A framework for parallel program refinement

J.-P. Bodeveix, M. Filali
IRIT
Université Paul Sabatier
118 Route de Narbonne
F-31062 Toulouse cédex France
email: bodeveix@irit.fr filali@irit.fr

April 26, 1995

Abstract

This paper presents a formal framework for developing provably correct parallel programs through refinements. We formalize rigorously, with respect to typing, some imperative programming constructs and propose refinements of them. Moreover, we try to make the semantics denotation as close as possible to the usual programming notation; the aim being to make easy the reasoning at the semantics level. This formalization is embedded within the HOL interactive theorem prover and consequently reusable as a framework for general programming by refinement.

1 Introduction

This paper presents a formal framework for developing provably correct parallel programs through refinements. We formalize rigorously, with respect to typing, some imperative programming constructs and propose refinements of them. After the study of some existing state representations in programming logics, we propose a new state representation allowing strong typing. Then, we adapt the semantics of usual sequential and parallel statements to our representation and show that classical or intuitive properties are still valid. In a third step, after introducing refinements in the context of transition systems, we propose and validate some refinement properties concerning sequential and parallel statements. In the last part of the paper, we show how these results have been embedded within the HOL interactive theorem prover and can be used to establish refinement properties on concrete problems like cache coherency protocols on multiprocessors.

2 Program state logic

2.1 Overview of some state representations

In this section, we first review some existing approaches for representing program states. Then, we present a "genuine" state representation, its motivation being to allow the use of well types notations of Pascal like programming languages. In some way, we try to reduce the gap between the syntactic domain where we are used to reason and the semantic domain where we can rigorously prove program properties.

2.1.1 States as mono valued functions

In this approach, the space of program variables `Var` and a space of values `Val` are assumed. A program state is a mapping from variables to values. Its type is

$$\texttt{Var} \rightarrow \texttt{Val}$$

For instance, in their programing logics theory[Tre92, vW90], `Var` is the type `string` and `Val` is the set of natural numbers. Then in order to reuse such logics, one has to encode all its variables as naturals.

2.1.2 States as type union valued functions

This representation has been used by [APP93] to describe UNITY[CM88] logics. A state is represented by a function from an enumerated of variable representatives to the union of variable values types. Then a variable is defined as a function from states to its type domain.

Suppose we have the following declaration in a PASCAL like language:

```
VAR i, j: INTEGER; b: BOOLEAN;
```

Then, we define the abstract data type `Rep` of variable representatives by the enumeration:

```
type Rep = I | J | B
```

Then, we define the union of variable types `Types` by the enumeration:

```
type Types = INT integer | BOOL boolean
```

Destructors are associated to this type in order to down-cast union-typed expressions to their effective type. Here, two functions are defined:

```
dest_int:  Type → integer
dest_int (INT n) = n

dest_bool:  Type → boolean
dest_bool (BOOL b) = b
```

A state σ is represented by a function from `Rep` to `Types`.

Now, a variable is defined as a function from states to its type. For instance, the the variable `i` is defined as follows:

$$i : (\texttt{Rep} \ \rightarrow \texttt{Types}) \rightarrow \text{ integer}$$
$$i(\sigma) = \texttt{dest\_int}\ \ (\sigma(I))$$

This representation raises a problem concerning the definition of assignment. An expression must be converted to the union type before being assigned. Thus, the type checker cannot detect that an expression is assigned to a variable of a different type. Type checking will indeed occur at proof time: an access to a badly assigned variable cannot be reduced to its value as destructor functions are partial.

2.2 Another state representation

A state is represented by a boolean valued function over *variable- value* associations. The space of well typed variable−value associations is introduced as an abstract data type of which constructors are the variables of the program. Consequently, a variable can also be considered as a function from well−typed values to the previously defined abstract data type. In a given state, the variable values are such that the association variable−value is mapped to true through the state function. Then, such a state function can also be considered as a set of associations variable−value.

In order to give a concrete view of such a representation we consider two examples concerning scalar and array variables.

2.2.1 Scalar variables

Suppose we have the following declaration in a PASCAL like language:

```
VAR i, j: INTEGER; b: BOOLEAN;
```

Then, we define the abstract data type `declarations` of well typed variable−value associations by its constructors:

```
type declarations =
          i INTEGER
        | j INTEGER
        | b BOOLEAN
```

This type declaration introduces three functions (the constructors of the data type):

$\texttt{i} : \texttt{INTEGER} \rightarrow \texttt{declarations}$
$\texttt{j} : \texttt{INTEGER} \rightarrow \texttt{declarations}$
$\texttt{b} : \texttt{BOOLEAN} \rightarrow \texttt{declarations}$

The state where the variables `i, j` and `b` have respectively the values i_0, j_0 and b_0 is represented by the function $\texttt{state}_0 : \texttt{declarations} \rightarrow \texttt{bool}$ such that[1]:

$\texttt{state}_0(i(x)) = (x = i_0)$

$\texttt{state}_0(j(x)) = (x = j_0)$

$\texttt{state}_0(b(x)) = (x = b_0)$

In our approach, the operation which consists in the creation of a "pair" (variable,value) defined by the application of the variable to the value is not injective if the variable is not supposed to be a type constructor. More precisely, this application can give the same association for different pairs. The definition of injectivity would require here quantification over types and a generalized equality ($\doteq$) between elements of different types. The expression of such a property would state the existence of a set of variables $\mathcal{V}$ such that [2]:

$$\begin{array}{l}\forall (x_1 : *x_1 \rightarrow *s)\ (v_1 : *x_1)\ (x_2 : *x_2 \rightarrow *s)\ (v_2 : *x_2) \\ x_1 \dot{\in} \mathcal{V} \wedge x_2 \dot{\in} \mathcal{V} \Rightarrow x_1(v_1) = x_2(v_2) \Rightarrow (x_1 \doteq x_2) \wedge (v_1 \doteq v_2)\end{array}$$

This formula raises two typing problems:

- the set $\mathcal{V}$ contains elements of different types (x_1, x_2).
- equality is applied between objects of different types $(x_1 \doteq x_2)$.

In fact, we would like to specify that $\mathcal{V}$ is the set of constructors of some type. In order to overcome this problem, we define weaker properties on individual variables by introducing the two polymorphic predicates `IS_VAR` and `D_VAR`.

$\texttt{IS\_VAR}\ (v : *v \rightarrow *s) = \forall\ a\ b.\ (v(a) = v(b)) = (a = b)$
$\texttt{D\_VAR}\ (x : *x \rightarrow *s, y : *y \rightarrow *s) = \forall\ vx\ vy.\ (x(vx) \neq y(vy))$

2.2.2 Array variables

The same formalism can be used to represent arrays. For the following declaration:

```
VAR t: ARRAY [INDEX] of INTEGER;
```

we associate the data type:

```
type declarations =
        t INDEX INTEGER
```

In the same way, this type declaration introduces the function:

$\texttt{t} : \texttt{INDEX} \rightarrow \texttt{INTEGER} \rightarrow \texttt{declarations}$

The state $\texttt{state}_0$ where all the elements of the array `t` are `0` except the one at index i_0 where the value is v_0 is encoded as follows.

$\forall i\ x.\ \texttt{state}_0(t(i, x)) = \text{if } (i = i_0) \text{ then } x = v_0 \text{ else } x = 0$

[1] We note that such a representation allows for multivalued variables. However in this paper we do not use such a feature.
[2] type variables are prefixed by *

2.2.3 Expressions and variable access

Although in our representation a variable may be multivalued, we define expressions as *functions* over states:

```
expression :  *state → *exp_type
```

Then the standard operators (boolean and arithmetic operators) can be lifted to such expressions. For this purpose, we define a unary and a binary polymorphic lifting functions. For instance, we define the `Bop_Lift` function as follows:

`Bop_Lift` $op = \lambda\ e_1\ e_2\ st.\ op\ (e_1(st), e_2(st))$

We use the same convention as [APP93] where the name given to the lifted operator is the name of the original operator suffixed by `*`. For instance, `Bop_Lift (<) = <*`.

Note that the multivaluation of variables could have been extended to expressions. However, in this framework, the definition of boolean algebra operators becomes tricky and some classical properties are lost.

As specific expressions, we define the variable access function `val`. This function must choose *one* of the values associated to the variable by the state. Moreover, in a given state, this choice must be the same for every access in the same expression. For this purpose, we use the Hilbert choice function[GM94] denoted ϵ. The term $\epsilon\ x : \sigma.\ P(x)$ denotes an arbitrary but fixed variable of type σ. This term verifies the predicate P if a term verifying P exists. Then, the polymorphic function `val` is defined as follows:

```
val :  (*v → *decl)  →  (*decl → bool)  →       *v
             v        ,        st         ↦  ϵ  x. st(v(x))
```

Note that `val(v)` is a function from states of type `*decl → bool` to variable values of type `*v`.

3 Statements logic

In order to allow for non determinism, we define statements as binary relations over states[Tre92]. If we consider states as sets of associations variable-value, a statement can also be interpreted as a function which consumes some elements of a set and produces new ones. However, we have not pursued further in this direction. An interesting study would be to explore the links with linear logic [Gir87].

In the following, we give the semantics of some basic statements like assignments, conditional, alternative and parallel statements. In the last section, we establish some well known results and give some examples.

3.1 Assignments

The assignment statement `x := e` establishes a relation between two states st_1 and st_2 such that the value of the variable `x` in st_2 is the value of the expression `e` in st_1, any other variable binding remaining unchanged. The assignment is defined as follows:

$$\begin{array}{rl} x := e \ = \ \lambda\ st_1\ st_2. & \forall\ y.\ st_2(x(y)) = (y = e(e1)) \ \wedge \\ & \forall\ s.\ (\forall\ y.\ s \neq x(y)) \Rightarrow st_2(s) = st_1(s) \end{array}$$

From this definition, we can prove the following theorems which state explicitly the usual properties of the assignment statement:

$$\begin{array}{rl} & (x := e)(st_1, st_2) \Rightarrow \text{val}\ (x)(st_2) = e(st_1) \\ \text{D\_VAR}\ (x, y) \Rightarrow & (x := e)(st_1, st_2) \Rightarrow \text{val}\ (y)(st_2) = \text{val}\ (y)(st_1) \end{array}$$

We have only defined the single assignment statement. The multiple assignment statement will be defined in section 3.4.3 through the parallel constructor $||$.

3.2 Alternatives

The definition of statements as binary relations over states yields a straightforward definition of non deterministic statements as in CSP[Hoa85]. We consider the binary operator $[\!]$ and the generalized one indexed by a set of alternatives A: $[\!]_A$.

$$
\begin{array}{lcl}
i_1 \; [\!] \; i_2 & = & \lambda \; st_1 st_2.\; i_1(st_1, st_2) \;\vee\; i_2(st_1, st_2) \\
[\!]_A i & = & \lambda \; st_1 st_2.\; \bigvee_{a \in A} i_a(st_1, st_2)
\end{array}
$$

The associativity and commutativity of the binary alternative operator are easily proved.

3.3 Conditional Statements

We introduce successively two conditional statements. The first one ($\longrightarrow$) is similar to a guarded statement[Dij76], while the second one (`IF`) is similar to the usual `if-then` statement.

$$
\begin{array}{ll}
i \longrightarrow c & = \lambda \; st_1 \; st_2.\; c(st_1) \wedge i(st_1, st_2) \\
i \text{ IF } c & = \lambda \; st_1 \; st_2.\; \mathbf{if}\; c(st_1) \;\mathbf{then}\; i(st_1, st_2) \;\mathbf{else}\; (st_1 = st_2)
\end{array}
$$

3.4 Parallel statements

The basic idea of the parallel construct is to *superpose* the changes performed by several statements. A variable binding is changed by a parallel statement if it is changed by one of the composing statements; otherwise it remains unchanged. We should note that such a definition of parallelism cannot be reduced to interleaving and non-determinism. For instance the exchange of two variables can be achieved by a parallel composition (§3.6.2) of the two symmetric assignments which is not equivalent to a non deterministic choice between the sequential compositions of the two assignments.

3.4.1 The binary PAR constructor

From the intuitive definition of parallelism, we first define a binary parallel operator $||$ between two statements i_1 and i_2. The parallel statement updates a state so that a binding is present in the final state if it is present in the final state of one of the statements modifying it. This is encoded as follows:

$$
\begin{array}{lll}
i_1 \;||\; i_2 = \lambda \; st \; st'.\; \exists \; st_1 \; st_2. & i1(st, st_1) \wedge i2(st, st_2) \wedge & \\
& \forall s.\; st'(s) = & (st(s) \neq st_1(s) \wedge st_1(s)) \vee \\
& & (st(s) \neq st_2(s) \wedge st_2(s)) \vee \\
& & (st(s) = st_1(s) \wedge st(s) = st_2(s) \wedge st(s))
\end{array}
$$

The previous definition can be simplified to the following one:

$$
\begin{array}{lll}
i_1 \;||\; i_2 = \lambda \; st \; st'.\; \exists \; st_1 \; st_2. & i1(st, st_1) \wedge i2(st, st_2) \wedge & \\
& \forall s.\; st'(s) = \;\mathbf{if}\; st(s) & \mathbf{then}\; st_1(s) \wedge st_2(s) \\
& & \mathbf{else}\; st_1(s) \vee st_2(s)
\end{array}
$$

We should remark that the $||$ constructor requires a point by point interpretation of states contrary to other constructors such as conditional alternatives.

3.4.2 The generalized PAR constructor

The previous definition can be easily extended to an indexed set of statements ins_i in the following way:

$$
\begin{array}{ll}
\text{PAR } ins = \lambda \; st \; st'.\; \exists \; e'. & \forall \; i.\; ins_i(st, e'_i) \wedge \\
& \forall \; s.\; st'(s) = \;\mathbf{if}\; st(s) \;\mathbf{then}\; \forall \; i.\; e'_i(s) \;\mathbf{else}\; \exists \; i.\; e'_i(s)
\end{array}
$$

3.4.3 Multi-assignments

The parallel operator lets us introduce multi-assignment statements as a parallel combination of single assignments. Then, single assignment theorems (§3.1) can be extended, for instance, to two-assignment statements.

$$\text{D\_VAR}\ (x, y)\ \wedge\ (x := ex \parallel y := ey)(st_1, st_2) \Rightarrow$$
$$\text{val}\ (x)(st_2) = ex(st_1) \wedge \text{val}\ (y)(st_2) = ey(st_1)$$

$$\text{D\_VAR}\ (x, z)\ \wedge\ \text{D\_VAR}\ (y, z)\ \wedge\ (x := ex \parallel y := ey)(st_1, st_2) \Rightarrow$$
$$\text{val}\ (z)(st_2) = \text{val}\ (z)(st_1)$$

The previous two rules illustrate the idea of the superposition of changes made by the components of a parallel statement. However, the behaviour of a multi-assignment to a single variable (which is usually syntactically forbidden) is counter intuitive; for instance, let us consider the multi-assignment $x := 1 \parallel x := 2$. If in the initial state the value of x is 0, then in the final state x is multivalued. But, if the initial value of x is 1, then the final value of x is 2! Another semantics could be to allow a non-deterministic behaviour in such cases, or to state explicitly how values are combined[BC84]. Nevertheless, we have done with this definition for its conciseness and since we do not consider parallel updates of the same variable.

At last, general multi-assignment rules cannot be stated because of typing problems: each variable has its own type and a set of such variables cannot be defined. However, we will see in section 5.3 how to overcome this problem by introducing a kind of meta theorem.

3.5 Assessment of the definitions

In order to assess our definitions, we establish some results of well known programming logics within our framework. We consider the weakest precondition semantics of Dijkstra [Dij76]. The following definition expresses the weakest precondition (or predicate transformer) of a statement i and a predicate p:

$$\text{wp}\ (i, p) = \lambda\ e.\ \forall e'.\ i(e, e') \Rightarrow p(e')$$

Then, by defining the substitution of variable x in expression e_1 by expression e_2 as follows:

$$\text{subst}\ (x, e_1, e_2) = \lambda\ st.\ e_2(\lambda\ s.\ \textbf{if}\ \exists z.\ s = x(z)\ \textbf{then}\ s = x(e_1(st))\ \textbf{else}\ st(s))$$

We have the following theorems[3]:

$$\forall x.\ \text{IS\_VAR}\ (x) \Rightarrow\ \text{wp}\ (x := e, \text{FALSE\_P}) = \text{FALSE\_P}$$
$$\forall x\ p.\ \text{IS\_VAR}\ (x) \Rightarrow\ \text{wp}\ (x := e, p) =\ \text{subst}\ (x, e, p)$$

3.6 Some results and examples

3.6.1 Independent parallel statements

Parallel constructs may be transformed into sequential constructs given some independence hypothesis expressed using the following `e_indep_x` predicate:

An expression is said to be independent of a variable x if any assignment to x does not modify its value.

$$\texttt{e\_indep\_x (exp, x)} = \forall st_1\ st_2\ e.\ (x := e)(st_1, st_2) \Rightarrow \texttt{exp}\ (st_1) = \texttt{exp}\ (st_2)$$

The following theorem states the equivalence between a sequential and a parallel assignment.

$$\forall x\ y\ ex\ ey.\ \texttt{IS\_VAR}\ (x) \wedge \texttt{IS\_VAR}\ (y) \wedge \texttt{D\_VAR}\ (x, y) \wedge \text{e\_indep\_x}\ (ey, x) \Rightarrow$$
$$(x := ex \parallel y := ey) = \text{Seq}\ (x := ex, y := ey)$$

3.6.2 Exchange of two variables

Under the hypothesis that x and y are different variables, the following result is a straightforward application of the multi-assignment rules given in section 3.4.3.

$$\text{D\_VAR}\ (x, y) \Rightarrow (x := y \parallel y := x)(st_1, st_2) \Rightarrow \text{val}\ (x)(st_2) = \text{val}\ (y)(st_1) \wedge \text{val}\ (y)(st_2) = \text{val}\ (x)(st_1)$$

[3] FALSE_P is the predicate identically false

4 Program refinements

Several formulations of refinements have been proposed[Nip92]. For our work on the protocols for multiprocessor memories [BFR94], we have found the definition given by [LT87] well suited since it considers transition systems[Arn92]. In the following, we recall the basic definition of refinement then we present some of the results we have established.

4.1 Definitions

4.1.1 Transition systems

We represent a generic transition system as a pair consisting in a set of initial states and a next relation between its states.

Definition 1 (Transition system) *A transition system on an alphabet A is a triplet (Σ, Q_0, T) where*

- *Σ is a state space,*
- *Q_0, the set of initial states, is a subset of Σ,*
- *T, the set of labelled transitions, is a subset of $A \times \Sigma \times \Sigma$.*

Notation 1 *A triplet (l, e, e') of the set of labelled transitions of a system S will be denoted $e \xrightarrow{l}_S e'$.*

Definition 2 (Initial states) *Let $S = (\Sigma, Q_0, T)$ be a transition system. Then* $\texttt{Init}_S$ *characterizes the initial states of the transition system: for any state e of Σ,* $\texttt{Init}_S$ *e is true iff $e \in Q_0$.*

Definition 3 (Next Relation) *Let $S = (\Sigma, Q_0, T)$ be a transition system. Then* $\texttt{Next}_S$ *(also denoted $\rightarrow_S$) is a relation defined over Σ such that $e \rightarrow_S e'$ iff there exits a label* `l` *such that $e \xrightarrow{l}_S e'$.*

4.1.2 Refinement relation

Definition 4 (Refinement relation) *Given two transition systems* `s` *and* `s'`, `s` *simulates* `s'` *iff there exits a* simulation relation *such that:*

- *to each initial state of* `s` *at least one initial state of* `s'` *corresponds,*
- *if from a state* `e1`, `s` *can move to* `e2`, *and* `e1` *corresponds to the state* `e1'` *of* `s'` *then there exists a state* `e2'` *such that* `e2'` *corresponds to* `e2` *and* `s'` *can move from* `e1'` *to* `e2'`.

We formalize such a definition as follows:

$$
\begin{array}{l}
\forall\ \texttt{s s'. s Sim s' = } \exists\ \texttt{R.} \\
\quad \forall\ \texttt{e. Init s e} \Longrightarrow \exists\ \texttt{e'. R e e'} \wedge \texttt{Init s' e'} \wedge \\
\quad \forall\ \texttt{e1 e1' e2. Next}_s\ \texttt{e1 e2} \wedge \texttt{R e1 e1'} \Longrightarrow \exists\ \texttt{e2'. R e2 e2'} \wedge \texttt{Next}_{s'}\ \texttt{e1' e2'}
\end{array}
$$

4.2 Refinements through operators

In the previous section, the simulation relation is defined in terms of the initial states and the `Next` relation of a transition system. However, refinement is usually proved independently for each label (also called operator in the following).

Definition 5 (Binary relation refinement) *A relation R (denoted $\xrightarrow{R}$) is a refinement of a relation R' through a relation φ ($R \sqsubseteq_\varphi R'$) iff*

$$\forall e1\ e2\ e1'.\ e1 \xrightarrow{R} e2\ \wedge\ \varphi\ e1 e1' \Longrightarrow \exists e2'.\ e1' \xrightarrow{R'} e2'\ \wedge\ \varphi\ e2 e2'$$

We extend this binary refinement relation to transition systems where labels are interpreted as binary relations over the system states. We can now link transition system simulation and refinement between their respective labeled transitions.

Theorem 1 (Refinement through operators) *Given two transition systems $S^1(\Sigma^1, Q_0^1, T^1)$ and $S^2(\Sigma^2, Q_0^2, T^2)$ with the same alphabet A, a sufficient condition for S^2 to simulate S^1 is that there exists a relation φ such that:*

- *for each label $l \in A$, $\xrightarrow{l}_{S^2} \sqsubseteq_\varphi \xrightarrow{l}_{S^1}$*
- *$\forall e \in Q_0^2$, $\exists e' \in Q_0^1$ such that $\varphi\, e\, e'$*

4.3 Refinement of sequential statements

In [BFR94], we have already presented refinement rules for the conditional and alternative constructs. In this paper, we investigate two new refinement rules for the sequential construct.

- The first one allows the independent refinement of the elements of a sequence.
- The second one allows the refinement of a sequence by a single statement that refines independently each element of the sequence.

4.3.1 Independent refinement

A sequence of statements can be refined by refining each of its components separately:

$$\frac{i_1 \sqsubseteq_\varphi i'_1 \quad i_2 \sqsubseteq_\varphi i'_2}{\text{Seq }(i_1, i_2) \sqsubseteq_\varphi \text{Seq }(i'_1, i'_2)}$$

Sketch of the proof:
The figure 1 represents the states connected by the sequential construct and its components as well as the states connected by the refinement relation φ.

Given st_1, st_3, st'_1 such that $Seq(i_1, i_2)(st_1, st_3)$ and $\varphi(st_1, st'_1)$, we look for a state st'_3 such that $st'_1 \xrightarrow{\text{Seq}(i'_1, i'_2)} st'_3$ and $\varphi(st_3, st'_3)$.

By the first refinement hypothesis: $i_1 \sqsubseteq_\varphi i'_1$, there exists a state st'_2 such that $st'_1 \xrightarrow{i'_1} st'_2$ and $\varphi(st_2, st'_2)$ and finally, by the second refinement hypothesis: $i_2 \sqsubseteq_\varphi i'_2$, there exists a state st'_3 such that $st'_2 \xrightarrow{i'_2} st'_3$ and $\varphi(st_3, st'_3)$.

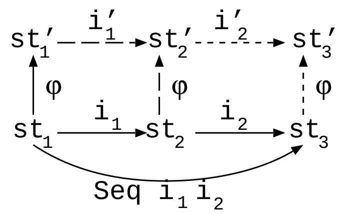

Figure 1: Independent SEQ refinement

4.3.2 Joint refinement

We first introduce the predicate `stable` (φ, i):

Definition 6 (Stability) *The refinement relation φ is invariant through the statement i.*

$$stable\ (\varphi, i) = \forall st\ st'_1\ st'_2.\ \varphi(st, st'_1) \wedge i(st'_1, st'_2) \Rightarrow \varphi(st, st'_2)$$

The idea of the joint refinement rule is to split the refinement of a sequence of statements by a single statement into the refinement of each component through a *projection* of the initial refinement operator.

$$\frac{i \sqsubseteq_{\varphi_1} i'_1 \quad i \sqsubseteq_{\varphi_2} i'_2 \quad \text{stable } (\varphi_1, i'_2) \quad \text{stable } (\varphi_2, i'_1)}{i \sqsubseteq_{\varphi_1 \wedge \varphi_2} \text{Seq } (i'_1, i'_2)}$$

Sketch of the proof:

The figure 2 represents the states connected by the sequential construct and its components as well as the states connected by the refinement relation φ.

Given st_1, st_2, st'_1 such that $st_1 \xrightarrow{i} st_2$, $\varphi_1(st_1, st'_1)$ and $\varphi_2(st_1, st'_1)$, we look for a state st'_3 such that $st'_1 \xrightarrow{\text{Seq}(i'_1, i'_2)} st'_3$ and $\varphi_i(st_2, st'_3)$.

- By the refinement hypothesis $i \sqsubseteq_{\varphi_1} i'_1$, there exists a state st'_2 such that $st'_1 \xrightarrow{i'_1} st'_2$ and $\varphi_1(st_2, st'_2)$.
- By the stability hypothesis `stable` (φ_2, i'_1), we have $\varphi_2(st_1, st'_2)$.
- By the refinement hypothesis $i \sqsubseteq_{\varphi_2} i'_2$, there exists a state st'_3 such that $st'_2 \xrightarrow{i'_2} st'_3$ and $\varphi_2(st_2, st'_3)$.
- By the stability hypothesis `stable` (φ_1, i'_2), we have $\varphi_1(st_2, st'_3)$.
- Thus there exists a state st'_3 satisfying the refinement requirements.

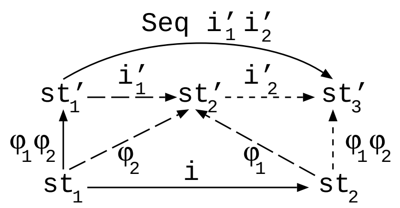

Figure 2: Joint SEQ refinement

4.4 Refinement of parallel constructs

4.4.1 Binding invariance

A binding is invariant or unchanged through a statement i if it has the same status in any states connected by i.

$$\texttt{unchanged}\ (s, i) = \forall st_1\ st_2.\ i(st_1, st_2) \Rightarrow st_1(s) = st_2(s)$$

We can prove the intuitive result concerning invariance through a parallel statement: if i_1 and i_2 are functional, we have:

$$\texttt{unchanged}\ (s, i_1 \parallel i_2) = \texttt{unchanged}\ (s, i_1) \wedge \text{unchanged}\ (s, i_2)$$

4.4.2 Projections

We define the projection of a relation φ with respect to a statement i (denoted by φ_i) as a relation which only depends on bindings unchanged by i.

$$\varphi_i = \lambda\ st\ st'.\ \exists\ st''.\ \varphi(st, st'') \wedge \forall s.\ \texttt{unchanged}\ (s, i) \Rightarrow st'(s) = st''(s)$$

Then, the projected relation is easily shown to be stable by the statement i.

$$\text{stable}\ (\varphi_i, i)$$

Furthermore, if φ is functional and if the statements i_1 and i_2 are independent (they cannot change a binding together[4]), then φ is the conjunction[5] of its projections over i_1 and i_2.

$$\texttt{IsF}\ (\varphi) \wedge \texttt{indep}\ (i_1, i_2) \Rightarrow \varphi = \text{And}\ (\varphi_{i_1}, \varphi_{i_2})$$

4.4.3 A restricted parallel construct

In order to allow a smooth proof of the following theorem, we introduce a new parallel construct denoted $\|\!|$. We define it using the `unchanged` global predicate.

$$\begin{aligned}(i_1 \|\!| i_2)(st_1, st_2) = \exists st'\ st''.\ \ & i_1(st_1, st') \wedge i_2(st_1, st'') \wedge \\ & \forall s\ .\ st_2(s) = \ (\texttt{unchanged}\ (s, i_1) \wedge st''(s)) \vee \\ & \qquad\qquad\quad (\texttt{unchanged}\ (s, i_2) \wedge st'(s))\end{aligned}$$

The following theorem establishes the relation between the two parallel constructs:

$$\forall st_1\ st_2\ i_1\ i_2.\ \texttt{indep}(i_1, i_2) \Rightarrow (i_1 \|\!| i_2)(st_1, st_2) \Rightarrow (i_1 \| i_2)(st_1, st_2)$$

Under the same independence hypothesis, the constructor $\|\!|$ can also be defined as follows:

$$\begin{aligned}\texttt{indep}\ (i_1, i_2) \Rightarrow\ \ & (i_1 \|\!| i_2)(st_1, st_2) = \\ & \exists st'\ st''.\ \ i_1(st_1, st') \wedge i_2(st_1, st'') \wedge \\ & \qquad\qquad \forall s\ .\ st_2(s) = \texttt{if unchanged}\ (s, i_1)\ \texttt{then}\ st''(s)\ \texttt{else}\ st'(s)\end{aligned}$$

4.4.4 Refinement theorem

The following refinement rule states that the refinement of a parallel construct can be split into the refinement of its components given some hypothesis.

$$\frac{\texttt{IsF}\ (\varphi) \quad \texttt{indep}\ (i'_1, i'_2) \quad i \sqsubseteq_{\varphi_{i'_2}} i'_1 \quad i \sqsubseteq_{\varphi_{i'_1}} i'_2}{i \sqsubseteq_{\varphi} i'_1 \| i'_2}$$

Sketch of the proof:

The figure 3 represents the states connected by the parallel construct and its components, as well as the refinement relation and its projections over the two statements i'_1 and i'_2. The refinement hypothesis provides the states st'_2 and st''_2 and the refinement relations $\varphi_{i'_1}(st_2, st'_2)$ and $\varphi_{i'_2}(st_2, st''_2)$. Now, the state st'''_2 can be introduced as the superposition of the states st'_2 and st''_2, as defined in the $\|\!|$ construct:

$$\forall s.\ st'''_2(s) = \text{if unchanged}\ (s, i'_1)\ \text{then}\ st''_2(s)\ \text{else}\ st'_2(s)$$

Thus, we have the relation $(i'_1 \|\!| i'_2)(st'_1, st'''_2)$ and by the theorem relating the two parallel constructs, we have $(i'_1 \| i'_2)(st'_1, st'''_2)$.

Then, we prove that the two projected refinement relations remain valid between the states st_2 and st'''_2. So is their conjunction φ.

[4] indep $(i_1, i_2) = \forall s.$ unchanged $(s, i_1) \vee$ unchanged (s, i_2)

[5] And $(\varphi_1, \varphi_2) = \lambda\ x\ y.\ \varphi_1(x, y) \wedge \varphi_2(x, y)$

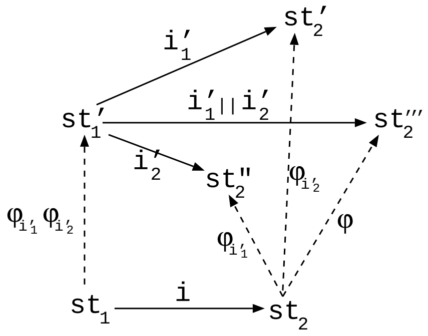

Figure 3: Parallel independent refinement

5 Refinements rules and tactics

We use a theorem prover first to validate the previously presented results and second to offer a framework to support the development of correct parallel algorithms.

5.1 The HOL interactive theorem prover

HOL[GM94] is an interactive theorem prover based on higher order logic. The formulation of this logic relies on a small number of axioms, inference and definition rules. Although HOL was initially intended for hardware design, it is now widely used for general program proving[CT94, Bac92].

5.2 HOL refinement theories

All the refinement theory we have developed is definitional. Starting with the initial theory, we have elaborated the presented framework without introducing any axiom. Consequently, since the initial HOL theory is consistent, the developed one is consistent as well. The figure 4 illustrates the hierarchy of the theories.

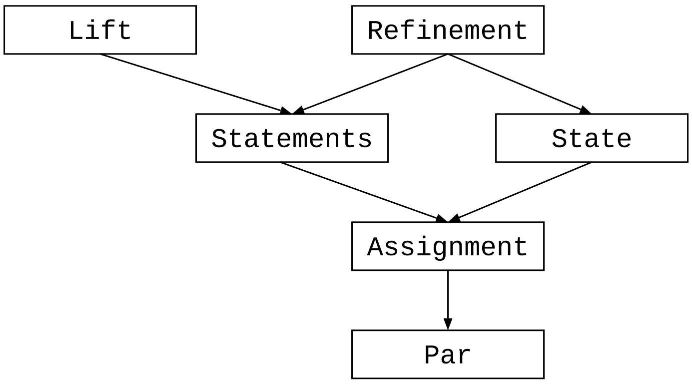

Figure 4: HOL hierarchy of theories

- The Lift theory (§2.2.3) defines the standard operators lifted to functions over states as well as the proof of lifted standard theorems on booleans, sets ...
- The Refinement theory (§4) defines basic relational algebra, the refinement relation and basic refinement theorems.
- The State theory (§2.2) defines environments and variables.

- The Statements theory (§3) defines basic sequential control structures and refinement theorems on them.
- The Assignment theory (§3.1) defines the assignment statement, theorems on environments modified through assignments.
- The Par theory (§3.4.3) defines the parallel operator, theorems concerning independent refinements of parallel statements as well theorems and conversions on multi-assignments.

5.3 Meta theorems in HOL

The theorems stated in this paragraph assume that we have a data type defined by an indexed set of constructors. Such a data type cannot be characterized in HOL since this would require at least sets with elements of different types. Consequently generals theorems about these data types cannot be stated rigorously in HOL. However, HOL *conversions* provide a way to generate specific instances of these theorems for a given data type.

In the following, we suppose that we have a data type `Decl` and an indexed family of typed constructors $(C_i : T_i \rightarrow \texttt{Decl})_{i \in I}$ such that:

- $\forall i \in I.\ \texttt{IS\_VAR}\ (C_i)$
- $\forall i\ j \in I.\ i \neq j \Rightarrow \texttt{D\_VAR}(C_i, C_j)$

An example of such a data type `Decl` coming from the study of the refinement of multiprocessor memory models, describing an array of cache and status registers, and a global memory is given below:

```
C_decls = VAL *ind *val
        | STATE *ind *state
        | MEM *val
```

This type declaration defines three families of constructors: `VAL (i)`, `STATE (i)` and `MEM`.

5.3.1 Assignments

The single assignment meta-rule applied to an assignment statement expresses the value of each variable in the new state in terms of expressions evaluated in the current state.

$$\bigwedge_{i \in I} \left\{ \begin{array}{ll} \forall e : T_i.\ (C_i := e)(st_1, st_2) = & \forall x : T_i.\ st_2(C_i(x)) = (x = e(st_1)) \wedge \\ & \bigwedge_{j \neq i}\ \forall x : T_j.\ st_2(C_j(x)) = st_1(C_j(x)) \end{array} \right.$$

We illustrate the application of this meta rule on a data type on the data type introduced in the previous paragraph. The HOL conversion associated to the single assignment meta rule proves the following theorem:

```
|- (∀ x0 ex e1 e2.
     ((VAL x0) := ex)e1 e2 =
       (∀ x0' x1. e2(VAL x0' x1) = ((x0' = x0) => (x1 = ex e1) | e1(VAL x0' x1))) ∧
       (∀ x0' x1. e2(STATE x0' x1) = e1(STATE x0' x1)) ∧
       (∀ x. e2(MEM x) = e1(MEM x))) ∧

   (∀ x0 ex e1 e2.
     ((STATE x0) := ex)e1 e2 =
       (∀ x0' x1. e2(VAL x0' x1) = e1(VAL x0' x1)) ∧
       (∀ x0' x1. e2(STATE x0' x1) = ((x0' = x0) => (x1 = ex e1) | e1(STATE x0' x1))) ∧
       (∀ x. e2(MEM x) = e1(MEM x))) ∧

   (∀ ex e1 e2.
     (MEM := ex)e1 e2 =
       (∀ x0 x1. e2(VAL x0 x1) = e1(VAL x0 x1)) ∧
       (∀ x0 x1. e2(STATE x0 x1) = e1(STATE x0 x1)) ∧
       (∀ x. e2(MEM x) = (x = ex e1)))
```

5.3.2 Multi-Assignments

The multi-assignment meta rule is similar to the single assignment one except that the effects of the different assignments are superposed. The meta rule is formulated as follows:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\forall e_1 : T_1 \cdots e_n : T_n} \\
\multicolumn{2}{l}{\quad (C_{i_1} := e_1 \parallel \cdots \parallel C_{i_n} := e_n)(st_1, st_2) =} \\
 & \forall x : T_{i_1}.\ st_2(C_{i_1}(x)) = (x = e_1(st_1)) \wedge \\
 & \qquad \vdots \\
 & \forall x : T_{i_n}.\ st_2(C_{i_n}(x)) = (x = e_n(st_1)) \wedge \\
\bigwedge_{i \notin \{i_1, \cdots, i_n\}} & \forall x : T_i.\ st_2(C_i(x)) = st_1(C_i(x))
\end{array}
$$

We illustrate the application of the multi-assignment rule on the same data type. Here, the HOL conversion needs the variables assigned to. The generation of all combinations of assignments is not realistic. Here, the conversion proves the following theorem defining the multi-assignment of the three variables `MEM`, `STATE p` and `VAL q`:

```
|- (MEM := m) || (STATE p := s) || (VAL q := v) =
   (λ e1 e2.
     (∀ x0 x1. e2 (VAL x0 x1) = ((x0 = q) => (x1 = v e1) | e1(VAL x0 x1))) ∧
     (∀ x0 x1. e2 (STATE x0 x1) = ((x0 = p) => (x1 = s e1) | e1(STATE x0 x1))) ∧
     (∀ x. e2 (MEM x) = (x = m e1)))
```

5.3.3 Refinements tactics

Refinement theorems concerning the sequential and parallel constructs have been also implemented in HOL as proof tactics. Thus, these tactics can be used by the backward proof engine to reduce a goal into subgoals. To give the flavor of the application of a tactic, we have peeked a fragment from the proofs developed for the study of refinements between multiprocessor memory models.

Suppose we have to prove the following refinement property:

```
∀ p st ost states. st IN states ