any new names that will be introduced by the application, then applies the appropriately instantiated rule. Any applicability conditions must be discharged (automatically if possible, otherwise using the normal Ergo proof interface) before the tactic returns.

## 5 Example: Euclid's Algorithm

This section contains an example of how a simple refinement can be done in our Ergo theory. A refinement diagram [Bac91] describing the development of Euclid's algorithm for finding the greatest common divisor of two integers appears in Figure 1.<sup>8</sup> We have drawn the refinement diagram vertically, with refinement steps denoted by doubled vertical lines (this layout permits more detail to be shown on one static diagram than does Back's horizontal layout; Back envisages the notation to be displayed and navigated by an interactive tool with hypertext-like browsing and outlining facilities [BHS92]).

---

<sup>8</sup>See also [CHN<sup>+</sup>94b], where the same refinement is described in two earlier refinement tools.

## 5.1 Specification

The specification could be expressed in our refinement theory as follows:<sup>9</sup>

```
refinement euclid ===  
  [[ tvar v,a,b:int • v:[ a>0 ∧ b>0 / v = gcd(a, b) ] ]].
```

Here:

- `v`, `a` and `b` will be declared as constants of type `ident` by the `refinement` tactic;
- `int` is a constant in the theory in which the refinement is done, intended to denote the integer programming type; and
- `gcd` is a function `int × int → int` in that theory, as are the functions `>`, `=` and `∧` (with the obvious types).

We can proceed to do the refinement (that is, to find an acceptable binding for `Code` to prove the theorem generated by `refinement`) by focusing on the specification statement `v:[...]`. The focus command [2] (focus on the second argument of `tvar`, which is the `cmd` argument) will do this, and we get a window with the specification statement as focus, relation  $\sqsubseteq$  and containing the hypothesis `nec(v,a,b:int)`. This new hypothesis was generated by the opening rule for `tvar`, and says that `v,a,b:int` may be assumed to hold in all states represented in this window.

## 5.2 Refinement: New Local Variable

Our first refinement is to introduce a typed local variable. The refinement rule that does this is:

---

<sup>9</sup>The program variables `v`, `a` and `b` are declared in a local variable block as a means for making their scope (the entire specification) explicit; eventually, we shall have some notation that explicitly states the context of a specification without the absurdity of making a block that hides the variables of interest.

```

refinement_rule tvarI(X:fresh_ident, T) ==
  X:ident and
  X meta_nfi [T, W, Pre, Post] and
  nec(qex(X, X:T))
  =>
  W:[Pre / Post]   ⊆   [[ tvar X:T • {X}∪W:[Pre / Post] ]].

```

The command `refinement_rule` is the tactic for stating rules (see Section 4.8); in this case the rule has parameters `X` and `T`: a fresh identifier `X` will be declared (by `apply`) if necessary. The requirements that `X` is indeed an identifier, and that it does not occur in the specification statement being refined, are explicit conditions (antecedents) on the rule body, as is a requirement that the type `T` is not empty (so that the initialization implicit in `tvar` is feasible).

The rule `tvarI` can be applied to the specification statement in the focus with the command `'apply(tvarI(w,int))'`. The effect is to generate the effect of a new constant declaration (thanks to `fresh_ident` in the parameter list of `tvarI`):

```
constant w:ident.
```

The rule is then instantiated according to its parameter list, and used to transform the focus. The result is the new focus:

```
[[ tvar w:int • v,w:[ a>0 ∧ b>0 / v=gcd(a,b) ] ]].
```

The three obligations on `tvarI` are discharged automatically (by the `apply` tactic and suitable heuristics) as follows:

```
w:ident:
```

```
using the newly-generated constant declaration for w.
```

```
w meta_nfi [int, v, a>0 ∧ b>0, v=gcd(a,b)]:
```

```
using the (axiomatic) definition of meta_nfi to explode the list of
terms into a list of atomic symbols (identifiers and constants), then
the axioms that different identifiers are distinct from one another and
from all constants.
```

```
qex(w, w:int) <=> true:
```

```
this is an instance of an axiom in the theory that defines the program-
ming types (all of which are non-empty), and nec(true) = true.
```

### 5.3 Refinement: Split Specification

Our refinement proceeds by developing the specification statement inside that block, so we focus on it (using the command [2] again). The window relation remains  $\sqsubseteq$ , the hypothesis  $\text{nec}(v, a, b : \text{int})$  from before is unchanged, and the new hypothesis  $\text{nec}(w : \text{int})$  appears.

