Formal Aspects of Computing (1990) 2: 247–272

Formal Aspects of Computing

Refinement Concepts Formalised in Higher Order Logic

R. J. R. Back[a] and J. von Wright[b]

[a] Åbo Akademi University, Department of Computer Science, Lemminkäisenkatu 14, SF-20520 Turku, Finland and [b] Swedish School of Economics and Business Education, Biblioteksgatan 16, SF-65100 Vasa, Finland

Keywords: Mechanical theorem proving; Program development; Weakest preconditions; Refinement calculus; Higher order logic

Abstract. A theory of commands with weakest precondition semantics is formalised using the HOL proof assistant system. The concept of refinement between commands is formalised, a number of refinement rules are proved and it is shown how the formalisation can be used for proving refinements of actual program texts correct.

1. Introduction

The refinement calculus is a theory of program transformations that preserve the total correctness of programs. It was first described by Back [Bac78, Bac80] and has been further elaborated by Back [Bac88], Morgan and others [GMR88] and Morris [Mor87]. It is based on the weakest precondition technique of Dijkstra [Dij76]. The refinement calculus has been used as a tool for stepwise refinement of sequential algorithms, and recently also for the derivation of parallel algorithms from sequential algorithms [Bac89, BaS89, Wri89].

The HOL system (Higher Order Logic) is a theorem proving assistant, which can be used to formalise theories and verify proofs of theorems within these theories. It is based on the LCF system [MGW79] and is described in [Gor88]. It has been used mainly for formal specification and verification of hardware. However, a simple imperative programming language is formalised in the HOL system by Gordon [Gor89] in an attempt to formalise Hoare logic.

Correspondence and offprint requests to: R. J. R. Back, Åbo Akademi University, Department of Computer Science, Lemminkäisenkatu 14, SF-20520 Turku, Finland.

Another formalisation of a simple programming language, using the original LCF, is used by Sokolowski [Sok87] in a mechanical proof of the soundness of Hoare's Logic. However, neither of these formulations permits nondeterminism in the programming language.

The assignment statement has proved to be the most difficult part when formalising imperative programming languages in mechanised logics. Mason [Mas87] performs a thorough analysis of this problem when formalising Hoare's logic in the LF system.

This paper describes the first results of a project aimed at developing tools for refining imperative programs using the HOL system. Our aim is to show that the concepts of the weakest precondition calculus and the refinement calculus can be formalised in HOL. Furthermore, we want to verify the basic theory of weakest preconditions and the refinement calculus by re-doing the proofs in HOL. By proving the transformation rules of the refinement calculus we show that using the formalisation for reasoning about programs is sound. In this way our formalisation can form the basis for a mechanical system for complete verification of program refinements.

1.1. Organisation of the Paper

The rest of the paper is organised as follows. Sections 2 and 3 describe briefly the refinement calculus and the HOL system. In Section 4 we describe a formalisation of predicates. Predicates are defined as functions from states to truth values. Using the HOL system it is proved that the predicates form a complete boolean lattice and that every monotonic function on predicates has a least fixpoint. These results provide a basis for the formalisation of weakest preconditions. In Section 5 we formalise the basis concepts of the refinement calculus. We first define a Dijkstra-style specification language with a weakest precondition semantics (the commands of the language are defined as syntactic entities). In Section 6 we discuss two different ways of formalising the refinement relation within the theory of commands. In each formalisation, some rules of refinement are proved. Section 7 contains a small example where the formalisation is used to prove a refinement of an actual program text correct. We also show how larger refinements can be performed. Finally, Section 8 contains some concluding remarks.

1.2. Remarks on Notation

The HOL system uses ASCII characters or character sequences for the symbols of higher order logic. We will use standard symbols to make the text more readable (however, we use T and F for the boolean truth values). We also assume standard precedence rules for these symbols, except when otherwise is explicitly stated. For readability, we usually omit outermost universal quantifiers in theorems and we do not write out explicit type information in terms and theorems (we assume that the typing is obvious from the context).

2. The Refinement Calculus

We assume that *Var* is a set of program variables and that D is a set of values. A *state* is an assignment of values for each variable, i.e., a total function from *Var* to D. The set of all states (the *state space*) is denoted Σ.

A *predicate* is a total function from Σ to the set $Bool = \{ff, tt\}$ of truth values and an *expression* is a total function from Σ to D. The set of all predicates is denoted *Pred*. *Pred* is a complete boolean lattice when ordered by the implication ordering (i.e., the pointwise extension of the implication ordering on *Bool*):

$$P \Rightarrow Q \stackrel{\text{def}}{=} \forall \sigma.(P(\sigma) \Rightarrow Q(\sigma))$$

The bottom element of *Pred* is the predicate *false* which assigns the value *ff* to every state and the top element is the predicate *true* which assigns the value *tt* to every state. Logical operations (conjunction, disjunction, implication and negation) on predicates are pointwise extensions of the corresponding operations on truth values. As *Pred* is a complete lattice we permit arbitrary conjunctions (meets) and disjunctions (joins). Also, every monotonic function on *Pred* is guaranteed to have a least fixpoint by the fixpoint theorem of Tarski [Tar55].

A *predicate transformer* is a (total) function from *Pred* to *Pred*. We define a *specification language* by introducing a syntactic class of *statements*. Every statement is given a semantics by defining its *weakest precondition* predicate transformer.

The weakest precondition for a statement S with respect to a predicate Q is denoted $\text{wp}(S, Q)$. It is a predicate which holds in a state σ_0 if and only if S is guaranteed to terminate in a state satisfying Q when executed in the initial state σ_0.

2.1. The Specification Language

We now define the syntax of our specification language. It is essentially Dijkstra's language of guarded commands [Dij76], extended with a nondeterministic update statement and a block construct. However, we permit unbounded nondeterminism, so we cannot use Dijkstra's original definition of the semantics of the iterative construct.

Statements of the specification language are defined recursively as follows:

$S ::=$	$\{b\}$	(assert statement)
|	$x := e$	(assignment statement)
|	$x.P$	(update statement)
|	$[\textbf{var}\ x; S]$	(block)
|	$S_1; S_2$	(sequential composition)
|	$\textbf{if}\ b_1 \rightarrow S_1 \,[]\, b_2 \rightarrow S_2\ \textbf{fi}$	(conditional composition)
|	$\textbf{do}\ b \rightarrow S\ \textbf{od}$	(iteration)

Here x is a variable, e is an expression, P, b_1, b_2 and b are predicates, and S, S_1 and S_2 are statements.

2.1.1. *Weakest Precondition Semantics*

Formally, the meanings of all statements are given by the following definitions of their weakest precondition predicate transformers:

$$\begin{aligned}
\mathrm{wp}(\{b\}, Q) &\stackrel{\mathrm{def}}{=} b \wedge Q \\
\mathrm{wp}(x := e, Q) &\stackrel{\mathrm{def}}{=} Q[e/x] \\
\mathrm{wp}(x.P, Q) &\stackrel{\mathrm{def}}{=} \exists x.P \wedge \forall x.(P \Rightarrow Q) \\
\mathrm{wp}([\mathbf{var}\, x; S], Q) &\stackrel{\mathrm{def}}{=} \forall x.\mathrm{wp}(S, \forall x.Q) \\
\mathrm{wp}(S_1; S_2, Q) &\stackrel{\mathrm{def}}{=} \mathrm{wp}(S_1, \mathrm{wp}(S_2, Q)) \\
\mathrm{wp}(\mathbf{if}\, b_1 \rightarrow S_1 \,[]\, b_2 \rightarrow S_2\, \mathbf{fi}, Q) &\stackrel{\mathrm{def}}{=} (b_1 \vee b_2) \wedge (b_1 \Rightarrow \mathrm{wp}(S_1, Q)) \wedge (b_2 \Rightarrow \mathrm{wp}(S_2, Q))
\end{aligned}$$

For the iterative construct we define $\mathrm{wp}(\mathbf{do}\, b \rightarrow S\, \mathbf{od}, Q)$ to be the least fixpoint of the monotonic function

$$\lambda X.((b \wedge \mathrm{wp}(S, X)) \vee (\neg b \wedge Q)) \tag{1}$$

as done in [Par80, DiG86].

The assignment statement, sequential composition and conditional composition have their usual meanings, as defined in e.g., [Gri81]. The assert statement $\{b\}$ asserts the truth of the predicate b. Thus it aborts if b does not hold and it terminates without affecting any variables if b holds. The update statement assigns a value to the variable x, such that the condition P is established. If this is not possible, the update statement aborts. Using the update statements, arbitrary input–output specifications can be expressed. The meaning of the block statement differs slightly from the meaning given in e.g., [Bac80, Mor87, GaM88]. In order to permit predicates over all the variables of *Var* the local variable is an ordinary variable whose value before entering the block is ignored and whose value is undefined when the block statement terminates. Thus, it is assumed that local variables are not used as global variables.

2.1.2. *Semantics of the Iterative Construct when Nondeterminism is Bounded*

The nondeterminism of a statement S is said to be *bounded* if any computation that is guaranteed to terminate has at most a finite number of possible final states. This is equivalent to the weakest precondition predicate transformer of S being continuous [Dij76]. For statements S with bounded nondeterminism it can be shown (in fact, we have proved it in the HOL system) that

$$\mathrm{wp}(\mathbf{do}\, b \rightarrow S\, \mathbf{od}, Q) = \exists n.\ H_n(b, S, Q) \tag{2}$$

where the function H_n is defined recursively for all natural numbers n by

$$\begin{aligned}
H_0(b, S, Q) &= \neg b \wedge Q \\
H_{n+1}(b, S, Q) &= H_0(b, S, Q) \vee \mathrm{wp}(\{b\}; S, H_n(b, S, Q))
\end{aligned}$$

As noted above, our specification language permits unbounded nondeterminism. Both the update statement and a block with an uninitialised local variable can introduce unbounded nondeterminism. Thus we have to use the fixpoint definition (1) of the weakest precondition of the iterative construct, rather than using (2) as a definition.

2.1.3. *Healthiness Conditions*

The following healthiness conditions are assumed by [Dij76] to hold for every program statement S:

1. *Strictness*, i.e., $\mathrm{wp}(S, \mathit{false}) = \mathit{false}$.
2. *Monotonicity*, i.e., $\mathrm{wp}(S, Q) \Rightarrow \mathrm{wp}(S, Q')$ whenever $Q \Rightarrow Q'$.
3. *Conjunctivity*, i.e., $\mathrm{wp}(S, Q \wedge Q') = \mathrm{wp}(S, Q) \wedge \mathrm{wp}(S, Q')$ for all predicates Q, Q'.
4. *Continuity*, i.e., the nondeterminism of S is bounded.

(Dijkstra's fifth condition, called disjunctivity, is redundant as it is implied by monotonicity.) As noted above, we do not assume continuity. However, all statements will be strict, monotonic and conjunctive. Recently, strictness and conjunctivity have been dropped for statements of specification languages, leaving only monotonicity as the ultimate healthiness condition [GaM88, BaW89a].

2.1.4. *Derived Statements*

Using the given statements it is possible to define a number of derived statements, i.e., statements defined in terms of the primitive statements. We define the **skip** and **abort** statements as follows:

$$\mathbf{skip} \stackrel{\text{def}}{=} \{\mathit{true}\}$$
$$\mathbf{abort} \stackrel{\text{def}}{=} \{\mathit{false}\}$$

We also generalise the **if**- and **do**-statements to permit an arbitrary finite number of branches. The one-branched **if**-statement is defined as

$$\mathbf{if}\ b \rightarrow S\ \mathbf{fi} \stackrel{\text{def}}{=} \{b\}; S$$

The many-branched **if**-statement can now be defined inductively:

$$\mathbf{if}\ b_1 \rightarrow S_1 \,[\!]\, \cdots \,[\!]\, b_{n+1} \rightarrow S_{n+1}\ \mathbf{fi} \stackrel{\text{def}}{=}$$
$$\mathbf{if}\ (b_1 \vee \cdots \vee b_n) \rightarrow (\mathbf{if}\ b_1 \rightarrow S_1 \,[\!]\, \cdots \,[\!]\, b_n \rightarrow S_n\ \mathbf{fi})$$
$$[\!]\, b_{n+1} \rightarrow S_{n+1}$$
$$\mathbf{fi}$$

Similarly, we define the two-branched **do**-statement:

$$\mathbf{do}\ b_1 \rightarrow S_1 \,[\!]\, b_2 \rightarrow S_2\ \mathbf{od} \stackrel{\text{def}}{=} \mathbf{do}\ (b_1 \vee b_2) \rightarrow (\mathbf{if}\ b_1 \rightarrow S_1 \,[\!]\, b_2 \rightarrow S_2\ \mathbf{fi})\ \mathbf{od}$$

and then inductively **do**-statements with more than two branches. For completeness, we also define the empty conditional composition and the empty iteration,

$$\mathbf{if\ fi} \stackrel{\text{def}}{=} \mathbf{abort}$$
$$\mathbf{do\ od} \stackrel{\text{def}}{=} \mathbf{skip}$$

2.2. The Refinement Calculus

The refinement relation $\leq$ on the set of all statements is defined as follows:

$$S \leq S' \stackrel{\text{def}}{=} \forall Q.(\mathrm{wp}(S, Q) \Rightarrow \mathrm{wp}(S', Q))$$

Here the quantification is over all predicates in *Pred*.

The refinement relation is reflexive and transitive, i.e., it is a preorder. It is not antisymmetric on the level of syntactic statements. However, if we choose to identify statements with their weakest precondition predicate transformers, as is done in [Mor87, GaM88, BaW89], then it is antisymmetric. The refinement relation induces an equivalence relation, *refinement equivalence*, denoted $\equiv$.

The refinement relation can be characterised using the notion of total correctness [Bac88]:

$$S \leq S' \quad \text{iff} \quad \forall P, Q.(P[S]Q \Rightarrow P[S']Q)$$

where $P[S]Q$ means that the statement S is totally correct with respect to precondition P and postcondition Q. Thus refinement preserves the total correctness of a statement.

From the transitivity of the refinement relation it follows that if we can show a sequence of refinements

$$S_0 \leq S_1 \leq \cdots \leq S_n$$

then we have shown, using stepwise refinement, that the final statement S_n satisfies any specification that the initial statement S_0 satisfies.

Programs can be refined by parts, because of the *subcomponent replacement* property. If $T(S)$ is a program with S as a subcomponent, then the following holds:

$$S \leq S' \Rightarrow T(S) \leq T(S')$$

2.2.1. Refinement Rules

The refinement calculus can be used to derive algorithms by stepwise refinement, starting from specifications [Bac88, Bac89, Mor90]. Case studies [Bas89, Wri89] show that the stepwise transformation of sequential algorithms into highly parallel algorithms can be done by repeatedly applying a limited number of refinement rules.

3. The HOL System

The HOL system is a proof assistant for higher-order logic based on the Edinburgh LCF theorem proving system. This in turn is a combination of predicate calculus and the typed lambda calculus. This section describes some of the most important features of the HOL system. A more detailed presentation can be found in [Gor88].

3.1. Theories, Types, Terms and Theorems

When working with the HOL system one always works inside some *theory*. Within a theory one can define types, constants and axioms and prove theorems. The *HOL theory* (to be distinguished from the HOL system) is a hierarchy of basic theories, defining among other things the types *bool*,

representing truth values and *num*, representing the natural numbers. Together with these go axioms and theorems of logic and arithmetic. Every user-defined theory has the HOL theory as a parent. Thus the definitions, axioms and theorems of the HOL theory are available in every theory.

The logic of HOL is higher-order logic with a strict type discipline. Thus every entity must have a type assigned to it (polymorphic types containing type variables are permitted).

Types can be combined into *function types*. A function which maps arguments of type *type1* to values of type *type2* has type $type1 \rightarrow type2$. Application of a function f to an argument x is written $f\,x$ or $f(x)$. Function application associates to the left so $f\,x\,y$ is the same as $(f\,x)y$ or $(f(x))(y)$. Most of the time we omit parentheses in order to make function expressions more readable.

Terms of the HOL logic are usually written within double quotation marks. Formulas are treated as terms of type *bool*.

Goals (sequents) are pairs (A,t), where A is a list of terms (the assumptions) and t is a term (the conclusion). If a sequent has been proved (see below), a corresponding *theorem* is returned. Theorems are printed without quotation marks and with a turnstile symbol ($\vdash$) separating the assumptions from the conclusion (an empty assumption list is usually not printed). Theorems have type *thm*. Axioms automatically have this type. Also every definition is represented by a theorem. The only other way to transform a term into a theorem is to prove it using existing theorems and inference rules.

It should be noted that the user can define any boolean term (even a contradiction) to be an axiom. It is therefore recommended [Gor88] that new concepts be formalised using definitions rather than new axioms, since this guarantees that no inconsistencies are introduced. We follow this principle throughout this paper, thus we do not introduce any new axioms.

3.2. Pre-Proved Theorems

In addition to definitions and axioms, the HOL theory contains a number of pre-proved theorems. Two examples of pre-proved theorems are

$$EXCLUDED\_MIDDLE \vdash \forall t.\ (t \vee \neg t)$$
$$ADD\_ASSOC \vdash \forall m\ n\ p.\ (m + (n + p) = (m + n) + p)$$

(we use the convention of preceding a theorem by its name).

3.3. Inference Rules and Forward Proofs in HOL

An inference rule is a function that maps terms and/or theorems to a theorem. As an example we take the primitive inference rule *MP* (Modus Ponens) which is of type $thm \rightarrow thm \rightarrow thm$. Given two argument theorems having the forms $\vdash t \Rightarrow t'$ and $\vdash t$ it returns the new theorem $\vdash t'$. A typical step in a forward proof looks the following way (# is the HOL system prompt and ;; is an input terminator symbol):

#let newthm = MP *th1 th2;;*
newthm = $\vdash$ ...

Here *th1* and *th2* are the names of two existing theorems and the resulting theorem (the dots) becomes bound to the name *newthm* in the current environment. If it is a theorem of special interest we can save it onto the theory file.

3.4. Tactics, Tacticals and Goal-Directed Proofs in HOL

If we want to prove a theorem which requires a long sequence of steps using a forward proof in HOL, we probably have to do an outline of the proof on paper first. Thus HOL works as a proof checker rather than as an interactive proof assistant.

It is possible to do interactive proofs in a goal-directed manner using *tactics* and *tacticals*. A tactic is a function which is applied to a goal and returns subgoals (behind the scenes, the tactic constructs a forward proof of the original goal from the subgoals). A tactical is a function that combines existing tactics into new, more complex tactics.

A goal can be set up using the function *set_goal*. After this, tactics can be applied to the current goal by use of the function *expand*. The subgoals returned by the expansion are added to the goal stack. The initial goal is given theorem-status if all subgoals are eventually reduced to existing theorems. At that point the HOL system constructs the proof of the initial goal corresponding to the sequence of expansions. Thus any theorem proved by goal-directed proof could also have been proved by forward proof.

Many tactics are inverses of a corresponding inference rule. For example, there is a tactic called *MP_TAC*, which takes a theorem $\vdash t$ as argument and reduces a goal t' to the new goal $t \Rightarrow t'$. A few of the most frequently used tactics are presented in Section 4.5, where they are used in an example proof.

We can define new, more complex tactics by means of tacticals. For example, the tactical *THEN* is used to combine tactics into a sequence, *REPEAT* repeats application of a tactic as long as it is applicable to the resulting goals and *ORELSE* tries a second tactic if the first one fails. The following is an example of a combined tactic:

(MP_TAC THEN REPEAT GEN_TAC) ORELSE BETA_TAC

(*GEN_TAC* is a tactic which strips off universal quantifiers while *BETA_TAC* reduces beta-redexes). In this way it is possible to program proof strategies.

3.5. The User Environment

The HOL system is embedded in the ML programming language. ML is a functional language, originally developed as a meta-language for the LCF theorem prover. When starting a HOL session, an ML environment is set up. The user interacts with the HOL system through ML, making definitions and evaluating expressions. It is possible to make very intricate tactics. However, this requires quite an amount of ML programming. In the examples of this paper, we do not assume that the reader is familiar with ML.

4. The Theory of Predicates

We now turn to the formalisation of the refinement calculus. We start from predicates, defining them semantically in terms of variables and values. The fact that we do not define predicates syntactically means that we will not be able to do textual substitutions (Section 4.1). Rather, we have to define substitutions semantically. This will make reasoning about assignment statements somewhat complicated. On the other hand, a syntactic definition of predicates would require embedding first-order logic inside HOL which would also make things rather complicated. This problem is discussed in [Gor89].

4.1. A Formalisation of the Basic Concepts

Program variables are represented by a special type *var* and values by a type *val*. We generally assume that all programs work with natural numbers, so *val* can really be seen as an alternative name for *num*. Typical program variables are denoted by the letters x and y while values are typically denoted by the letter d. We represent sets of variables by their characteristic functions. Thus a set v of variables has type $var \rightarrow bool$, and a variable x belongs to v if and only if $v\,x$ holds.

Since a state is an assignment of a value to each variable, states are represented by the function type $var \rightarrow val$, abbreviated *state*. Typical states are denoted by the letter s. Thus we assume that every program has access to all possible program variables. Uninitialised variables are assumed to be assigned an arbitrary value.

We use *pred* as an abbreviation for the type $state \rightarrow bool$. Thus, *pred* stands for predicates. Note that we do not assume that a predicate can be expressed as a first-order (or higher-order) formula. Predicates are typically denoted by the letters b, p and q.

4.1.1. Variable Substitutions in States

In some situations we need states with a value substituted for a variable. If s is a state, x is a variable and d is a value, then $s[d/x]$ is the state which differs from s only in that it assigns the value d to x. This is formalised by the constant *bind*, defined in the following theorem:

$$bind\_def \vdash \forall d\, x\, s.\ bind\, d\, x\, s = \lambda y.\ (y = x) \Rightarrow d \mid s\, y$$

where $b \Rightarrow x \mid y$ is the HOL system's notation for the conditional expression *if b then x else y*. The definition is given to the HOL system by typing

```
#let bind_def = new_definition('bind_def',
        "bind d x s = λy. ((y = x) ⇒ d | s y)");;
```

As a result, the definitional theorem is saved in the theory file (it is also bound to the name *bind_def* in the current environment).

4.1.2. Operators and Relations on Predicates

We now define operators on predicates, corresponding to the logical symbols; constants, connectives and quantifiers. We define the predicates *false* and *true* of type *pred*, the unary operator *not* of type $pred \to pred$, the binary infix operators *and*, *or* and *imp* of type $pred \to pred \to pred$ and the quantifier operators *exists* and *forall* of type $var \to pred \to pred$. The defining theorems are the following:

$$\begin{aligned}
&false_def \vdash && false = \lambda s.\ F\\
&true_def \vdash && true = \lambda s.\ T\\
&not_def \vdash && not\ q = \lambda s.\ \neg q\ s\\
&and_def \vdash && p\ and\ q = \lambda s.\ (p\ s \wedge q\ s)\\
&or_def \vdash && p\ or\ q = \lambda s.\ (p\ s \vee q\ s)\\
&imp_def \vdash && p\ imp\ q = \lambda s.\ (p\ s \Rightarrow q\ s)\\
&exists_def \vdash && exists\ x\ p = \lambda s.\ (\exists d.\ (p\ (bind\ d\ x\ s)))\\
&forall_def \vdash && forall\ x\ p = \lambda s.\ (\forall d.\ (p\ (bind\ d\ x\ s)))
\end{aligned}$$

i.e., we lift the connectives and quantifiers from *bool* to *pred*. As can be seen above, we name defining theorems by adding the suffix *_def* to the name of the constant being defined.

