Formal Methods in System Design, 3: 49–81 (1993)
© 1993 Kluwer Academic Publishers

Mechanizing Some Advanced Refinement Concepts

J. VON WRIGHT
University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, England

J. HEKANAHO, P. LUOSTARINEN AND T. LÅNGBACKA
Dept. of Computer Science, Åbo Akademi University, Lemminkäisenkatu 14, SF-20520 Turku, Finland

Abstract. We describe how the HOL theorem prover can be used to check and apply rules of program refinement. The rules are formulated in the refinement calculus, which is a theory of correctness preserving program transformations. We embed a general command notation with a predicate transformer semantics in the logic of the HOL system. Using this embedding, we express and prove rules for data refinement and superposition refinement of initialized loops. Applications of these proof rules to actual program refinements are checked using the HOL system, with the HOL system generating these conditions. We also indicate how the HOL system is used to prove the verification conditions. Thus, the HOL system can provide a complete mechanized environment for proving program refinements.

Keywords: program verification, semantics of programming languages, deduction and theorem proving

1. Introduction

Stepwise refinement is a methodology for developing programs from high-level program specifications into efficient implementations. In this approach to program development, the development of a program includes the proof of its correctness. This can be compared with program *verification*, where the program is first developed, using informal methods, and then checked for desired correctness properties. The *refinement calculus* [1, 2] is a formalization of the stepwise refinement approach, based on the weakest precondition calculus of Dijkstra [3]. The refinement calculus is a calculus of program transformations that preserve total correctness. If C and C' are commands (program fragments), then the refinement $C \leq C'$ holds if and only if C' satisfies every total correctness assertion that C satisfies.

The refinement relation is reflexive and transitive. Furthermore, the ordinary control structures of sequential programming (sequential composition, conditional composition, and iteration) are monotonic with respect to the refinement relation. This means that if the refinement $C \leq C'$ holds, then we can replace C by C' in any program context.

A refinement step $C \leq C'$ is often proved by appealing to a high-level refinement rule. Particularly important are rules which encode useful program development principles. One example of such a principle is data refinement (sometimes called data retification), where a data structure is replaced by another one in such a way that total correctness of the program is preserved. Data refinement has been extensively studied [4, 5, 6] and it is a central program development principle, e.g., in the VDM method [7]. Another high-level refinement principle is superposition, where a layer of computation is added on top of an existing program [8, 9, 10]. Rules for data refinement and superposition make it possible to develop programs from specifications in just a small number of major refinement steps.

In recent years, there has been a growing interest in using mechanized formal systems (theorem provers) in the design of software. Most of this work has been directed at verification where the correctness of an already-developed program is checked. Examples of this can be found in [11], using the Boyer-Moore prover, in [12], using the Larch theorem prover (LP), and in [13], using the Nuprl system. The HOL theorem prover, which has previously been used mostly for hardware verification [14], has also been used for program verification. Various programming notations have been semantically embedded in the HOL logic. Examples include a simple while-language [15], CSP [16], UNITY [17], and guarded command languages [18, 19].

To our knowledge, there have been very few attempts to construct tools that support program refinement, based on theorem proving. Jim Grundy has implemented a system to support a transformational style of reasoning in HOL. He has used it to construct a tool for refinement in a framework where programs are represented by predicates [20]. There are also a few examples of *refinement editors*, which support program refinement in the same style as we consider in this paper [21, 22]. However, these tools generally work on two independent levels. A verification condition generator produces formulas that must be checked for validity, in order for a given refinement to hold. These formulas are then checked using a separate system, e.g., by manual inspection or using a theorem prover.

The aim of this paper is to show that it is possible to use a theorem prover as a basis for a unified system for program refinement. Such a system would consist of three parts, all represented in the theorem prover. First, we need a programming notation, semantically embedded in the logic of the theorem prover. Second, we need a set of refinement laws, the correctness of which have been established by proving the laws as theorems in the logic. Third, we need infrastructure for handling program refinements within the logic. We describe a mechanization of some aspects of the refinement calculus in the HOL theorem prover.

We represent the semantics of a simple programming notation in HOL. As a basis for this we use previously reported work on mechanizing a command language with a weakest precondition semantics using the HOL system [19, 23]. The main contribution of this paper is the representation in HOL of three high-level refinement principles. These are *data refinement, backward data refinement,* and a special formulation of *superposition refinement*. For each of these principles, we have formulated proof rules and proved their correctness using HOL. We also show by means of small examples how these proof rules can be used to check actual program refinements using HOL. Thus, our intention is to show how the HOL system can be used both to prove the correctness of high-level refinement rules and to check applications of these rules in actual program developments.

2. The HOL system

The HOL system is an interactive theorem-proving environment for higher-order logic. Its logic is an extension of Church's Simple Type Theory [24], with type variables. Essentially, it extends typed first-order logic by permitting lambda expressions that denote functions. It also permits higher-order functions and quantification over arbitrary types.

All HOL terms must have a well-defined type. Atomic types are, e.g., `bool` (the booleans) and `num` (the natural numbers). Compound types can be constructed using the pairing operator $\times$ and the function space operator $\rightarrow$. HOL also permits polymorphic types, through the use of type variables. Type variables always begin with an asterisk $*$.

The user accesses HOL through an ML interface. By evaluating ML expressions, the user can create new theories, access existing theories, make definitions, and prove theorems.

An important feature of the HOL system is the amount of existing infrastructure for defining concepts and for proving theorems. Theorems can be proved by forward proof, applying inference rules (ML functions) to terms and existing theorems. The HOL system also supports goal-directed proof: the user sets up a goal and then proves it using goal-reducing functions called tactics. Elaborate tactics can be programmed in the ML language. There are a number of libraries in the HOL system. In these libraries, the user can find theorems about different data structures proved and ready to use, e.g., theorems about natural numbers, lists, trees, sets, etc.

A more detailed description of the HOL system and its logical foundations can be found in [14].

Notation. We use standard symbols for the logical connectives, and the symbols F and T for truth values (values of boolean type). Also, we let the scope of binders and quantifiers extend as far to the right as possible. We usually write function application without parentheses, and for higher-order functions application associates to the left. Thus $f\ x$ is the same as $f(x)$ and $f\ x\ y$ is the same as $(f\ x)y$.

HOL terms and interaction with the HOL system are written in `teletype` font. Theorems that have actually been proved using HOL are indicated by a leading turnstile symbol $\vdash$. To make formulas more readable, we generally omit type information that can be inferred from the context. We use small letters to denote the HOL representation of objects. For example, we write P or Q for predicates, and p or q for the HOL representation of predicates.

The reader should note the following features of the HOL syntax. Lists are written in square brackets, with semicolon separating the elements, e.g., [1; 2; 3]. `NIL` is the empty list and the list functions `HD`, `TL`, `APPEND`, and `LENGTH` have their obvious meaning.

3. Predicates and predicate transformers

The notion of refinement that we have represented in HOL is based on weakest preconditon semantics [3]. In such a semantics, the meaning of a command is a predicate transformer, i.e., a function from predicates to predicates. In order to define this semantics, we first explain what predicates are, and how we represent them in HOL.

3.1. States

Let u be an arbitrary list of distinct variable names. The *state space* $\sum_u$ over the variables u is a mapping which assigns a value to every variable in u. We may assume that every variable x is associated with a value set (the *type* of x). For example, $(b = T, x = 0, x = 1)$ is a state in $\sum_u$, where u is the list (b, x, y) and b is boolean while x and y range over the natural numbers.

In HOL, this state space is represented by the type `bool×num×num` and a state is represented by a tuple of values. Note that in the HOL representation there are no variable names. Instead, each position in a state tuple corresponds to a variable. Thus, the state mentioned above is simply represented as $(T, 0, 1)$.

When we work in HOL with state spaces in general, we assume that they have the polymorphic type `*s`. In particular cases, this type will be instantiated to a state tuple.

3.2. Predicates

For a given list of variables u, we define the *predicate space* $Pred_u$ to be the set of all functions from $\sum_u$ to the set of truth values $\{F, T\}$. The elements of $Pred_u$ are called *predicates of arity* u. We say that a predicate P *holds in the state* σ if $P\ \sigma\ =\ T$.

A predicate P is said to be *stronger* than a predicate Q if Q holds whenever P holds. This gives us a partial order $\leq$ ("stronger-than" or "implies") on $Pred_u$:

$$P \leq Q \overset{\mathrm{def}}{=} (\forall \sigma \, . \ P\sigma \Rightarrow Q\sigma)$$

We also lift logical connectives from booleans to predicates. For example, $P \wedge Q$ is the predicate which holds in a state σ if and only if both P and Q hold in σ. Negation, disjunction, and implication on predicates are defined similarly. The predicate which is everywhere false is called *false*. Similarly, the predicate which is everywhere true is called *true*.

The predicate space $Pred_u$ with the partial order $\leq$ is a *complete lattice*, which means that we can generalize conjunctions and disjunctions to be taken over arbitrary sets of predicates. For example, $\bigwedge_{i \in I} P_i$ is the weakest predicate which is stronger than all the predicates P_i, where i ranges over an arbitrary index set I.

Representing predicates in HOL. Since states have generic type `*s`, predicates have type `*s→bool`, which we abbreviate `(*s)pred`. As an example, consider the predicate $x < y$ on the same state space as in the example above. It is represented in HOL as the term

$$\lambda(\mathtt{b, x, y}).\ \mathtt{x} < \mathtt{y}$$

Similarly, any given predicate can be written as a term in higher-order logic, provided that we know what the underlying state space is.

Substitution in predicates is represented by a combination of function application and lambda abstraction. As an example, the substitution of 2 for x in the above predicate is represented as

$$\lambda(\mathtt{b, x, y}).\ (\lambda(\mathtt{b, x, y}).\ \mathtt{x} < \mathtt{y})(\mathtt{b}, 2, \mathtt{y})$$

which by the HOL rule of beta conversion is equal to

$$\lambda(\mathtt{b, x, y}).\ 2 < \mathtt{y}$$

This corresponds exactly to the syntactic substitution of 2 for x in $x < y$.

The partial order on predicates is defined in the obvious way:

$$\vdash_{def} \mathtt{p\ implies\ q} = \forall \mathtt{s}.\ \mathtt{p\ s} \Rightarrow \mathtt{q\ s}$$

We define the logical operators on predicates as indicated above. For example, the defining theorem for conjunction is as follows:

$$\vdash_{def} \mathtt{p\ and\ q} = \lambda \mathtt{s}.\ \mathtt{p\ s} \wedge \mathtt{q\ s}$$

where `s` has type `*s`. We define the operators `not`, `or`, `imp`, and the predicates `false` and `true` similarly.

We also define greatest lower bounds (generalized conjunctions) and least upper bounds (generalized disjunctions) over sets of predicates. Sets are represented by their characteristic functions so the defining theorems are as follows:

$$\vdash_{def} \mathtt{glb\ P} = \lambda \mathtt{s}.\ \forall \mathtt{p}.\ \mathtt{P\ p} \Rightarrow \mathtt{p\ s}$$
$$\vdash_{def} \mathtt{lub\ P} = \lambda \mathtt{s}.\ \exists \mathtt{p}.\ \mathtt{P\ p} \wedge \mathtt{p\ s}$$

where P has type (*s)pred→bool, i.e., it represents a set of predicates.

For simplicity, we have defined `and`, `or`, `imp`, and `implies` to be infix curried operators.

Chains and limits. A sequence $A = (a_0, a_1, \ldots)$ of elements of some type $*$ is formalized as a function `A:num→ *`. Thus the ith element a_i is represented by `A i`.

An ascending sequence of predicates is called a *chain*. The join of such a chain is called its *limit*. This leads us to make the following definitions:

$$\vdash_{def} \texttt{chain Q} = \forall\texttt{n. (Q n) implies (Q(n + 1))}$$
$$\vdash_{def} \texttt{limit Q} = \texttt{lub} (\lambda\texttt{q. }\exists\texttt{n. q = Q n})$$

Limits of chains are used in Section 5.2, to give an alternative characterization of the semantics of iteration.

3.3. Predicate transformers

A *predicate transformer* is a function that maps predicates to predicates. If the predicate transformer f has type $Pred_v \rightarrow Pred_u$, we say that f has *arity* $u \leftarrow v$. The reason for the reverse arrow notation is that such a predicate transformer f can be interpreted as a command executed in an initial state in $\sum_u$ and terminating in a final state in $\sum_v$ (see Section 4).

The conjunction and disjunction operators on predicates can be lifted to predicate transformers. Thus, e.g., $f \wedge f'$ is a predicate transformer which maps a predicate P to the predicate $f\ P \wedge f'\ P$. Similarly, the partial order can be lifted:

$$f \leq f' \overset{\text{def}}{=} \forall P .\ f\ P \leq f'\ P$$

Since predicate transformers are functions, they can be composed. The composition $f \circ f'$ is defined and has arity $u \leftarrow w$ if f has arity $u \leftarrow v$ and f' has arity $v \leftarrow w$.

Characteristic properties of predicate transformers. A predicate transformer f is *monotonic* if it preserves the partial order on predicates, i.e., if

$$P \leq Q \Rightarrow f\ P \leq f\ Q$$

holds for all predicates P and Q. Monotonicity is important, since nonmonotonic predicate transformers never denote programs in our framework.

In addition to monotonicity, we define a number of other properties that predicate transformers may have. A predicate transformer f is said to be *conjunctive* if it distributes over nonempty conjunctions, i.e., if

$$f(\bigwedge_{i\in I} P_i) = \bigwedge_{i\in I}(f\ P_i)$$

holds for arbitrary nonempty index set I. Similarly, f is *disjunctive* if it distributes over nonempty disjunctions and *continuous* if it distributes over chains of predicates. Furthermore, f is said to be *strict* if f *false* $=$ *false*. The importance of these properties will become clear in Section 4.

3.4. Predicate transformers in HOL

In HOL, predicate transformers are represented by the type `(*s')pred` $\rightarrow$ `(*s)pred`, which we abbreviate `(*s,*s')ptrans`. Here, `*s` and `*s'` are the types of the underlying initial and final state spaces (note that these state spaces need not be the same).

Characteristic properties of predicate transformers are defined in a straightforward way. For example, monotonicity is defined as follows

```
⊢def monotonic f = ∀p q. p implies q ⇒ (f p) implies (f q)
```

We will return to predicate transformers in HOL in Section 5.

A least fixpoint operator. For the semantics of iteration, we need least fixpoints of mototonic predicate transformers. We do not go into details here, we simply assume that `fix f` gives the least fixpoint of an arbitrary monotonic predicate transformer `f`. The definition of `fix` and the proofs of basic fixpoint properties are quite straightforward in HOL. In particular, the following two theorems together show that `fix` really is the least fixpoint operator:

```
⊢ monotonic f ⇒ (f(fix f) = fix f)
⊢ (f p) implies p ⇒ (fix f) implies p
```

4. A command notation

In the weakest precondition semantics, a command C denotes a predicate transformer $wp(C,\cdot)$ such that $wp(C,Q)$ holds in a state σ if and only if execution of C from initial state σ is guaranteed to terminate in a final state where Q holds [3]. For example, the weakest precondition semantics of sequential composition and assignment are given as follows:

$$wp(C;C',Q) = wp(C,wp(C',Q))$$
$$wp(x := E,Q) = Q[E/x]$$

In this section, we introduce a command notation with a weakest precondition semantics. For simplicity, we identify commands with their semantic functions, so we write $C\ Q$ rather than $wp(C, Q)$. If $C\ Q$ holds in a state σ, we say that *C establishes the postcondition Q when executed in initial state σ.*

4.1. A basic command notation

We do not define a language with a close syntax. Instead, we define a number of useful constructs, by describing what predicate transformers they denote. The notation can then be extended by introducing new commands later on.

First of all, we introduce some constructs that have traditionally been used in weakest precondition semantics. The assert command, (multiple) assignment, sequential composition, and conditional composition denote predicate transformers according to the following definitions:

$$\{P\}Q \overset{\mathrm{def}}{=} P \wedge Q$$
$$(x := E)Q \overset{\mathrm{def}}{=} Q[E/x]$$
$$(C; C')Q \overset{\mathrm{def}}{=} C(C'Q)$$
$$(\textbf{if}\ G \rightarrow C \,\Box\, G' \rightarrow C'\ \textbf{fi})Q \overset{\mathrm{def}}{=} (G \vee G') \wedge (G \Rightarrow C\ Q) \wedge (G' \Rightarrow C'\ Q)$$

The *assert* command $\{P\}$ is an assertion about the state. If P holds, then it does nothing, otherwise it does not establish any postcondition, not even *true* (in this case, it can be interpreted as being nonterminating). The other commands have their standard intuitive interpretation [3].

Each command in our notation has an *arity*, which is the same as the arity of the predicate transformer it denotes. The arity indicates an assumption about what the global state space is. For example, if we work in the global state space (b, x, y), then the arity of the assignment $x := E$ is $(b, x, y) \leftarrow (b, x, y)$. Note that the arity of a command is not directly related to the free variables occurring in a syntactic representation. The free variables of the assignment command $x := x + y$ are x and y, but the arity of this command depends on what the underlying global state space is. Generally speaking, the arity of this assignment is $u \leftarrow u$ for any u containing both x and y.

For iteration, we use the notation

$$\textbf{do}\ G_1 \rightarrow C_1 \ \Box \ldots \Box G_n \rightarrow C_n\ \textbf{od}$$

Intuitively speaking, an iteration is executed by repeatedly evaluating all the guards G_i and nondeterministically executing one of the commands C_i whose guard evaluated to T. This is repeated until no guard evaluates to T, at which point execution terminates. The semantics of iteration is most easily given by using least fix points. Details about how this is achieved can be found elsewhere, e.g., in [25].

Healthiness conditions. The commands introduced so far all satisfy the so-called "healthiness conditions" of Dijkstra [3]. They are monotonic, conjunctive, strict, and continuous. Conjunctivity can intuitively be interpreted as saying that the nondeterminism associated with the iteration is "demonic." Strictness means that a command can never establish the postcondition *false*. It can be interpreted as saying that commands are total, in the sense that they can always be executed, no matter what the initial state is. Continuity corresponds to the assumption that nondeterminism is bounded, i.e., there is never a choice between an infinite number of possible final states.

4.2. A nondeterministic assignment

We define a notation for a nondeterministic assignment, which generalises the ordinary assignment. The intention is to define a command which nondeterministically chooses among a number of possible final states. A similar command was introduced in [26], as a way of expressing arbitrary input-output specifications.

Assume that M is a relation on $\sum_u \times \sum_v$, written as a curried function. The *nondeterministic assignment* $\langle u * v' \,.\, M\rangle$ is then defined as follows:

$$\langle u * v' \,.\, M\rangle Q \; u \overset{\text{def}}{=} \forall v' \,.\, M \; u \; v' \Rightarrow Q \; v'$$

The nondeterministic assignment is not strict. If the initial state u is such that $M \; u \; v$ does not hold for any choice of v, then the above definitions say that $\langle u * v' \,.\, M\rangle$ establishes any postcondition Q, even *false*. In this case we say that the nondeterministic assignment is *miraculous*, because it seemingly can make the predicate *false* hold. These kinds of "miracles" have been shown to be useful in program development [27]. They represent nonimplementable commands, but they can be useful in intermediate steps of program development.

A nondeterministic assignment can be noncontinuous. This is the case if it permits unbounded nondeterminism (i.e., if there is an infinite number of possible final states for some initial state).

A nondeterministic assignment has arity $u \leftarrow v$. Intuitively, this means that it can remove some variables from the state space and add other, new variables. The priming of the final state v in the definition above is needed to distinguish variables that occur in both the initial and the final state.

As an example, consider the following command:

$$\langle x, y * x', y', z' \,.\, x' = x \wedge y' = y \wedge z' > y\rangle$$

Given an initial state where $x = 0$ and $y = 1$ it adds a variable z which is assigned a nondeterministically chosen value, greater than the initial value of y. The values of x and y are left unchanged.

The nondeterministic assignment generalizes the multiple assignment, which is deterministic. As an example, consider the assignment $x := x + y$ in the state space (b, x, y). This assignment is written as a nondeterministic assignment in the following way:

$$\langle b, x, y * b', x', y' \,.\, b' = b \wedge x' = x + y \wedge y' = y\rangle$$

Note that we explicitly have to state that the variables b and y remain unchanged.

Special cases of nondeterministic assignments. The *guard command* $[P]$ represents the assumption that the predicate P holds in the state. It is an abbreviation for the nondeterministic assignment

$$\langle u * u' \,.\, P\ u \wedge u' = u\rangle$$

If the predicate P holds then the command $[P]$ does nothing, otherwise it is miraculous. It is easy to check that it denotes the following predicate transformer:

$$[P]Q \overset{\text{def}}{=} P \Rightarrow Q$$

Guard commands are used as components in *actions* (also known as *guarded commands*). An action $G \rightarrow C$ is the sequential composition $[G]; C$. In this notation, G is called the *guard* and C the *body* of the action.

We also introduce special notation for commands that add or remove variables. The *initialization command* $\langle +x \,.\, P\rangle$ adds variables x to the state and assigns them values so that P is established (if this is not possible, then it is miraculous). Dually, the *finalization* command $\langle -x\rangle$ removes the variables x from the state without changing the values of the remaining variables. The initialization command has arity $u \leftarrow x, u$ while the finalization command has arity $x, u \leftarrow x$. Their formal definitions are as follows:

$$\langle +x \,.\, P\rangle Q \overset{\text{def}}{=} \forall x \,.\, P \Rightarrow Q$$
$$\langle -x\rangle Q \overset{\text{def}}{=} Q$$

It is easy to see that both of these commands are special cases of the nondeterministic assignment.

Blocks with local variables. Using initialization and finalization commands we can define *initialized blocks:* we consider $|[\,\mathbf{var}\ x.\ P; C\,]|$ to be an abbreviation for the command $\langle +x.\ P\rangle; C; \langle -x\rangle$. The initialized block then has the following meaning:

$$|[\,\mathbf{var}\ x.\ P; C\,]|Q = \forall x \,.\, P \Rightarrow C\ Q$$

This definition corresponds to previous definitions of the semantics of a block with local variables [28].

4.3. The refinement relation

Recall that commands C and C' are ordered $C \leq C'$ if $C\ Q \leq C'\ Q$ holds for all predicates Q. When this is the case, we say that C *is refined by* C'.

Refinement can also be expressed in terms of total correctness. We say that command C is *totally correct* with respect to precondition P and postcondition Q whenever $P \leq C\ Q$ holds, i.e., if C is guaranteed to establish Q whenever it is executed in an initial state where P holds. The refinement $C \leq C'$ then holds exactly when C' satisfies every total correctness specification satisfied by C. Thus, $C \leq C'$ means that C' can be used as a replacement for C in any program. Since we can write general specifications in our command notation, the refinement relation is useful in program development; if we want to show that a program C' satisfies a specification C, we show that the refinement $C \leq C'$ holds.

Refinement of initialized loops. Our main interest lies in the refinement of *initialized loops*. These are commands of the form

$$|[\,\mathbf{var}\ x.\ P; \mathbf{do}\ G_1 \to C_1 \,\square \ldots \square\, G_n \to C_n\ \mathbf{od}\,]|$$

Some of our rules consider initialized loops with only one action.

The initialized loop is a basic building block in sequential programs. It can also be given a parallel interpretation (the *action systems* of [29]). Furthermore, initialized loops can be interpreted as reactive components [30]. Thus, the initialized loop is a very useful program construct.

5. Representing commands in HOL

Recall that commands (i.e., predicate transformers) have generic type `(*s, *s')ptrans`.

We represent commands by defining new constants that correspond to the command notation introduced in Section 4. Below are the definitions for those commands that will be used later:

```
⊢def guard p q          = p imp q
⊢def assert p q         = p and q
⊢def assign e q         = q (e s)
⊢def nondass m q        = λs. ∀s'. m s s' ⇒ q s
⊢def (c seq c')q        = c(c' q)
⊢def add p q            = λs. ∀x. p(x,s) ⇒ q(x,s)
⊢def remove q           = λ(x,s). q s
⊢def if2(g,c)(g',c') q = (g or g') and (g imp (c q))
                            and (g' imp (c' q))
⊢def do1(g,c) q         = fix (λp. ((not g) and q) or (g and (c p)))
```

A few remarks are appropriate here. In the `assign` command, `e` is a state-state function. In the definitions of the `add` and `remove` commands, we explicitly say that the added (removed) variable(s) must be the first component of the state space. In the iteration, the pair `(g,c)` represents the action $G \to C$.

The assign command is used to model ordinary multiple assignments. As an example, consider the assignment $x := x + y$ in the state space (b, x, y), which is represented as

$$\texttt{assign}\ \lambda(b, x, y)\ (b, x + y, y).$$

As explained in Section 4.2, any multiple assignment can also be written as a nondeterministic assignment. For example, the assignment considered above can be written as

$$\texttt{nondass}\ \lambda(b, x, y)(b', x', y').\ (b' = b) \wedge (x' = x + y) \wedge (y' = y)$$

Here we use the following convention: the initial state is represented by a tuple of unprimed variables while the variables in the final state are primed. This makes the HOL representation easily readable. Note, however, that it is just a convention. It would be equally correct to represent the same assignment as

$$\texttt{nondass}\ \lambda(b, x, x')(z, y', y).\ (z = b) \wedge (y' = x + x') \wedge (y = x')$$

since the two representations are alpha-equivalent.

The block with local variables is defined in terms of the simpler commands, as in (1):

$$\vdash_{def} \texttt{block p c} = (\texttt{add p})\ \texttt{seq c seq remove}$$

Note that the name of the local variable does not show in the definition. This is because variables are identified by their position in the state tuple. In particular, the local variable of the block is the first component in the state tuple.

General iteration. Our representation of the iteration, do1, permits only one action (g,c). To define the general iteration we can use the fact that it can be rewritten as a simple iteration, using a conditional composition:

$$\begin{aligned}&\textbf{do}\ G_1 \rightarrow C_1 \,\square \ldots \square\, G_n \rightarrow C_n\ \textbf{od}\\ &\quad = \textbf{do}\ G_1 \vee \ldots \vee G_n \rightarrow \textbf{if}\ G_1 \rightarrow C_1 \,\square \ldots \square\, G_n \rightarrow C_n\ \textbf{fi}\ \textbf{od}\end{aligned}$$

In order to represent the general iteration in HOL, we first make a few preliminary definitions. These make use of the HOL facility for making definitions by primitive recursion over lists. We first define the guard of a list al of actions:

$$\vdash_{def} (\texttt{lguard NIL} = \texttt{false}) \wedge (\texttt{lguard}(\texttt{CONS}(g, c)\,\texttt{al}) = \texttt{g or}(\texttt{lguard al}))$$

Here al is a list of actions, so it has type ((*s)pred×(*s,*s')ptrans)list. Next, we define a generalized conditional composition as a recursive binary conditional:

$$\begin{aligned}&\vdash_{def} (\texttt{lif NIL} = \texttt{abort}) \wedge\\ &(\texttt{lif (CONS (g, c) al)} = \texttt{if2 (g, c) (lguard al, lif al)})\end{aligned}$$

We can then easily define the general iteration command ldo.

```
⊢def ldo al = dol (lguard al,lif al)
```

Reasoning about `ldo` can generally be reduced to reasoning about `dol`.

Initialized loops. It is now straightforward to represent an initialized loop using `block` and `ldo`. The loop $|[\,\mathbf{var}\ x.\ P; \mathbf{do}\ G_1 \rightarrow C_1 \square \ldots \square G_n \rightarrow C_n\ \mathbf{od}\,]|$ is represented as follows:

```
block p (ldo [(g1,c1);...;(gn,cn)])
```

where `p` represents the initialization predicate P, and each action $G_i \rightarrow C_i$ is represented by a pair `(gi,ci)`.

The refinement relation. The refinement relation is defined as an infix constant in the obvious way:

```
⊢def c ref c′ = ∀q. (c q) implies (c′ q)
```

This relation is then easily proved to be a partial order. A large number of useful refinement rules can be proved directly from the definitions. Two simple rules are given in the following theorems:

```
⊢ p implies p′ ⇒ (assert p) ref (assert p′)
⊢ skip seq c = c
```

Note that the second one of the above rules is an equality (i.e., a mutual refinement). Although refinement is the interesting relation between commands, it is useful to prove equalities whenever they hold. This is because equalities can be used directly as rewrite rules by the HOL system. In contrast, rewriting using refinements may require extensive interaction with the system.

5.1. Characteristic properties of commands

Monotonicity, conjunctivity, disjunctivity, and continuity are defined in a straightforward way:

```
⊢def monotonic c   = ∀p q. p implies q ⇒ (c p) implies (c q)
⊢def conjunctive c = ∀P. c (glb P) = glb (λq. ∃p. P p ∧ (q = c p))
⊢def disjunctive c = ∀P. c (lub P) = lub (λq.  ∃p. P p ∧ (q = c p))
⊢def continuous c  = ∀Q. chain Q ⇒ (c (limit Q) =,limit (λn. c(Q n)))
```

where `P` has type `(*s)pred→bool` and `Q` has type `num→(*s)pred` (i.e., `P` represents a set and `Q` a sequence of predicates).

In predicate transformer semantics, it is usually implicitly assumed that all commands are monotonic and conjunctive. When working with HOL, we must make such assumptions explicit. An example is the following rule:

$$C;\{P\} = \{C\ P\};C$$

which is valid when C is a conjunctive command. It is easy to prove this rule as an HOL theorem, but we have to make the conjunctivity assumption explicit:

$\vdash$ `conjunctive c` $\Rightarrow$ `(c seq (assert p) = (assert(c p)) seq c)`

This is an example of a rule for *propagating assertions*. Such rules are important in the refinement calculus, since they permit context-dependent information to be introduced at different places in a program. This information can then be used in the refinement process [1].

5.2. *Continuity and bounded nondeterminism*

In Section 8 we encounter a situation where loop bodies are assumed to be continuous. In such cases, we can use another formulation of the semantics of iteration. In our HOL theory, we have proved the theorem

$\vdash$ `continuous c` $\Rightarrow$ `(do1(g, c) q = limit (H g c))`

where the chain of approximations is defined by primitive recursion:

```
⊢def (H g c 0 q = (not g) and q) ∧
     (H g c (n+1) q = ((not g) and q) or (g and c(H g c n q)))
```

This corresponds to the original definition of the semantics of iteration, given by Dijkstra [3].

As noted in Section 4.2, a nondeterministic assignment is noncontinuous if there are an infinite number of possible final states for some initial state. To be able to identify continuous assignments, we have proved the following theorem:

$\vdash$ `(∀s. finite (λs'. m s s'))` $\Rightarrow$ `continuous (nondass m)`

where finiteness is defined using functions from natural numbers:

$\vdash_{def}$ `finite A = ∃f n. ∀a. A a ⇒ ∃i.(i < n) ∧ (f i = a)`

(i.e., a set A is finite if and only if there is an onto function from the set $\{1, 2, ..., n\}$ to A, for some n).

6. Data refinement

Data refinement means replacing local variables in a program with new local variables of another type so that the new program is a refinement of the original. Typically, the purpose of a data refinement is to replace variables that represent an abstract view of data (e.g., sets) with new variables that are more concrete, in the sense that they are more easily implemented (e.g., bit strings). Data refinement is generally proved by exhibiting an *abstraction relation* which shows how that abstract and concrete states are related.

In our framework, a data refinement is a refinement step of the following form:

$$|[\mathbf{var}\ a.\ P; C]| \le |[\mathbf{var}\ k.\ P'; C']| \tag{2}$$

In what follows, we consider a method of proving such refinements, which is easily represented in HOL. We only give a brief overview of the method; background and detailed correctness proofs can be found in [31, 32].

6.1. *Abstraction commands*

We consider the refinement (2) assuming that the list of global (universal) variables is u. We call the local variables a the *abstract variables* and k the *concrete variables*. Assume that R is a predicate on a, k, u (an *abstraction relation*). We then define two new commands corresponding to the relation R: the *angelic abstraction command* $\langle \vee + a - k.\ R\rangle$ and the *demonic abstraction command* $\langle \wedge + a - k.\ R\rangle$. The definitions are the following:

$$\langle \vee + a - k.\ R\rangle Q \overset{\text{def}}{=} \exists a.\ R \wedge Q$$
$$\langle \wedge + a - k.\ R\rangle Q \overset{\text{def}}{=} \forall a.\ R \Rightarrow Q$$

Intuitively, these commands can be described as follows. Both commands start in a concrete state (k, u). They both add the abstract variables a, choosing values so that R is established. Finally, they remove the concrete variables, so that they terminate in an abstract state (a, u). In the angelic abstraction command the choice of final state is made angelically, while it is made demonically in the demonic abstraction command.

This means that the demonic abstraction command establishes a postcondition Q if Q holds in the final state, regardless of how the choice of final state is made. In contrast, the angelic abstraction command establishes a postcondition Q if there exists at least one choice of final state that makes Q hold. For a more detailed discussion of how angelic choice can be interpreted, we refer to [33].

6.2. *A data refinement relation*

For an arbitrary command α we define the indexed refinement relation $\le_\alpha$ as follows: $C \le_\alpha C'$ is defined to hold when the following refinement holds:

$$\alpha; C \le C'; \alpha \tag{3}$$

In the special case when α is the angelic abstraction command $\langle \vee + a - k.\ R\rangle$ we call $\le_\alpha$ the *data refinement relation*. In this case, definition (3) is equivalent to the condition that

$$R \wedge CQ \Rightarrow C'(\exists a.\ R \wedge Q)$$

holds universally for all predicates Q. This means that our definition coincides with the traditional notion of data refinement between commands [4, 6, 27].

Proof rules for data refinement. Now let R be an abstraction relation and let α be the corresponding angelic abstraction command, $\alpha = \langle \vee + a - k.\ R \rangle$. The following general rule of data refinement shows how to data-refine blocks with local variables, i.e., it gives conditions under which we can replace one data representation by another one in a program:

Theorem 1. Assume that α is an angelic abstraction command. Then the refinement

$$|[\,\mathbf{var}\ a.\ P; C]| \leq |[\mathbf{var}\ k.\ P'; C'\,]|$$

is valid if the following conditions hold:

1. $P' \leq \alpha P$
2. $C \leq_\alpha C'$

Condition 1 can be rewritten, using the definition of the angelic abstraction command, as

$$P' \leq \exists a.\ R \wedge P$$

In order to prove data refinements correct, we need rules for proving refinements of the form $C \leq_\alpha C'$. We concentrate on those rules that are needed to prove data refinements of initialized loops, where the action bodies are nondeterministic assignments.

For the nondeterministic assignment, we quote a rule from [32]. Intuitively, the rule says that given initial states related by the abstraction relation R and given an execution of $C' = \langle k, u * k', u'.P' \rangle$, there must exist an execution of $C = \langle a, u * a', u'.P \rangle$ such that the final states are related by R. The rule encodes what is often called *simulation*: every possible execution of C' must simulate a possible execution of C (see figure 1).

This rule has the following formulation in our framework:

Theorem 2. The data refinement

$$\langle a, u * a', u'.P \rangle \leq_\alpha \langle k, u * k', u'.P' \rangle$$

is valid if the following condition holds:

$$\begin{aligned}&\forall a, k, u, k', u'.\, R(a, k, u) \wedge P'(k, u, k', u')\\&\quad \Rightarrow \exists a'.\ R(a', k', u') \wedge P(a, u, a', u')\end{aligned}$$

$$\begin{tikzcd}
a, u \arrow[r, dashed, "C"] \arrow[d, dash, "R"'] & a', u' \arrow[d, dash, dashed, "R"] \\
k, u \arrow[r, "C'"'] & k', u'
\end{tikzcd}$$

Fig. 1. Data refinement.

This rule can also be used for multiple assignments, since they are special cases of the nondeterministic assignment.

A rule for iterations. Data refinement between two iterations is most easily proved by showing that the concrete action simulates the abstract one, and that the two iterations have equivalent termination conditions. Thus the data refinement

$$\textbf{do } G \rightarrow C \textbf{ od} \leq_\alpha \textbf{do } G' \rightarrow C' \textbf{ od} \tag{4}$$

is valid if the following two conditions hold:

1. $C \leq_\alpha [G']; C'$
2. $\alpha(G) = G'$

For general iterations, this rule is generalized as follows:

Theorem 3. The data refinement

$$\textbf{do } G_1 \rightarrow C_1 \square \ldots \square G_n \rightarrow C_n \textbf{ od} \leq_\alpha \textbf{do } G'_1 \rightarrow C'_1 \square \ldots \square G'_n \rightarrow C'_n \textbf{ od}$$

is valid if the following conditions hold:

1. $\alpha(\neg G_i) \leq \neg G'_i$ *for all* i
2. $C_i \leq_\alpha [G'_i]; C'_i$ *for all* i
3. $\alpha(\bigvee_i G_i) \leq \bigvee_i G'_i$

Note that in this rule, condition 1 contains a separate enablement condition for each action, condition 2 is a simulation condition for each action, and condition 3 is a termination condition for the iteration as a whole.

The rules in Theorems 1–3 give us all we need to prove data refinements between initialized loops when the action bodies are written as nondeterministic assignments.

7. Representing data refinement in HOL

We now show how data refinements can be expressed and proved correct in HOL. We begin by representing the angelic abstraction command $\langle \vee + a - k.\ R\rangle$ as `abst r`:

$$\vdash_{def} \texttt{abst r q} = \lambda(\texttt{k}, \texttt{u}).\ \exists \texttt{a}.\ \texttt{r}(\texttt{a}, \texttt{k}, \texttt{u}) \wedge \texttt{q}(\texttt{a}, \texttt{u})$$

Here `r` is the abstraction relation, `a` stands for the abstract variables, `k` for the concrete variables, and `u` for the common (global) variables. Then the data refinement relation is defined in a straightforward manner:

$$\vdash_{def} \texttt{dataref r c c}' = ((\texttt{abst r})\ \texttt{seq c})\ \texttt{ref}\ (\texttt{c}'\ \texttt{seq}\ (\texttt{abst r}))$$

7.1. *Rules of data refinement*

We begin by showing the HOL theorem corresponding to the basic rule for data refinement of blocks (Theorem 1).

$$\begin{aligned}&\vdash (\forall \texttt{k u}.\ \texttt{p}'(\texttt{k,u}) \Rightarrow \exists \texttt{a}.\ \texttt{r(a,k,u)} \wedge \texttt{p(a,u)}) \wedge \texttt{dataref r c c}'\\ &\qquad \Rightarrow (\texttt{block p c})\ \texttt{ref}\ (\texttt{block p}'\ \texttt{c}')\end{aligned}$$

The proof of this theorem in HOL is quite extensive. However, it contains no surprises; it is essentially a coding of the informal paper-and-pen proof of the corresponding theorem.

The rule for data-refining nondeterministic assignments (Theorem 2) is represented in the following theorem:

$$\begin{aligned}&\vdash (\forall \texttt{a c u c}'\ \texttt{u}'.\ \texttt{r(a,c,u)} \wedge \texttt{Q(c,u)(c}',\texttt{u}')\\ &\qquad \Rightarrow \exists \texttt{a}'.\ \texttt{r(a}',\texttt{c}',\texttt{u}') \wedge \texttt{P(a,u)(a}',\texttt{u}'))\\ &\qquad \Rightarrow \texttt{dataref r (nondass P) (nondass Q)}\end{aligned}$$

The proof of this rule in HOL is straightforward. In fact, the implication can be strengthened to equality.

Finally, we have the rule for data-refining iterations (Theorem 3). As an HOL theorem, the rule is expressed as follows:

```
⊢ (LENGTH al = LENGTH al')
  ∧ EVERY (λ(g,c). monotonic c) al'
  ∧ EVERY (λ((g,c),(g',c')). dataref r c ((guard g') seq c'))
          (ZIP al al')
  ∧ EVERY (λ((g,c),(g',c')). (∀a k u. r(a,k,u) ∧ g'(k,u)
        ⇒ g(a,u))) (ZIP al al')
  ∧ (∀a k u. r(a,k,u) ∧ (lguard al)(a,u) ⇒ (lguard al')(k,u))
  ⇒(ldo al) ref (ldo al')
```

where EVERY P xl means that the property P holds for all elements of the list xl and ZIP zips together two lists (both EVERY and ZIP are built-in HOL constants).

The proof in HOL of this rule is rather tricky. It involves an inductive argument which requires rewriting the general conditional statement as

$$\mathbf{if}\ G_1 \rightarrow C_1 \,\square \ldots \square\, G_n \rightarrow C_n\ \mathbf{fi} = \{\bigvee_{i=1}^{n} G_i\}; ([G_1]; C_1 \wedge \ldots \wedge [G_n]; C_n)$$

where $\wedge$ is a separately defined conjunction operator on commands.

As mentioned before, these three rules together give us all we need to prove a data refinement between two initialized loops when the bodies are written as nondeterministic assignments. In practice, the bodies may contain other constructs, e.g., assignments, sequential compositions, and conditionals. However, such commands can always be mechanically rewritten as nondeterministic assignments.

7.2. *Applying the rule to a case*

In order to show how our formalization can be used in practice, we describe the outline of a refinement example. We consider two programs searching for the maximal value of some function f within the interval $0 \ldots n-1$. The first program performs the search by computing the function values as they are needed, whereas the second program computes all the necessary function values in the beginning and stores them in a list. The second program requires more memory, but it may be more efficient in some cases.

We assume that f is the function and m the variable where the maximal value is stored. Our goal is then to prove that the 'abstract' program

```
|[ var $a, i. (a = f\ 0 \wedge i = 0 \wedge m = 0)$;
      do $(i < n \wedge m < a) \rightarrow a, i, m := f(i+1), i+1, a$
      □ $( i < n \wedge a \leq m) \rightarrow a, i := f(i+1), i+1$
      od
]|
```

is refined by the "concrete" program

```
|[ var $k$. $(k = [f\ 0; ..; f(n-1)] \wedge m = 0)$;
    do $(k \neq NIL \wedge m < HD\,k) \rightarrow m, k := HD\,k, TL\,k$
    ▯ $(k \neq NIL \wedge HD\,k \leq m) \rightarrow k := TL\,k$
    od ]|
```

The occurrence of $m = 0$ in the initialization indicates that both programs assume that m has been initialized to 0 before the blocks are entered.

The abstraction relation used in the proof of data refinement is

$$R: \ (i < n \Rightarrow a = HD\ k) \wedge n = LENGTH\ k \\ \wedge (\forall j. i + j < n \Rightarrow k_j = f(i+j))$$

where we use the notation k_j for the jth element of the list k.

In the HOL formalization of this problem, both f and n are global constants, i.e., they do not appear in the state tuple. The global state component is m:num. The 'abstract' program has the local component (a:num,i:num) while the local component of the 'concrete' program is k:(num)list.

Before we represent the programs in HOL, we define a function seqlist which returns the list [f 0;..;f(n-1)], given f and n as arguments:

```
⊢def (seqlist f 0  =  NIL)
   ∧ (seqlist f (n + 1) = APPEND (seqlist f n)[f n])
```

Now the two programs are written in the HOL command notation and the goal is set up:

```
"(block (λ((a,i),m). (a = f 0) ∧ (i = 0) ∧ (m = 0))
    (ldo [(λ((a,i),m). i < n ∧ m < a), assign λ((a,i),m).
                                               ((f(i+1),i+1),a);
          (λ((a,i),m). i < n ∧ a ≤ m), assign λ((a,i),m).
                                               ((f(i+1),i+1),m)]))
 ref
 (block (λ(k,m). (k = seqlist f n) ∧ (m = 0))
    (ldo [(λ(k,m). ¬(k = NIL) ∧ m < (HD k)), assign λ(k,m).
                                                     (TL k,HD k);
          (λ(k,m). ¬(k = NIL) ∧ (HD k) ≤ m), assign λ(k,m).
                                                     (TL k,m)]))"
```

In order to prove the goal, we first match it against the rule for data refinement of blocks. As a part of this matching, we supply the abstraction relation, which in HOL has the form

```
"λ((a,i),k,m). ((i < n) ⇒ (a = HD k)) ∧ (n = (LENGTH k) + i) ∧
               (∀j. (i + j < n) ⇒ (EL j k = f (i + j)))"
```

where EL j k is the jth element of the list k. This yields two subgoals:

```
    "∀k m. (k = seqlist f n) ∧ (m = 0) ⇒
           (∃a. R((a,i),k,u) ∧ (a = f 0) ∧ (i = 0) ∧ (m = 0))"

    "dataref R
      (ldo [(λ((a,i),m). i < n ∧ m < a), assign λ((a,i),m).
                                                      ((f(i+1),i+1),a);
            (λ((a,i),m). i < n ∧ a ≤ m), assign λ((a,i),m).
                                                      ((f(i+1),i+1),m)])
      (ldo [(λ(k,m). ¬(k = NIL) ∧ m < (HD k)), assign λ(k,m).
                                                          (TL k, HD k);
            (λ(k,m). ¬(k = NIL) ∧ (HD k) ≤ m), assign λ(k,m).
                                                            (TL k,m)])"
```

where R is an abbreviation for the abstraction relation. The first subgoal is proved using built-in tools for arithmetic reasoning and some theorems characterizing the `seqlist` function.

Matching the second subgoal against the rule for data-refining iterations, we obtain eight subgoals (after some rewriting, which among other things proves automatically that the two action lists have equal length):

```
"∀ a i k u. R((a,i),k,u) ∧ (i < n ∧ u < a ∨ i < n ∧ a ≤ u) ⇒
  ¬(k = NIL) ∧ u < (HD k) ∨ ¬(k = NIL) ∧ (HD k) ≤ u"

"∀a i k u. R((a,i),k,u) ∧ ¬(k = NIL) ∧ (HD k) ≤ u ⇒ i < n ∧ a ≤ u"

"∀a i k u. R((a,i),k,u) ∧ ¬(k = NIL) ∧ u < (HD k) ⇒ i < n ∧ u < a"

"dataref R
  (assign λ((a,i),m). ((f(i+1),i+1),m))
  ((guard (λ(k,m). ¬(k = NIL) ∧ (HD k) ≤ m)) seq (assign λ(k,m).
                                                        (TL k,m)))"

"dataref R
  (assign(λ((a,i),m). ((f(i+1),i+1),a)))
  ((guard(λ(k,m). ¬(k = NIL) ∧ m < (HD k))) seq (assign λ(k,m).
                                                     (TL k,HD k)))"

"monotonic(assign λ(k,m). (TL k,m))"

"monotonic(assign λ(k,m). (TL k,HD k))"
```

The two monotonicity subgoals are proved automatically by a simple standard tactic that we have written in ML.

The first three subgoals (which correspond to the termination condition and the two enablement conditions) require reasoning over the data domains of the program variables, in this case natural numbers and lists. The proofs are not very difficult using the arithmetic and list libraries of the HOL system.

The two remaining subgoals can be matched against the rule for data-refining nondeterministic assignments. However, we must first rewrite the commands as nondeterministic assignments. We show what happens in the first case (the second one is similar). A simple automatic rewrite procedure that we have implemented produces the following goal:

```
"dataref R
 (nondass (λ((a,i),m) ((a',i'),m'). (a',i'),m' = (f(i+1),i+1),m))
 (nondass (λ(k,m) (k',m'). ¬(k = NIL) ∧ (HD k) ≤ m
                                              ∧ (k',m' = TL k,m)))"
```

Now we can match this goal against the rule for data refinement of nondeterministic assignments. After some automatic rewriting, we obtain the following goal:

```
"((i < n) ⇒ (a = HD k)) ∧ (n = (LENGTH k) + i) ∧
 (∀j. (i + j < n) ⇒ (EL j(TL k) = f((i+1) + j))) ∧
 ¬(k = NIL) ∧ (HD k ≤ m)
 ⇒ ((i+1) < n) ⇒ (f(i+1) = HD(TL k))) ∧
     (n = LENGTH(TL k) + (i+1)) ∧
     (∀j. ((i+1) + j < n) ⇒ (EL j(TL k) = f((i+1) + j)))"
```

This goal is easily proved using basic facts from the arithmetic and list libraries.

Comments on the data refinement. Our example shows that HOL can handle a small data refinement well. Simple subgoals, such as tautologies and monotonicity conditions, can be proved automatically (that is, we have written proof strategies that do these proofs). The main innovative step in a data refinement proof is finding the abstraction relation R. Here, HOL does not help; users must supply the abstraction relation themselves. Our example also shows that the abstraction relation is at the same time an invariant. The first conjunct of R is the "real" abstraction relation, while the two other conjuncts are an invariant of the concrete program, needed to make the proof go through.

Once the abstraction relation is given, the steps needed to obtain the low-level subgoals (verification conditions) are straightforward. In our case, these subgoals were not too difficult, given some basic properties of the `seqlist` function. In fact, proving these properties was the hardest part of the proof.

8. Backward data refinement

In most cases, a simulation between two initialized loops can be proved using the ordinary rule of data refinement. However, it is well known that the rule is not complete [26]. In particular, it is not sufficient in some situations where a nondeterministic choice is made later in the concrete loop than in the abstract one. In such cases, a dual notion of *backward data refinement* can be used. This method is investigated in [32], where proofs of the basic results of this section can be found. The backward data refinement method is closely related to the methods of *upward simulation* and *backward simulation* described in [34, 35].

Backward data refinement can be defined using the indexed refinement relation (3). Assume that α is the *demonic* abstraction command $\langle \wedge + a - k.\ R \rangle$. We then say that C is *backward data-refined* by C' if $C \leq_\alpha C'$ holds.

8.1. *Rules for backward data refinement*

The rules needed for proving refinement of initialized loops by backward data refinements are similar to the rules for ordinary data refinement, given in Section 6.

We first state the rule for data-refining a block. This rule shows that backward data refinement is a sound method for data-refining blocks with local variables.

Theorem 4. Assume that α is the demonic abstraction command $\langle \wedge + a - k.\ R \rangle$. Then the refinement

$$|[\,\mathbf{var}\ a.\ P; C\,]| \leq |[\,\mathbf{var}\ k.\ P'; C'\,]|$$

is valid if the following conditions hold:

1. $P' \leq \alpha P$
2. $C \leq_\alpha C'$
3. $\forall k, u.\ \exists a.\ R(a, k, u)$

In this case, condition 1 can be rewritten as

$$R \wedge P' \leq P$$

Condition 3 says that the abstraction relation R must be total in concrete states, i.e., every concrete state must correspond to at least one abstract state.

Backward data refinement of nondeterministic assignments is proved using the following rule:

Theorem 5. The backward data refinement

$$\langle a, u * a', u'.P \rangle \leq_\alpha \langle k, u * k', u'.P' \rangle$$

is valid if the following condition holds:

$$\begin{aligned} &\forall k, u, a', k', u'.\ R(a', k', u') \wedge P'(k, u, k', u') \\ &\qquad\qquad \Rightarrow (\exists a.\ R(a, k, u) \wedge P(a, u, a', u')) \end{aligned}$$

Compare this to the rule for ordinary data refinement (Theorem 2), where we find a final state, given initial states related by R. Here we must find an initial

$$\begin{tikzcd}
a, u \arrow[r, dashed, "C"] \arrow[d, dash, dashed, "R"'] & a', u' \arrow[d, dash, "R"] \\
k, u \arrow[r, "C'"'] & k', u'
\end{tikzcd}$$

Fig. 2. Backward data refinement.

state, given final states related by R (see figure 2). This justifies calling the method *backward* data refinement.

8.2. Backward data refinement of iterations

We consider only the simple case when the iteration has a single action. In this case, we have a rule which closely resembles the corresponding rule for ordinary data refinement, given in theorem 3.

Theorem 6. The backward data refinement

$$\textbf{do } G \rightarrow C \textbf{ od} \leq_\alpha \textbf{do } G' \rightarrow C' \textbf{ od}$$

is valid if the following conditions hold:

1. $\alpha(\neg G) \leq \neg G'$
2. $C \leq_\alpha [G']; C'$
3. $\alpha(G) \leq G'$
4. C and C' are continuous
5. $\forall k, u.\ \exists a.\ R(a, k, u)$
6. α is continuous.

Condition 6 can be reformulated as follows: the abstraction relation R must be *image-finite*, i.e., every concrete state must correspond to at most a finite number of abstract states.

8.3. Backward data refinement in HOL

The demonic abstraction command used in backward data refinement is represented in our formalization by the command `dabst r`:

$$\vdash_{def} \texttt{dabst r q} = \lambda(\mathrm{k}, \mathrm{u}).\ \forall \mathrm{a}.\ \mathrm{r}(\mathrm{a}, \mathrm{k}, \mathrm{u}) \Rightarrow \mathrm{q}(\mathrm{a}, \mathrm{u})$$

We then define the backward data refinement relation `bwdref` in the following way:

$$\vdash_{def} \texttt{bwdref r c c}' = ((\texttt{dabst r})\ \texttt{seq c})\ \texttt{ref}\ (\texttt{c}'\ \texttt{seq}\ (\texttt{dabst r}))$$

In our HOL theory of backward data refinement we have proved a number of theorems that show how individual commands can be refined. For instance, the rule for blocks (theorem 4) is represented by the following theorem:

```
⊢ (∀k u. ∃a. r(a,k,u))
  ∧ (∀a k u. r(a,k,u) ∧ p'(k,u) ⇒ p(a,u))
  ∧ bwdref r c c'
  ⇒ (block p c) ref (block p' c')
```

The rule for backward data refinement of a nondeterministic assignment (theorem 5) is as follows:

```
⊢ (∀k u a' k' u'. r(a',k',u') ∧ Q(k,u)(k',u') ⇒ ∃a. r(a,k,u)
                                                     ∧ P(a,u)(a',u'))
  ⇒ bwdref r (nondass P) (nondass Q)
```

Finally, we have the rule for backward data refinement of a one-action iteration (theorem 6):

```
⊢ (∀a k u. r(a,k,u) ⇒ (g(a,u) = g'(k,u)))
  ∧ bwdref r c ((guard g') seq c')
  ∧ continuous c ∧ continuous c'
  ∧ (∀k u. ∃a. r(a,k,u))
  ∧ continuous (dabst r)
  ⇒ bwdref r (do1(g,c)) (do1(g',c,))
```

The proof in HOL of this rule is quite difficult. However, it was worth the effort, since it constitutes a check that the proof reported in [32] is correct.

8.4. Applying the rule to a case

We have applied the rule to a small example. This is a standard example of a situation where the ordinary rule of data refinement is not sufficient. Both programs decrement the value of the variable x arbitrarily. In the first program, the final value of x is chosen in the initialization, while this decision is postponed in the second program.

The refinement we want to prove is as follows:

$$\begin{array}{l} |[\,\mathbf{var}\ a.\ a \leq x; \mathbf{do}\ a < x \qquad \rightarrow x := x - 1\ \mathbf{od}\,]| \\ \leq \\ |[\,\mathbf{var}\ k.\ 0 < x; \mathbf{do}\ k \wedge (0 < x) \rightarrow \langle k,\ x * k',\ x'.x' = x - 1\rangle\ \mathbf{od}\,]| \end{array}$$

In the first program, the variable a ranges over the natural numbers. It is initially given a nondeterministically chosen value, which eventually becomes the final value of x. In the second program, the boolean variable k is assigned T or F (nondeterministically) at each loop, while x is decremented. The iteration terminates when k is assigned the value F. The abstraction relation used in the proof of this refinement is

$$R:\ (a \leq x) \wedge (k \wedge (0 < x)) = (a < x)$$

The application of the proof rules for backward data refinement is straightforward. First both programs are coded in HOL. Then the goal is set up, the abstraction relation is supplied and the goal is matched against the proof rules after rewriting the action bodies as nondeterministic assignments. In this case we get seven subgoals (verification conditions). The continuity conditions on the action bodies can be proved automatically. The five remaining subgoals are as follows:

```
"∃a. (k ∧ (0 < x) = (a < x)) ∧ (a ≤ x)"

"continuous (dabst(λ(a,k,x). (k ∧ (0 < x) = (a < x)) ∧ (a ≤ x)))"

"(k ∧ (0 < x) = (a < x)) ∧ (a ≤ x) ∧ (0 < x) ⇒ (a ≤ x)"

"(k ∧ (0 < x) = (a < x)) ∧ (a ≤ x) ⇒ ((a < x) = k ∧ (0 < x))"

"(k' ∧ 0 < u' = a'< u') ∧ a'≤ u' ∧ (k ∧ 0 < u) ∧ (u' = u - 1) ⇒
 (∃a. (k ∧ 0 < u = a < u) ∧ a ≤ u ∧ a' = a ∧ u' = u - 1)"
```

Of these, all but the second require only simple arithmetic reasoning. Since the demonic abstraction command is a special case of the nondeterministic assignment, we can prove continuity in the same way as for nondeterministic assignments. In order to facilitate such continuity proofs, we have proved the following lemma:

$\vdash (\forall \texttt{k u. finite} (\lambda \texttt{a. r(a, k, u)})) \Rightarrow \texttt{continuous (dabst r)}$

Comments on the backward data refinement. Backward data refinement is more theoretically interesting than practically useful. Its theoretical interest comes from the fact that there are cases when a data refinement cannot be proved by the ordinary method and that in certain situations, the ordinary and backward methods together provide a complete method [32, 34].

We chose the example above because it is one in which the ordinary method would not have worked. Exactly as for ordinary data refinement, the HOL system produces the verification conditions once the abstraction relation is given. Since the example was so simple, the verification conditions were easy to prove.

9. Superposition refinement

Data refinement replaces old local variables with new ones. It can also add new local variables to the state. One important use of this is explored in [29], where initialized loops (called *action systems*) are interpreted as parallel programs. In this application, the new local variables are used for distributing global information.

Sometimes we would like to add global variables to the state, in order to add functionality to a program. In this case, the rules of data refinement cannot be used, since they only permit addition of a local state component. In this section we consider a modified refinement relation, which permits the addition of new global variables.

9.1. A superposition refinement relation

Superposition is a notion which has been used in many different contexts, see, e.g., [8, 10]. Here we use it to mean the addition of components to the global state space and of computations which affect only the added state components. Given commands C of arity $u \leftarrow u$ and C' of arity $x, u \leftarrow x, u$, we say that C is *superposition refined* by C', written $C \sqsubseteq C'$, if the following refinement holds:

$$C \leq |[\,\textbf{var}\ x.\ \textit{true}; C'\,]| \tag{5}$$

Thus the variables x may give C' some additional functionality, but without affecting the input-output behavior on the variables u. The superposition refinement relation is easily shown to be reflexive and transitive, if we permit the use of an empty state component and assume that the following equality holds:

$$|[\,\textbf{var}\ \epsilon.\ \textit{true}; C\,]| = C$$

where ϵ is an empty list of variables.

A proof rule for superposition. We give, without proof, a rule for superposition of initialized loops with one action in the iteration. Intuitively, the rule is obviously correct, and the proof is not difficult.

Theorem 7. The superposition refinement

$$|[\,\mathbf{var}\ z.\ P; \mathbf{do}\ G \to C\ \mathbf{od}\,]| \sqsubseteq |[\,\mathbf{var}\ z.\ P'; \mathbf{do}\ G \to C'\ \mathbf{od}\,]|$$

is valid if the following conditions hold:

1. $P' \leq P$
2. $C \sqsubseteq C'$

The rule for superposition refinement of nondeterministic assignments is equally simple.

Theorem 8. The superposition refinement

$$\langle u * u'.\ P\rangle \sqsubseteq \langle x,\ u * x',\ u'.\ P'\rangle$$

is valid if $P' \leq P$ holds.

9.2. Representing superposition in HOL

We can translate our definition of the superposition refinement relation directly into HOL:

$\vdash_{def}$ `c sref c' = c ref (block true c')`

Here `c'` has one state component (the first) more than `c`. Thus, if the state space of `c` is `*s`, then `c'` works on the extended state space `*×*s`.

It should be noted that our definition does not capture the intended idea of (5) completely. For example, the relation `sref` cannot be proved to be reflexive or transitive. This is because HOL does not permit empty tuples and because the two levels of local variables in `block p (block p' c)` cannot be treated as a single component.

In proving superposition theorems about our command constructors, we need to be able to relate predicates and relations on the two state levels involved. We make the following definitions:

```
⊢def same_pred p p'   = ∀x u. p u = p'(x,u)
⊢def weaker_pred p p' = ∀x u. p'(x,u) ⇒ p u
⊢def weaker_rel P P'  = ∀x u y v. P'(x,u)(y,v) ⇒P u v
```

Thus, e.g., `weaker_pred p p'` means that the predicate `p` on the state space `*s` is weaker than the predicate `p'` on the state space `*×*s`. Dually to `weaker_pred` and `weaker_rel`, we can also define `stronger_pred` and `stronger_rel`.

We can now prove some theorems about superposition of different program constructs. The main rule is for superposition of initialized loops (theorem 7):

```
⊢ monotonic c ∧ monotonic c′ ∧ weaker_pred p p′ ∧ same_pred g g′
                                                      ∧ c sref c′
  ⇒ (block p (do1(g,c))) sref (block p′ (do1(g′,c′)))
```

We also need a rule for nondeterministic assignments (theorem 8):

```
⊢ weaker_rel P P′ ⇒ (nondass P) sref (nondass P′)
```

In practice some additional rules are helpful. In particular, we need the following ‘pseudo-transitivity’ rules:

```
⊢ c1 sref c2 ∧ c2 ref c3 ⇒ c1 sref c3
⊢ c1 ref c2 ∧ c2 sref c3 ⇒ c1 sref c3
```

9.3. *Applying the rule for initialized loops*

As a simple example, we consider a superposition which counts the number of times that a loop is executed. The original program is an arbitrary loop:

$$|[\,\mathbf{var}\ x.\ P; \mathbf{do}\ G \rightarrow C\ \mathbf{od}\,]|$$

and the new program has an added counter t:

$$|[\,\mathbf{var}\ x.\ P \wedge t = 0; \mathbf{do}\ G \rightarrow C; t := t + 1\ \mathbf{od}\,]|$$

In the HOL formalization of this example, we assume that the body C is written as a nondeterministic assignment. The old program is simply

```
block p (do1(g, nondass R))
```

where `p`, `g`, and `R` are unspecified. However, the new program is more complicated, since all predicates must be lifted to the extended state space:

```
block (λ(t,u). p u ∧ (t=0))
        (do1((λ(t,u). g u,(nondass λ(t,u)(t′,u′). R u u′)
                                      seq (assign λ(t,u). (t+1,u)))
```

As a first step in solving it, we rewrite the body of the new program as a nondeterministic assignment. The goal we want to prove is then the following:

```
"(block p (do1(g,nondass R)))
 sref
 (block (λ(t,u). p u ∧ (t = 0))
   (do1((λ(t,u). g u),nondass λ(t,u)(t′,u′). R u u′
                                                ∧ (t′ = t+1)))
```

We now match this goal against the proof rule for loops to obtain five subgoals (verification conditions). Two are monotonicity conditions which are proved by a simple specialized monotonicity-proving tactic. The three remaining subgoals are the following

```
"weaker_pred p (λ(t,u).  p u ∧ (t=0))"

"same_pred g (λ(t,u).  g u)"

"(nondass R) sref (nondass λ(t,u) (t′,u′). R u u′ ∧ (t′ = t+1))"
```

All these subgoals are straightforward to prove in HOL. In fact, it is not very difficult to compose a strategy, which proves them all (and other similar goals) automatically.

10. Conclusion

We have shown how the refinement calculus can be formalized as a theory in HOL. In particular, we have shown how to formalize three advanced and theoretically interesting refinement methods. For each of these methods there is a number of proof rules, which we have proved correct within the refinement calculus theory in HOL. Our proofs can be seen as a check that the rules are in fact valid. However, the main aim of these rules is to let the HOL system generate verification conditions for a refinement step. By proving these verification conditions using the system, we obtain an HOL theorem which states that the refinement is valid.

Our approach can be seen as a basis for program development in a totally secure environment. We have embedded the semantics of our programming notation in the logic by defining a new HOL constant for each construct in the programming notation. Thus our refinement theory is a conservative extension of the logic, so it is certain to be consistent [14]. This means that when we have proved a theorem stating that a refinement holds, then this refinement is guaranteed to be valid.

The examples given in this paper are quite small. The examples in Sections 8 (backward data refinement) and 9 (superposition refinement) are merely intended to show the spirit of the approach. The example in Section 7 (ordinary data refinement) is less trivial. It also indicates one problem of our approach, i.e., that the HOL system is not always well suited for reasoning over the data structures that occur in programs.

To our knowledge, the notion of superposition that we define in Section 9 is new. It is a simple notion, intended to permit the addition of new global components to the state of a program. Our example shows that this notion of superposition can easily be handled in our HOL formalization of the refinement calculus.

Refinement of larger programs is done most easily by refining subcomponents of the program separately [1]. For this, one needs a tool that makes it possible to focus on a subcomponent of a program, do a refinement and replace the original component by its refinement. Grundy's window inference tool [20] should be well suited for this. In a preliminary experiment, we have performed the data refinement of the example in Section 7 by data-refining each program component separately, using the window inference tool. The experience was very encouraging.

Before one can use the HOL system as a tool for refinement of larger programs, it is also necessary to provide a better user interface. The syntax used in interaction with HOL is primitive, but it is possible to build an extra layer for parsing and pretty-printing on top of the system. This makes it possible to use a syntax which is closer to ordinary programming notation. For an example of how this can be done, we refer to [15].

Doing proofs in HOL is tedious. In particular, reasoning in basic data domains (e.g., arithmetic and list theory) can be frustrating. Since most verification conditions from refinement rules require such reasoning, we would need automated (or semiautomated) strategies for proving different kinds of verification conditions produced by the proof rules. We already have such strategies for some of the simplest verification conditions, i.e., monotonicity and continuity conditions. However, the more difficult verification conditions often require elaborate reasoning over lists of natural numbers or other data–type–dependent reasoning for which no efficient procedures may exist.

Acknowledgments

We wish to thank the anonymous referees and Jim Grundy for their helpful comments, which led to a number of changes in the presentation.

References

1. R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.
2. C.C. Morgan. *Programming from Specifications.* Prentice-Hall, 1990.
3. E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.
4. R.J.R. Back. Changing data representation in the refinement calculus. In *21st Hawaii International Conference on System Sciences*, January, 1989.
5. P.H. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87(1):143–162, 1991.
6. J.M. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.
7. C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, 1986.
8. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
9. L. Bougé and N. Francez. A compositional approach to superposition. In *Proc. 14th Conference on Principles of Programming Languages,* San Diego, USA, January 1988, 240–249.

10. R.J.R. Back and K. Sere. Superposition refinement of parallel algorithms. In *Formal Description Techniques IV*, K.R. Parker and G.A. Rose, (eds.), Elsevier Science Publishers (North-Holland) 1992, 475–493.
11. D.M. Goldschlag. Mechanizing UNITY. In *TC2 Working Conference on Programming Concepts and Methods,* M. Broy, (ed.), Israel, 1990, IFIP, 374–401.
12. U. Engberg, P. Groenning and L. Lamport. Mechanical verification of concurrent systems with TLA. In *Proc. 4th Workshop on Computer-Aided Verification*, Montreal, Canada, June, 1992, Springer- Verlag.
13. M. Aagard and M. Leeser. Verifying a logic synthesis tool in Nuprl. In *Proc. 4th Workshop on Computer-Aided Verification,* Montreal, Canada, June, 1992, Springer-Verlag.
14. M.J.C. Gordon. 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.
15. M.J.C. Gordon. 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.
16. A.J. Camilleri. Mechanizing CSP trace theory in higher order logic. *IEEE Transactions of Software Engineering*, 16(9):993–1004, 1990.
17. F. Andersen. A Theorem Prover for UNITY in Higher Order Logic. PhD thesis, Technical University of Denmark, Lyngby, Denmark, 1992.
18. S. Agerholm. Mechanizing program verification in HOL. DAIMI IR–111, Aarhus University, 1992.
19. R.J.R. Back and J. von Wright. Refinement concepts formalised in higher-order logic. *Formal Aspects of Computing*, 2:247–272, 1990.
20. J. Grundy. A window inference tool for refinement. In *Proc. 5th Refinement Workshop*, Jones et al. (ed.), London, Jan., 1992, Springer-Verlag.
21. T. Vickers. An overview of a refinement editor. In *5th Australian Software Engineering Conference*, Sydney, May 1990.
22. L. Groves and R. Nickson. A tactic driven refinement tool. In *Proc. 5th Refinement Workshop*, Jones et al. (eds.), London, Jan., 1992, Springer-Verlag.
23. J. von Wright and K. Sere. Program transformations and refinements in HOL. In *Proc. 1991 International Workshop on Higher Order Logic Theorem Proving and its Applications*, Davis, USA, August, 1991. IEEE/ACM.
24. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
25. E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
26. R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of Mathematical Center Tracts. Mathematical Centre, Amsterdam, 1980.
27. C.C. Morgan. Data refinement by miracles. *Information Processing Letters*, 26:243–246, January, 1988.
28. J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.
29. R.J.R. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17–30, 1991.
30. R.J.R. Back. Refinement Calculus, part II: Parallel and reactive programs. In *REX Workshop for Refinement of Distributed Systems*, volume 430 of Lecture Notes in Computer Science, Nijmegen, The Netherlands, 1989, Springer-Verlag.
31. R.J.R. Back and J. von Wright. Refinement calculus, part 1: Sequential programs. In *REX Workshop for Refinement of Distributed Systems*, volume 430 of Lecture Notes in Computer Science, Nijmegen, the Netherlands, 1989, Springer-Verlag.
32. J. von Wright. The lattice of data refinement. Reports on computer science and mathematics 130, Åbo Akademi, 1992. To appear in *Acta Informatica*

33. R.J.R. Back and J. von Wright. Combining angels, demons and miracles in program specifications. *Theoretical Computer Science*, 100:365–383, 1992.
34. J. He, C.A.R. Hoare and J.W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25:71–76, 1987.
35. B. Jonsson. On decomposing and refining specifications of distributed systems. In *REX Workshop for Refinement of Distributed Systems*, volume 430 of Lecture Notes in Computer Science, Nijmegen, The Netherlands, 1989, Springer-Verlag.