The next step is to split the specification statement: one component of the resulting sequential composition will be the loop initialization, the other the loop itself. The appropriate refinement rule is:

$$\text{refinement\_rule semI(Mid) ===} \\ \text{W: [Pre / Post] } \sqsubseteq \text{ W: [Pre / Mid] semi W: [Mid / Post] .}$$

This unconditional rule takes an intermediate assertion  $\text{Mid}$  as a parameter; we supply the desired loop invariant  $v > 0 \wedge w > 0 \wedge \text{gcd}(v, w) = \text{gcd}(a, b)$ . The result is the new focus

$$v, w : \left[ a > 0 \wedge b > 0 \ / \ v > 0 \wedge w > 0 \wedge \text{gcd}(v, w) = \text{gcd}(a, b) \right] \text{ semi} \\ v, w : \left[ v > 0 \wedge w > 0 \wedge \text{gcd}(v, w) = \text{gcd}(a, b) \ / \ v = \text{gcd}(a, b) \right] .$$

### 5.4 Refinement: Loop Initialization

We focus on the first component of that composition [1]. We want the multiple assignment command  $v, w := a, b$ , so we apply the refinement rule  $\text{assI}$  (shown in Section 4.8) using the command  $\text{apply}(\text{assI}, v, w := a, b)$ . Instantiating and applying this rule gives the desired assignment command as the new focus. The obligations are discharged as follows:

$\text{meta\_subset}([v, w], [v, w])$ :  
 a tactic in the theory of lists reduces this to  $v : \text{ident}$  and  $w : \text{ident}$ ; both those facts follow from the implicit **constant** declarations introduced earlier.

$a > 0 \wedge b > 0 \Rightarrow \text{subs}(\dots)$ :  
 We discharge this obligation manually by:

1. Focusing on the right-hand side of  $\Rightarrow$ ; this adds the conjuncts  $a > 0$  and  $b > 0$  from the left-hand side to the hypothesis list.
2. Applying the defining axiom for `subs`, whose obligation is that the matrix is non-modal; this is automatically discharged. The result is  $a > 0 \wedge b > 0 \wedge \text{gcd}(a,b) = \text{gcd}(a,b)$ .
3. The first two conjuncts of this are identical to the hypotheses added in 1 above, and the last conjunct is an instance of an axiom that says  $=$  is reflexive.

Ergo heuristics could easily be written to do simple reasoning like this automatically.

After discharging the obligations, we `quit` from the window with the initialization, then focus [2] on the second specification statement.

## 5.5 Refinement: Introduce DO

The current focus is:

$$v, w: [v > 0 \wedge w > 0 \wedge \text{gcd}(v, w) = \text{gcd}(a, b) \ / \ v = \text{gcd}(a, b)].$$

The rule for introducing a two-branch DO is:

```

refinement_rule do2I([B1,B2], Inv, Vart,
                    Vart0:fresh_ident) ===
Vart0 meta_nfi [W, Inv, Vart, B1, B2] and
nec(Pre  $\Rightarrow$  Inv) and
nec(Inv  $\wedge$   $\neg$ (B1  $\vee$  B2)  $\Rightarrow$  Post)
 $\Rightarrow$ 
W:[Pre / Post]  $\sqsubseteq$ 
do B1 then
  W:[B1  $\wedge$  Inv  $\wedge$  Vart=Vart0 /
    Inv  $\wedge$  0  $\leq$  Vart  $\wedge$  Vart < Vart0
alt B2 then
  W:[B2  $\wedge$  Inv  $\wedge$  Vart=Vart0 /
    Inv  $\wedge$  0  $\leq$  Vart  $\wedge$  Vart < Vart0
od.

```

The parameters to this rule are:

**[B1,B2]**: a list of guards, currently restricted to be of length two;<sup>10</sup> we supply  $[v > w, w > v]$ .

**Inv**: an invariant, for which we supply  $v > 0 \wedge w > 0 \wedge \text{gcd}(v, w) = \text{gcd}(a, b)$ .

**Vart**: a variant, for which we supply  $v + w$ .

**Vart0**: a fresh name to capture the initial value of the variant in each branch,<sup>11</sup> for which we supply  $v0$ .

The instantiated obligations (allowing *Pre*, *Inv* and *Post* to continue to stand for the precondition, loop invariant and postcondition) are:

$v0$  `meta_nfi`  $[[v, w], \text{Inv}, v + w, v > w, w > v]$ :  
discharged as in Section 5.2.

`nec`( $Pre \Rightarrow Inv$ ):

*Pre* and *Inv* are identical (since we chose *Inv* as the intermediate assertion in Section 5.3), and  $\text{nec}(A = A) = \text{true}$ .

`nec`( $Inv \wedge \neg(v > w \vee w > v) \Rightarrow Post$ ):

We rewrite  $\neg(v > w \vee w > v)$  to  $v = w$  using a rule of arithmetic and the types  $v : \text{int}, w : \text{int}$  from the hypothesis list. Then, we substitute  $v$  for  $w$  in *Inv*; the result contains a conjunct  $\text{gcd}(v, v) = \text{gcd}(a, b)$ . The left-hand side of that rewrites (using a fact from the gcd theory) to  $v$ , and the resulting conjunct is identical to *Post*.

It now remains to introduce the assignment commands in the branches of the `do`; nothing new is illustrated by this so the details are omitted here.

---

<sup>10</sup>Later, we shall generalize this rule to one that takes an arbitrary number of guard parameters and introduces a `DO` with that many branches.

<sup>11</sup>Properly, this name should be declared (with a `con`) inside each guarded command.

## 6 Conclusions

We have reviewed the modelling of the refinement calculus in higher-order logic and described how we were able to construct a refinement theory in Ergo. This theory is based on weakest preconditions and a simple modal logic, and was constructed in Ergo without the need to build a deep embedding of refinement and predicate transformer semantics involving higher-order logic. The example derivation showed (in a small way) how window inference could help control the context in which proof obligations are discharged. It also showed how we could combine reasoning in a modal logic (with lifted functions) with meta-level reasoning about identifiers.

Our tool based on the refinement Ergo theory has so far been used only on tiny examples (like the one in this paper). To make it practical for larger examples, we need a much better user interface, whose design and implementation using a language-based editor is the subject of another branch of the project. The refinement back-end itself needs to be enhanced in the following ways:

1. Providing better tactic support for applying refinement rules and navigating refinement graphs.
2. Providing tactics and heuristics to assist with the discharging of some of the obligations that arise in refinement, in particular the obligations about identifiers (which are normally considered extra-logical).
3. Enhancing the wide-spectrum language with formally-defined types, recursion, procedures, modules and other constructs.
4. Modifying the refinement relation where necessary to cope with these new constructs, and to support related notions of refinement (such as data refinement of modules).
5. Considering how window inference can be used to more effectively manage the kinds of context that are peculiar to refinement.
6. Better formalizing the concepts of modality, substitution, identifier distinctness etc.
7. Developing models of refinement and proof that can be supported by our Ergo tactics and with the language-based editor that forms the front-end of our tool.

## References

- [B<sup>+</sup>85] F. L. Bauer et al. *The Munich Project CIP, Volume I*, volume 183 of Lecture Notes in Computer Science. Springer-Verlag, 1985.
- [B<sup>+</sup>87] F. L. Bauer et al. *The Munich Project CIP, Volume II*, volume 292 of Lecture Notes in Computer Science. Springer-Verlag, 1987.
- [Bac78] R. J. R. Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, University of Helsinki, 1978.
- [Bac88] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.
- [Bac91] R. J. R. Back. Refinement diagrams. In Joseph M. Morris and Roger C. Shaw, editors, *Fourth Refinement Workshop, Workshops in Computing*, pages 125–137. BCS FACS, Springer-Verlag, 1991.
- [BG94] Jonathan Bowen and Mike Gordon. Z and HOL. In J. P. Bowen and J. A. Hall, editors, *Z User Workshop, Cambridge 1994*, BCS FACS Workshops in Computing, pages 141–167. Springer-Verlag, 1994.
- [BHS92] R. J. R. Back, J. Hekanaho and K. Sere. Centipede — a program refinement environment. Ser. A 139, Åbo Akademi, 1992.
- [Bir48] G. Birkhoff. *Lattice Theory*, volume 25 of Colloquium Publications. American Mathematical Society, 1948.
- [BvW89] R. J. R. Back and J. von Wright. A lattice-theoretical basis for a specification language. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of Lecture Notes in Computer Science, pages 139–156. Springer-Verlag, 1989.
- [BvW90a] R. J. R. Back and J. von Wright. Command lattices, variable environments and data refinement. Series A 102, Åbo Akademi – Departments of Computer Science and Mathematics, 1990.