We also define the implication relation *implies* on predicates:

$$implies\_def \vdash p\ implies\ q = \forall s.\ (p\ s \Rightarrow q\ s)$$

4.1.3. The Variables of a Predicate

We say that a predicate p *depends on* a variable x if there are values d and d' such that $p[d/x] \neq p[d'/x]$. Furthermore, p is a predicate on a set of variables v if p is independent of x for all variables x that are not in v. These notions are formalised by the following definitions, defining the two infix constants *indep* and *pred_on*:

$$\begin{aligned}
&indep_def \vdash && p\ indep\ x = \forall s\ d.\ (p\ (bind\ d\ x\ s) = p\ s)\\
&pred_on_def \vdash && p\ pred_on\ v = \forall x.\ (\neg v\ x \Rightarrow p\ indep\ x)
\end{aligned}$$

Thus $p\ indep\ x$ means that the predicate p does not depend on the variable x while $p\ pred\_on\ v$ means that p is a predicate on the set v of variables (recall that a set of variables is formalised as a function from *var* to *bool*).

4.1.4. Expressions and Substitutions

Expressions can be viewed as functions from states to values. Thus we formalise expressions using the type $state \to val$, abbreviated *expr*. Typical expressions are denoted by the letter e.

We define the infix constant *indepexp* of type $expr \to var \to bool$ for expressions corresponding to the constant *indep* for predicates. The defining theorem is

$$indepexp\_def \vdash e\ indepexp\ x = \forall s\ d.\ (e\ (bind\ d\ x\ s) = e\ s)$$

Substitution of an expression for a variable in a predicate is formalised as follows:

$$subst\_def \vdash subst\ e\ x\ p = \lambda s.\ (p\ (bind\ (e\ s)\ x\ s))$$

4.2. Fixpoints

As we want to define the semantics of the iterative construct, we have to define a fixpoint operator for monotonic functions on predicates. For this, we first define general conjunctions (meets) and disjunctions (joins).

4.2.1. Arbitrary Conjunctions and Disjunctions

As in the case of variables, a set of predicates is formalised as a function of type $pred \rightarrow bool$. If P is a set of predicates then the conjunction of all predicates in P is formalised as $glb\ P$ and the disjunction as $lub\ P$, with the defining theorems

$$glb\_def \vdash glb\ P = \lambda s.\ (\forall p.\ (P\ p \Rightarrow p\ s))$$
$$lub\_def \vdash lub\ P = \lambda s.\ (\exists p.\ (P\ p \wedge p\ s))$$

It is straightforward to show that these actually define greatest lower bounds and least upper bounds of sets of predicates (see Section 4.4 below).

From the Tarski fixpoint theorem [Tar55] it follows that any monotonic function f on a complete lattice L has a least fixpoint. This fixpoint is the meet of the set of elements x such that $f(x) \leq x$. Thus, for functions of type $pred \rightarrow pred$ we define an operator fix by

$$fix\_def \vdash fix\ f = glb\ (\lambda p.\ ((f\ p)\ implies\ p))$$

and prove (see Section 4.5 below) that if f is monotonic then $fix\ f$ is the least fixpoint of f.

4.3. Chains, Limits and Continuity

In order to formalise the concepts of bounded and unbounded nondeterminism we need the notion of continuous functions on predicates. This in turn requires a formalisation of chains and limits.

A sequence of predicates $Q_1, Q_2, \ldots$ is formalised as a function Q from natural numbers to predicates, i.e., it has type $num \rightarrow pred$. Chains (ascending sequences), limits and continuity are formalised by the following definitions:

$$chain\_def \vdash chain\ Q = \forall n.\ ((Q\ n)\ implies\ (Q\ (SUC\ n)))$$
$$limit\_def \vdash limit\ Q = \lambda s.\ \exists n.\ Q\ n\ s$$
$$continuous\_def \vdash continuous\ f = \forall Q.\ (chain\ Q \Rightarrow (f\ (limit\ Q) = limit\ (\lambda n.\ f\ (Q\ n))))$$

4.4. Theorems

We now present a few theorems of the theory *pred* that we have proved in the HOL system. They indicate what kind of lemmas are needed in the proofs of refinement rules.

First, we show two theorems concerning substitutions in states. These are

$$bind\_twice\_thm \vdash bind\ d\ x\ (bind\ d'\ x\ s) = bind\ d\ x\ s$$

$$bind\_bind\_thm \vdash \neg(x = y) \Rightarrow (bind\ d\ x\ (bind\ d'\ y\ s) = bind\ d'\ y\ (bind\ d\ x\ s))$$

The first theorem states that of successive substitutions to the same variable, only the outermost counts. The second theorem states that substitutions to distinct variables can be made in any order. Both theorems are easily lifted to the case of substitutions in predicates.

In Section 4.5 we show how the second theorem is proved by first setting up the corresponding goal and then applying various tactics.

4.4.1. Theorems Concerning Predicates

We have proved a number of basic properties of the operators on predicates. Among these are the following, which all are straightforward to prove:

1. Idempotence, commutativity, associativity and absorption of the operators *and* and *or*. Thus *pred* is a lattice.
2. That the *implies* relation is a partial order (i.e., it is reflexive, antisymmetric and transitive). It is the partial order induced by the lattice structure.
3. Theorems showing that *true* (*false*) is the top (bottom) element and that *not* is the boolean inverse operator on predicates.

As mentioned above, we have shown that *glb* is actually a greatest lower bound operator. This is shown by proving (in HOL) the following theorems:

$$glb\_bound\_thm \vdash P\ p \Rightarrow (glb\ P)\ implies\ p$$

$$glb\_greatest\_thm \vdash (\forall p.\ (P\ p \Rightarrow q\ implies\ p)) \Rightarrow (q\ implies\ (glb\ P))$$

The corresponding theorems for *lub* are similar.

We can then show that *fix* is the least fixpoint operator for monotonic functions on predicates, by proving the following two theorems (*pmonotonic* has been defined to formalise the concept of monotonic function on predicates):

$$fix\_fp\_thm \vdash pmonotonic\ f \Rightarrow (f\ (fix\ f) = fix\ f)$$

$$fix\_least\_thm \vdash (f\ p)\ implies\ p \Rightarrow (fix\ f)\ implies\ p$$

We have also proved the following theorem, useful for showing that a predicate is indeed the least fixpoint of a function.

$$fix\_char\_thm \vdash pmonotonic\ f \wedge (f\ q)\ implies\ q \wedge (\forall p.\ ((f\ p)\ implies\ p \Rightarrow q\ implies\ p)) \Rightarrow (q = fix\ f)$$

4.5. Example Proof of a Theorem

We show the proof of the theorem *bind_bind_thm* of Section 4.4. All the lines starting with the prompt character # are supplied by the user, while all the

other lines are the HOL system's responses. We have replaced the HOL syntax ASCII characters for logical symbols with the ordinary logical symbols to make the dialogue more readable. Note that in the HOL syntax, *the scope of quantifiers extends as far to the right as possible,* thus quantifiers bind weaker than other connectives.

After starting up the HOL system and setting up the theory of predicates as the current theory we set up the goal:

```
# set_goal([ ], "∀m n y z s.
  ¬(y = z) ⇒ (bind m y (bind n z s) = bind n z (bind m y s))");;
```

Now we expand the goal, using first the *STRIP_TAC* tactic, which is repeated until it fails, and then the *REWRITE_TAC* tactic with the definition of the constant *bind*:

```
# expand(REPEAT STRIP_TAC THEN REWRITE_TAC[bind_def]);;
```

Since expansion with *STRIP_TAC* strips universal quantifiers and undischarges assumptions, this results in the following sequent (the assumptions are printed below in square brackets):

```
"(λv. ((v = y) ⇒ m | (λv. ((v = z) ⇒ n | s v))v))
  = λv. ((v = z) ⇒ n | (λv. ((v = y) ⇒ m | s v))v)"
    ["¬(y = z)"]
```

The goal now is an equality between two functions. We expand further, using the conversion *FUN_EQ_CONV* which converts a term *"f = g"* to the theorem $\vdash (f = g) = (\forall x.\ f\,x = g\,x)$ (a conversion is a function which takes a term as an argument and returns a theorem). We also use *BETA_TAC* which makes beta-reductions.

```
# expand(CONV_TAC FUN_EQ_CONV THEN BETA_TAC
#THEN GEN_TAC);;
OK..
"((x = y) ⇒ m | ((x = z) ⇒ n | s x))
  = ((x = z) ⇒ n | ((x = y) ⇒ m | s x))"
    ["¬(y = z)"]
```

Now we apply the tactic *ASM_CASES_TAC* which makes a case split using the argument *"x = y"*. Rewriting with the assumptions finishes off the proof in both cases:

```
# expand(ASM_CASES_TAC "x = y" THEN ASM_REWRITE_TAC[ ]);;
OK..
goal proved
. |- ((x = y) ⇒ m | ((x = z) ⇒ n | s x))
   = ((x = z) ⇒ n | ((x = y) ⇒ m | s x))
. |- (λv. ((v = y) ⇒ m | (λv. ((v = z) ⇒ n | s v))v))
   = λv. ((v = z) ⇒ n | (λv. ((v = y) ⇒ m | s v))v)
|- ∀m n y z s. ¬(y = z) ⇒ (bind m y (bind n z s) = bind n z (bind m y s))

Previous subproof:
goal proved
```

When the proof is finished the HOL system prints all the goals of the goal stack that were proved by the last expansion (assumptions are printed as dots).

The important one is the goal on the top of the stack, the initial goal. We save this theorem, giving it the name *bind_bind_thm* and at the same time binding it to the same name in the current environment:

```
# let bind_bind_thm = save_top_thm 'bind_bind_thm';;
```

5. The Theory of Commands

We now create a new theory, representing the statements of our specification language by a new recursive type *cmd*. We call the statements in our formalisation "commands", thus making a distinction between the statements and their formalisation in the theory in HOL. Our theory has the theory of predicates as a parent (in addition to the basic HOL theory).

5.1. Commands and Their Weakest Preconditions

The HOL system has a package for defining concrete recursive types [Mel89]. The user inputs a kind of BNF grammar for the type and the system then automatically proves some basic theorems about the type (e.g., uniqueness of representation and the principle of structural induction). We use this package to give a syntactic definition for our programming language. The commands are written in our formalisation as follows:

assert b	for	$\{b\}$
assign x e	for	$x := e$
update x p	for	$x.p$
seq c c'	for	$c; c'$
if b b' c c'	for	**if** $b \to c \,[]\, b' \to c'$ **fi**
do b c	for	**do** $b \to c$ **od**
block x c	for	[**var** x; c]

The syntax of a command constructor determines its type. For example the constant *assert* has type $pred \to cmd$ showing that if b is a predicate then *assert b* is a command.

5.1.1. *Definition of Weakest Precondition*

The weakest precondition semantics is given by introducing a recursively defined function *wp* of type $cmd \to pred \to pred$. The weakest preconditions of the different basic commands are thus defined as follows.

```
wp (assert b) q = b and q
wp (assign x e) q = subst e x p
wp (update x p) q = (exists x p) and (forall x (p imp q))
wp (seq c c') q = wp c (wp c' q)
wp (if b b' c c') q = (b or b') and (b imp (wp c q)) and (b' imp (wp c' q))
wp (do b c) q = fix (λp. ((b and (wp c p)) or ((not b) and )))
wp (block x c) q = forall x (wp c (forall x q))
```

(these definitions actually constitute the single defining theorem *wp_def*, see Section 5.2).

Now the healthiness concepts can be defined: strictness, monotonicity, conjunctivity and boundedness.

strict_def ⊢ *strict c = (wp c false = false)*
monotonic_def ⊢ *monotonic c = ∀p q. (p implies q ⇒ (wp c p) implies (wp c q))*
conjunctive_def ⊢ *conjunctive c = ∀p q. (wp c (p and q) = (wp c p) and (wp c q))*
bounded_def ⊢ *bounded c = continuous (wp c)*

Finally, we define a function *free_in* recursively over the commands. It is defined so that *x free_in c* holds whenever the command *c* mentions the variable *x* or some predicate or expression mentioned in *c* depends on *x*. However, the local variable in a block is not free in the block. Section 5.2 shows how definitions are given to the HOL system.

5.1.2. Structural Induction

The package for defining recursive types provides an automatic proof of a structural induction theorem. For the type *cmd*, this theorem states that any property that holds for the primitive commands (assign, update and assert) and that is preserved by the constructors (sequential composition, conditional composition and iteration) holds for all commands. This theorem is given the name *cmd_induct*.

5.2. Setting Up the Theory of Commands in HOL

We now show how the theory of commands is created. First, we give the new theory the name *ref*. It automatically has the basic theory HOL as a parent. To this we add *pred*, the theory of predicates.

```
#new_theory 'ref';;
#new_parent 'pred';;
```

Now the type of commands is defined recursively. The first argument to the function *define_type* is the name of the primitive recursion theorem for the new type while the second argument is a BNF-style expression defining the type (the *cmd* to the left of the equal sign is the name of the new type).

```
#let cmd = define_type 'cmd'
#   'cmd = assert pred
#        | assign var expr
#        | update var pred
#        | seq cmd cmd
#        | if pred pred cmd cmd
#        | do pred cmd
#        | block var cmd';;
```

The HOL system returns the primitive recursion theorem (with the name *cmd*). We do not show this theorem as we will not use it.

Next the theorem of structural induction for commands is proved automatically.

```
# let cmd_induct = prove_induction_thm cmd;;
cmd_induct =
⊢ ∀P.
(∀b. P(assert b)) ∧
(∀x p. P(assign x p)) ∧
(∀x p. P(update x p)) ∧
(∀c c'. P c ∧ P c' ⇒ P(seq c c')) ∧
(∀c c'. P c ∧ P c' ⇒ (∀b' b. P(if b b' c c'))) ∧
(∀c. P c ⇒ (∀b. P(do b c))) ∧
(∀c. P c ⇒ (∀x. P(block x c))) ⇒
(∀c. P c)
```

The function *wp* is defined recursively, with the HOL system checking that the definition is consistent. The arguments to the function *new_recursive_definition* are a flag (*false* shows that *wp* is not an infix), the name of the primitive recursion theorem of the underlying type, the name of the definition and finally a term that gives the definition (we do not show the HOL system's reply, as it simply restates the definition as a theorem).

```
# let wp_def = new_recursive_definition false cmd 'wp_def'
#   "(wp (assert b) q = b and q) ∧
#   (wp (assign x e) q = λs. q (bind (e s) x s)) ∧
#   (wp (update x p) q = (exists x p) and (forall x (p imp q))) ∧
#   (wp (seq c c') q = wp c (wp c' q)) ∧
#   (wp (if b b' c c') q =
#     (b or b') and (b imp (wp c q)) and (b' imp (wp c' q))) ∧
#   (wp (do b c) q = fix (λp. (b and (wp c p)) or ((not b) and q))) ∧
#   (wp (block x c) q = forall x (wp c (forall x q)))";;
```

Now the four healthiness properties are formalised.

```
# let strict_def = new_definition('strict_def', "strict c = (wp c false = false)");;
# let monotonic_def = new_definition('monotonic_def',
#   "monotonic c = ∀p q. p implies q ⇒ (wp c p) implies (wp c q)");;
# let conjunctive_def = new_definition('conjunctive_def',
#   "conjunctive c = ∀p q. (wp c (p and q) = (wp c p) and (wp c q))");;
# let bounded_def = new_definition('bounded_def',
#   "bounded c = continuous (wp c)");;
```

5.3. Derived Commands

Derived commands, e.g., *skip* and *abort* are simple to define:

```
skip_def  ⊢  skip = assert true
abort_def ⊢  abort = assert false
```

We prove that they have the following weakest preconditions:

```
skip_thm  ⊢  wp skip q = q
abort_thm ⊢  wp abort q = false
```

General sequential composition, conditional composition and iteration are defined recursively. We formalise a sequence of commands as a function of type $num \rightarrow cmd$. A finite sequential composition $c_1; c_2; \ldots; c_n$ is formalised by the constant *Seq* of type $num \rightarrow (num \rightarrow cmd) \rightarrow cmd$ (the first argument indicating the length of the sequence), by the following defining theorem:

$$\begin{aligned} Seq\_def \vdash\ & (Seq\ 0\ C = skip) \wedge \\ & (Seq\ (SUC\ n)\ C = seq\ (Seq\ n\ C)\ (C\ (SUC\ n))) \end{aligned}$$

Thus an empty sequence is equal to *skip*. This is natural since **skip** is the identity element for sequential composition in our specification language.

To define general conditional composition we have to define the disjunction of a finite sequence of predicates. This is done by a construction, similar to that of the sequential composition above. Sequences of predicates have type $num \rightarrow pred$ and we define the constant *gg* as

$$\begin{aligned} gg\_def \vdash\ & (gg\ 0\ B = false) \wedge \\ & (gg\ (SUC\ n)\ B = (gg\ n\ B)\ or\ (B\ (SUC\ n))) \end{aligned}$$

Now general conditional composition is formalised by the constant *If*; the defining theorem is:

$$\begin{aligned} If\_def \vdash\ & (If\ 0\ B\ C = abort) \wedge \\ & (If\ (SUC\ n)\ B\ C = if\ (gg\ n\ B)\ (B\ (SUC\ n))\ (If\ n\ B\ C)\ (C\ (SUC\ n))) \end{aligned}$$

General iteration is defined in terms of general conditional composition, formalised by the constant *Do*:

$$Do\_def \vdash Do\ n\ B\ C = do\ (gg\ n\ B)\ (If\ n\ B\ C)$$

It is straightforward to show that *Seq*, *If* and *Do* are extensions of *seq*, *if* and *do* by proving the following theorems

$$\begin{aligned} & \vdash Seq\ 2\ C = seq\ (C\ 0)\ (C\ 1) \\ & \vdash If\ 2\ B\ C = if\ (B\ 0)\ (B\ 1)\ (C\ 0)\ (C\ 1) \\ & \vdash Do\ 1\ B\ C = do\ (B\ 0)\ (C\ 0) \end{aligned}$$

We also define the two-way iteration command, giving it the name *do2*. The defining theorem is

$$do2\_def \vdash do2\ b\ b'\ c\ c' = do\ (b\ or\ b')\ (if\ b\ b'\ c\ c'))$$

and it is straightforward to prove that this is a special case of *Do*:

$$\vdash Do\ 2\ B\ C = do2\ (B\ 0)\ (B\ 1)\ (C\ 0)\ (C\ 1)$$

5.4. Bounded Nondeterminism

Within our formalisation we have proved that Dijkstra's classical definition for the weakest precondition of *do* for the bounded case (2) holds. The *H*-function is formalised by the constant *H* with type $num \rightarrow pred \rightarrow cmd \rightarrow pred \rightarrow pred$ given by the following defining theorem:

$$\begin{aligned} H\_def \vdash\ & (H\ 0\ b\ c\ q = (not\ b)\ and\ q) \wedge \\ & (H\ (SUC\ n)\ b\ c\ q = ((not\ b)\ and\ q)\ or \\ & (wp\ (seq\ (assert\ b)\ c)\ (H\ n\ b\ c\ q))) \end{aligned}$$

and the theorem giving *wp* for *do* in the bounded case is

$$do\_bounded\_thm \vdash bounded\ c \Rightarrow (wp\ (do\ b\ c) = \lambda s.\ \exists n.\ H\ n\ b\ c\ q\ s)$$

We have also shown that *assert* and *assign* are bounded and that boundedness is preserved by *seq*, *if* and *do*. Thus unbounded nondeterminism can be introduced only by *update* and *block*.

5.5. Healthiness Conditions

We have proved the three healthiness properties for (almost) all commands using the HOL system. These are

1. Strictness of all commands.
2. Monotonicity of all commands.
3. Conjunctivity of all commands (actually, we have not managed to prove this for *do* in the unbounded case).

The following frame condition, stating that *wp* preserves the variable environment of its arguments, has also been proved:

$$frame\_thm \vdash c\ cmd\_on\ v \wedge q\ pred\_on\ v \Rightarrow (wp\ c\ q)\ pred\_on\ v \qquad (3)$$

where $c\ cmd\_on\ v$ means that every free variable of the command c is in the variable set v. The frame condition is important when reasoning about local variables in blocks.

All of these healthiness conditions have been proved using structural induction. The proofs of strictness and monotonicity are quite straightforward (although the monotonicity of iteration is not trivial). Conjunctivity is quite tricky to prove for iteration in the case of bounded nondeterminism, requiring double induction on the natural numbers. Conjunctivity for iteration in the case of unbounded nondeterminism seems to require transfinite induction. Proving it would require a theory of ordinals and quite elaborate results from lattice theory. The frame theorem is obvious on a syntactic level. However, in our formalisation it requires a proof which is not trivial.

6. The Refinement Relation

We now turn to the refinement relation between commands. We first define the refinement relation in the ordinary way, i.e., as the pointwise extension of the *implies* relation on predicates. As we assume a fixed variable environment we cannot in general assume that local variables are drawn from outside this fixed set of variables. This means that we have to define a second refinement relation which takes into account the global set of variables within which the refinement holds.

6.1. Refinement without Variable Environments

6.1.1. Definition of the General Refinement Relation

The general refinement relation *ref* is an infix of type $cmd \rightarrow cmd \rightarrow bool$. The defining theorem is

$$ref\_def \vdash c\ ref\ c' = \forall q.\ ((wp\ c\ q)\ implies\ (wp\ c'\ q))$$

The refinement relation is easily shown to be a preorder, i.e., reflexive and transitive. The equivalence relation $\equiv$ induced by the refinement relation is formalised as *req*:

$$\vdash req\_def \; c \; req \; c' = c \; ref \; c' \wedge c' \; ref \; c$$

6.1.2. *Refinement Rules*

We have proved a number of simple refinement rules in our formalisation. In the following we show some of these rules.

1. Rules stating that *abort* is refined by any command and that a *skip* command can be dropped from any sequence of commands,

$$\vdash abort \; ref \; c$$
$$\vdash (seq \; skip \; c) \; req \; c$$
$$\vdash (seq \; c \; skip) \; req \; c$$

2. Rules for deleting, weakening and adding context assertions,

$$\vdash (assert \; b) \; ref \; skip$$
$$\vdash (b \; implies \; b') \Rightarrow (assert \; b) \; ref \; (assert \; b')$$
$$\vdash (wp \; c \; q = true) \Rightarrow c \; ref \; (seq \; c \; (assert \; q))$$

3. The rule for loop unfolding,

$$\vdash (do \; b \; c) \; req \; (if \; b \; (not \; b) \; (seq \; c \; (do \; b \; c)) \; skip)$$

4. The rule for explicitly choosing an enabled branch of an if-command,

$$\vdash (seq \; (assert \; b) \; (if \; b \; b' \; c \; c')) \; ref \; c$$

and the corresponding rule for the *do2*-command:

$$\vdash (seq \; (assert \; b) \; (do2 \; b \; b' \; c \; c')) \; ref \; (seq \; c \; (do2 \; b \; b' \; c \; c'))$$

In the proof of the last rule, we use both the loop unfolding rule and the rule for choosing an enabled branch of an if-command. This is an example of how a hierarchy of refinement rules can be built up.

6.2. Refinement with Respect to a Fixed Variable Environment

An important aspect of program development in the refinement calculus is the introduction and elimination of local variables. One useful rule states that if the statement S does not mention the variable x then the refinement equivalence $S \equiv [\mathbf{var}\, x; S]$ holds. However, it does not hold in the formalization given above. Instead, we have to consider refinement with respect to some specific set of global variables to be able to deal with local variables.

6.2.1. *Definition of the Refinement Relation*

The refinement relation with respect to some fixed variable environment is called *refv* and it has type $(var \rightarrow bool) \rightarrow cmd \rightarrow cmd \rightarrow bool$. The defining

theorem is

$$\vdash refv\_def \vdash refv\ v\ c\ c' = (c\ cmd\_on\ v) \wedge (c'\ cmd\_on\ v) \wedge (\forall q.\ q\ pred\_on\ v \Rightarrow (wp\ c\ q)\ implies\ (wp\ c'\ q))$$

where $c\ cmd\_on\ v$ is defined to hold whenever the set v contains all free variables of c. Thus *refv* is a refinement relation within a fixed variable environment v. It holds between commands c and c' iff

1. the variables of both c and c' are in v, and
2. for all predicates q on v, $wp\ c\ q$ implies $wp\ c'\ q$.

The refinement relation *refv* is easily shown to be reflexive and transitive. Its corresponding equivalence relation is called *reqv*.

The relation between *ref* and *refv* is shown by the following two theorems:

$$\vdash c\ ref\ c' = refv\ (\lambda x.T)\ c\ c'$$
$$\vdash (c\ cmd\_on\ v) \wedge (c'\ cmd\_on\ v) \wedge (c\ ref\ c') \Rightarrow refv\ v\ c\ c'$$

Here $(\lambda x.T)$ is the variable environment containing all variables. Thus the first theorem states that general refinement is the same as refinement with respect to the environment containing all variables. The second theorem states that within a fixed variable environment, *ref* implies *refv*.

6.2.2. *Refinement Rules*

The refinement rules proved for *ref* hold for *refv* also. Furthermore, we have proved a rule for introducing local variables, stating that a variable x not belonging to the variable environment v can be introduced:

$$\vdash (c\ cmd\_on\ v) \wedge (\neg v\ x) \Rightarrow refv\ v\ c\ (block\ x\ c))$$

The proof of this rule rests on the frame condition (3) of Section 5.5.

6.3. Example Proof of a Refinement Rule

As an example of how refinement rules are proved we show the proof of the rule for explicitly choosing an enabled branch of an if-command.

First the goal is set up:

```
#set_goal([ ],"∀b b' c c'. (seq (assert b) (if b b' c c')) ref c");;
"∀b b' c c'. (seq (assert b) (if b b' c c')) ref c"
```

Now we do a rewriting, using the definitions of *ref* and *wp*. This moves us down to the level of predicates.

```
#expand(REWRITE_TAC[ref_def;wp_def]);;
OK..
"∀ b b' c c'.
   (b and ((b or b') and ((b imp (wp c q)) and ((b' imp (wp c' q)))))
      implies (wp c q)"
```

Next we could rewrite using the definitions of *and*, *or*, *imp* and *implies*, moving down to the level of *bool*. However, we will instead use a tautology

prover for predicates (we have implemented this as the tactic PRED_TAUT_TAC).

```
#expand(PRED_TAUT_TAC);;
OK..
goal proved
```

Thus the theorem is proved.

We see that the proof of this refinement rule was quite simple. This is a general observation: it is often easy to prove refinement rules that are "propositional" in the sense that they do not speak about particular variables, expressions or predicates. We will see below that the same does not hold when working with actual program texts.

7. Proving Refinements of Actual Program Texts

The refinement rules proved in the previous section were general rules, not involving any specific variables or data. In this section we show how refinements of actual program texts can be made using the formalisation of the refinement calculus presented in this paper.

When doing refinements to actual program texts we have to work with the properties of data types. Our example uses natural numbers, and we can use the existing theory *num* that is a part of the basic HOL theory.

7.1. Formalising a Program Refinement

Assume that X, Y and Z are program variables (we use capital letters to avoid confusion between program variables and meta-variables). We want to show that the following refinement equivalence holds:

$$X := X + Y; X := X + Z \equiv X := X + Y + Z$$

Although this is a very simple fact, formalising and proving it will show quite well how the formalisation is used in practice and some problems in connection with refining actual program texts.

We first have to express the problem in our formalisation. We want to prove the following:

```
(seq (assign X λs.(s X + s Y)) (assign X λs.(s X + s Z)))
  req (assign X λs.(s X + s Y + s Z))
```

(note that we have to introduce the state s) so we set this up as our goal:

```
#set_goal([ ], "(seq(assign X(λs.s X + s Y))(assign X(λs.s X + s Z))) req
# (assign X(λs.s X + s Y + s Z))");;
"(seq(assign X(λs. (s X) + (s Y)))(assign X(λs. (s X) + (s Z)))) req
  (assign X(λs. (s X) + ((s Y) + (s Z))))"
```

7.2. Proving the Refinement

The proof mainly consists of rewriting using the definitions of *req*, *wp* and the associativity of addition. We assume that the following lemmas have been proved and are available:

1. The theorem *req_thm*:

$$\vdash c\ req\ c' = \forall q.\ (wp\ c\ q = wp\ c'\ q)$$

stating that two commands are refinement equivalent if and only if their weakest precondition predicate transformers are equal.

2. The theorem *fun_eq_lemma*:

$$\vdash (\forall f.\ (f\,x = f\,y)) = (x = y)$$

where the function f has the polymorphic type $* \rightarrow bool$ (this theorem is a formulation of Leibnitz's principle of identity).

3. The theorem *var_distinct*, which states that variables with distinct names are distinct.

4. The theorem *ADD_ASSOC*:

$$\vdash m + (n + p) = (m + n) + p$$

stating the associativity of addition on the natural numbers.

The third of these theorems is proved automatically when defining the type of variables, while the fourth one is a pre-proved theorem (part of the theory *num* in the basic HOL theory).

We begin by rewriting using *req_thm* and the definition of *wp*, beta-reducing, removing universal quantifiers (*GEN_TAC*) and using extensional equality of functions (we have defined the tactic *FUN_TAC* so that it reduces a goal $f = g$ to the goal $f\,x = g\,x$, doing a beta-reduction if it is possible).

```
#expand(REWRITE_TAC[req_thm;wp_def] THEN BETA_TAC
#THEN GEN_TAC THEN FUN_TAC);;
OK..
"q (bind ((bind ((s X) + (s Y))X s) X + (bind ((s X) + (s Y)) X s) Z) X
        (bind((s X) + (s Y))X s))
  = q(bind ((s X) + ((s Y) + (s Z))) X s)"
```

Now we generalise on the predicate q and rewrite using *fun_eq_lemma*. This way q disappears from the goal.

```
#expand(SPEC_TAC("q","q") THEN REWRITE_TAC[fun_eq_lemma]);;
OK..
"bind ((bind ((s X) + (s Y)) X s)X + (bind ((s X) + (s Y))X s)Z) X
      (bind ((s X) + (s Y)) X s)
  = bind ((s X) + ((s Y) + (s Z))) X s"
```

Next we rewrite using the definition of *bind* and simplify the resulting expression by beta-reducing and using extensional equality of functions.

```
#expand(REWRITE_TAC[bind_def] THEN BETA_TAC
#THEN FUN_TAC);;
OK..
"((x = X) ⇒ ((s X) + (s Y)) + ((Z = X) ⇒ (s X) + (s Y) | s Z) | ((x = X)
  ⇒ (s X) + (s Y) | s x))
  = ((x = X) ⇒ (s X) + ((s Y) + (s Z)) | s x)"
```

Now follows a case split on $(x = X)$, then rewriting using the assumptions and

the fact that X and Z are distinct (the case $\neg(x = X)$ gives the goal $s\,x = s\,x$ which is proved immediately, so only the case $(x = X)$ remains).

```
#expand(ASM_CASES_TAC "x = X"
#THEN ASM_REWRITE_TAC[var_distinct]);;
OK..
"((s X) + (s Y)) + (s Z) = (s X) + ((s Y) + (s Z))"
    [ "x = X" ]
```

The final step is a rewriting using the associativity of addition.

```
#expand(REWRITE_TAC[ADD_ASSOC]);;
OK..
goal proved
. . .
⊢ (seq(assign X(λs. (s X) + (s Y)))(assign X(λs. (s X) + (s Z)))) req
     (assign X(λs. (s X) + ((s Y) + (s Z))))
```

We chose the above example in order to show some difficulties associated with reasoning about assignments in our formalisation. The problems stem partly from our semantic definitions of expressions and predicates, partly from the fact that program variables are somewhat different from ordinary variables of the logic (see the discussion in [Mas87]).

7.3. Refining a Subcomponent of a Program

The subcomponent monotonicity property permits us to replace any subcomponent S of a program with another component S' that refines S. This property cannot be proved universally in our HOL formalisation, since we cannot express the relation of "being a subcomponent of". However, we can write a conversion which proves this property for any particular case.

We specify a subcomponent of a program by rewriting the program as a beta-redex, with the bound variable standing for the subcomponent in question. Thus, if $c1$ and $c1'$ are commands and we have proved the theorem

$$\vdash c1 \mathbin{ref} c1'$$

and we want to replace $c1$ with $c1'$ in a program, e.g., $do\ b\ (seq\ c1\ c2)$, we can proceed as follows. We first rewrite the context as a beta-redex:

$$"(\lambda c.\ do\ b\ (seq\ c\ c2))c1"$$

and then we apply our subcomponent monotonicity proving conversion to this term, yielding the theorem

$$\vdash \forall c\ c'.\ (c \mathbin{ref} c' \Rightarrow (do\ b\ (seq\ c\ c2)) \mathbin{ref} (do\ b\ (seq\ c'\ c2))$$

Instantiating this theorem with $c1$ and $c1'$ and applying modus ponens yields the refinement we wanted to prove:

$$\vdash (do\ b\ (seq\ c1\ c2)) \mathbin{ref} (do\ b\ (seq\ c1'\ c2))$$

In Section 6 we proved some elementary refinement rules. Now we see how these rules can be applied to parts of a program. Since the refinement relation has been proved to be transitive, our formalisation permits us to perform a

sequence of such small subcomponent replacements. Thus our formalisation supports the basic idea of the refinement calculus: program development by provably correct stepwise transformations of program components.

8. Conclusions and Future Work

We have shown how the refinement calculus can be formalised in the HOL system (the HOL88 version). The formalisation was done in two steps. The first step is a formalisation of the theory of the complete boolean lattice of predicates over a given set of variables. The second step is the formalisation of commands, weakest preconditions and refinement. This was accomplished without introducing any new axioms, only definitions using existing concepts. Thus the theory is guaranteed to be consistent [Gor88]. In this formalisation we have proved a number of basic refinement rules, that are useful when doing refinements to actual program texts. We gave a small example, showing how a refinement of a program text is proved correct within our formalisation.

Case studies have shown that the refinement calculus is a powerful tool for developing algorithms from specifications and for transforming sequential algorithms into parallel algorithms. However, the correctness proofs are often long, involving large amounts of routine work with proof details. This shows the potential use of our formalisation of the refinement calculus; the mechanised proofs using the HOL system are guaranteed to be correct (provided that the definitions of the concepts capture their intended meaning).

The HOL system has proved to be suitable for our formalisation. The higher-orderness permits us to quantify over arbitrary types (e.g., predicates) and the type definition package permits us to define our specification language in an efficient and natural way. Using the language of tactics and tacticals we can program proof strategies that are used repeatedly, thus making the proofs of refinements more automatic.

The specification language we have used is more restricted than the refinement calculus language of Back and Wright [BaW89b]. In particular, we cannot express multiple assignments or recursion. Also we have not permitted miraculous commands or angelic nondeterminism, which have proved useful in data refinement. Such generalisations of the language will be investigated in future work.

The syntax of our formalisation is not very user-friendly. It would be nice to have a parser-pretty-printer that would allow the user to enter program texts using the ordinary syntax. Such an interface is possible to make, but at present it has to be tailored to the specific specification language syntax used and (for the present version of the HOL system) it must be coded in LISP. An example of a similar interface is described in [Gor89]. It seems possible that future versions of the HOL system would permit a more flexible syntax.

The example in Section 7 shows that even a simple refinement may require quite a long proof. This is the case especially when assignments are involved, since we have to reason about expressions and substitutions. However, as the examples of general refinement rules and of subcomponent replacement showed, reasoning is easier when it comes to program structures, not referring to particular expressions or predicates.

To be of practical use, our work needs to be extended in a number of directions. The programming language should be extended to permit e.g., multiple assignments. A library of lemmas about predicates, expressions and substitutions and a hierarchy of transformation rules, ranging from very specific refinement rules to very general ones, should be built. A user-friendly interface should permit ordinary program syntax and easy ways of indicating what rules should be applied to what parts of the program text. This will be the focus of future work.

References

[Bac78] Back, R. J. R.: On the Correctness of Refinement in Program Development, Ph.D. thesis Report A-1978-4, Department of Computer Science, University of Helsinki, 1978.

[Bac80] Back, R. J. R.: *Correctness Preserving Program Refinements: Proof Theory and Applications,* Vol. 131 of *Mathematical Centre Tracts,* Mathematical Centre, Amsterdam, 1980.

[Bac88] Back, R. J. R.: A Calculus of Refinements for Program Derivations. *Acta Informatica,* 25, 593–624 (1988).

[Bac89] Back, R. J. R.: Refining Atomicity in Parallel Algorithms. In: *PARLE Conference on Parallel Architectures and Languages Europe,* Eindhoven, the Netherlands, Springer Verlag, 1989.

[BaS89] Back, R. J. R. and Sere, K.: Stepwise Refinement of Action Systems. In: *Mathematics of Program Construction,* Lecture Notes in Computer Science 375, Springer-Verlag, 1989.

[BaW89a] Back, R. J. R. and Wright, J. von: A Lattice-Theoretical Basis for a Specification Language. In: *Mathematics of Program Construction,* Lecture Notes in Computer Science 375, Springer-Verlag, 1989.

[BaW89b] Back, R. J. R. and Wright, J. von: Refinement Calculus, part I: Sequential Programs. In: *REX Workshop for Refinement of Distributed Systems,* Nijmegen, The Netherlands, 1989.

[GMR88] Gardiner, P. H., Morgan, C. C. and Robinson, K. A.: On the Refinement Calculus. Techn. Rep. PRG 70, Programming Research Group, Oxford University, 1988.

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

[DiG86] Dijkstra, E. W. and Gasteren, A. J. M. van: A Simple Fixpoint Argument without the Restriction to Continuity. *Acta Informatica,* 23, 1–7 (1986).

[GaM88] Gardiner, P. H. and Morgan, C. C. Data refinement of predicate transformers. *Theoretical Computer Science.* (To appear.)

[Gor88] Gordon, M. J. C.: HOL: A Proof Generating System for Higher-Order Logic. In: *VLSI Specification, Verification and Synthesis,* G. Birtwistle and P. A. Subrahmanyam (eds), Kluwer Academic Publishers, 1988.

[Gor89] Gordon, M. J. C.: Mechanizing Programming Logics in Higher-Order Logic. In: *Current Trends in Hardware Verification and Theorem Proving,* G. Birtwistle and P. A. Subrahmanyam (eds), Springer-Verlag, 1989.

[Gri81] Gries, D.: *The Science of Programming,* Springer-Verlag, 1981.

[Mas87] Mason, I. A.: Hoare's Logic in the LF. Techn. Rep. 87-32, Laboratory for Foundations of Computer Science, University of Edinburgh, 1987.

[Mel89] Melham, T.: Automating Recursive Type Definitions in Higher Order Logic. In: *Current Trends in Hardware Verification and Theorem Proving,* G. Birtwistle and P. A. Subrahmanyam (eds), Springer-Verlag, 1989.

[MGW79] Milner, R., Gordon, M. J. C. and Wadsworth, C.: Edinburgh LCF: A mechanised logic of computation. Lecture Notes in Computer Science 78, Springer-Verlag, 1979.

[Mor90] Morgan, C. C.: *Programming from Specifications,* Prentice-Hall, 1990.

[Mor87] Morris, J. M.: A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming,* 9, 287–306 (1987).

[Par80] Park, D.: On the semantics of fair parallelism. In: Lecture Notes in Computer Science 86, pp. 504–526, Springer-Verlag, 1980.

[Sok87] Sokolowski, S.: Soundness of Hoare's Logic: An Automated Proof Using LCF. *ACM Transactions on Programming Languages and Systems*, 9(1), 100–120 (1987).

[Tar55] Tarski, A.: A Lattice Theoretical Fixed Point Theorem and its Applications. *Pacific J. Mathematics*, 5, 285–309 (1955).

[Wri89] Wright, J. von: Stepwise Derivation of a Parallel Matrix Multiplication Algorithm. *Reports on Computer Science and Mathematics* 84, Åbo Akademi, 1989.

Received October 1989
Accepted in a revised form April 1990 by J. V. Tucker