- [BvW90b] R. J. R. Back and J. von Wright. Refinement concepts formalised in Higher Order Logic. *Formal Aspects of Computing*, 2:247–272, 1990.
- [CHN<sup>+</sup>94a] David Carrington, Ian Hayes, Ray Nickson, Geoffrey Watson and Jim Welsh. Requirements for a program refinement engine. Technical Report 94-43, Software Verification Research Centre, University of Queensland, 1994.
- [CHN<sup>+</sup>94b] 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, University of Queensland, 1994.
- [Dij76] E. W. Dijkstra. *A Discipline of Programming*. Academic Press, 1976.
- [DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
- [GM91] Paul Gardiner and Carroll Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.
- [Mor87] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.
- [Mor94] Carroll Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.
- [MR87] Carroll C. Morgan and Ken Robinson. Specification statements and refinement. *IBM Journal of Research and Development*, 31(5):546–555, September 1987.
- [MV90] Carroll Morgan and Trevor Vickers. Types and invariants in the refinement calculus. *Science of Computer Programming*, 14:281–304, 1990.
- [Nel89] Greg Nelson. A generalization of Dijkstra’s calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.

- [RC93] Peter J. Robinson and Anthony Cheng. Qu-Prolog 3.2 reference manual. Technical Report 93-18, Software Verification Research Centre, University of Queensland, 1993.
- [RS93] Peter Robinson and John Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1):47–61, 1993.
- [RW93] Peter J. Robinson and K. Whitwell. The demonstration interactive theorem prover Demo3.3. Technical Report 93-4, Software Verification Research Centre, University of Queensland, 1993.
- [Spi92] J. M. Spivey. *The Z Notation: A Reference Manual (second edition)*. Prentice Hall International, 1992.
- [SRH94] John Staples, Peter J. Robinson and Daniel Hazel. A functional logic for higher level reasoning about computation. *Formal Aspects of Computing*, 6:1–38, 1994.
- [UW94] Mark Utting and Keith Whitwell. Ergo user manual. Technical Report 93-19, Software Verification Research Centre, University of Queensland, 1994.
- [vW94] J. von Wright. Program refinement by theorem prover. In David Till, editor, *Sixth Refinement Workshop*, Workshops in Computing. BCS FACS, Springer-Verlag, 1994.

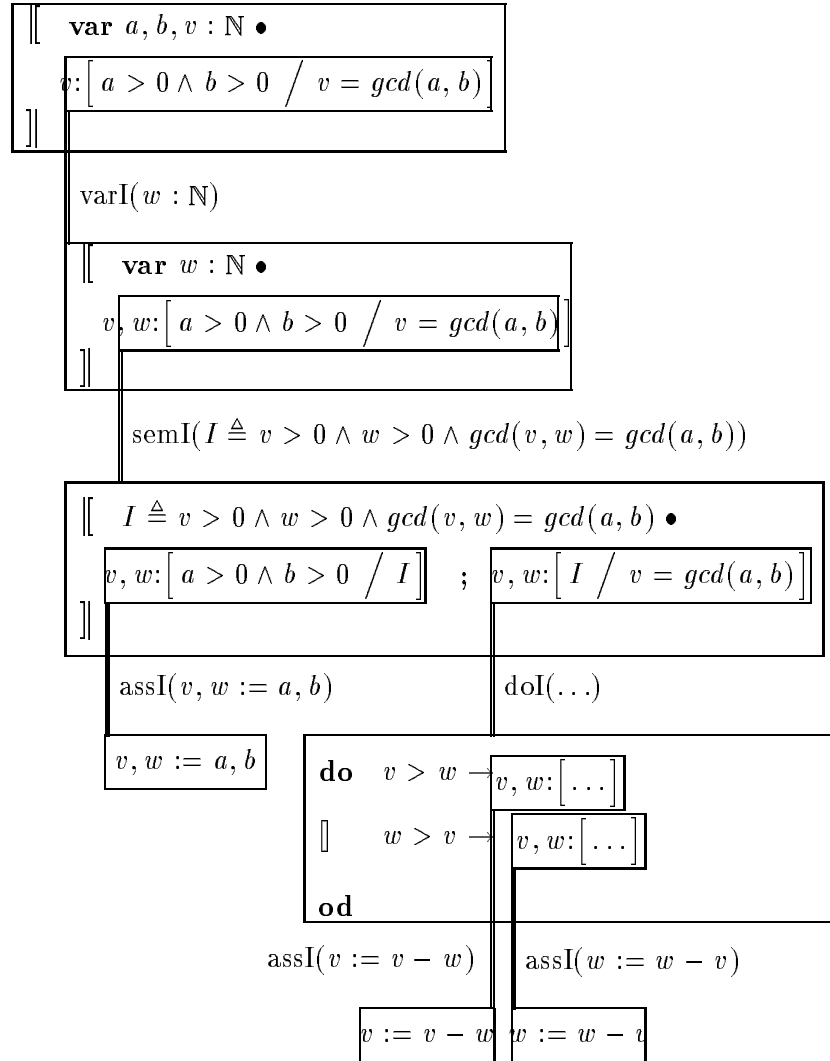


Figure 1: Derivation of Euclid's Algorithm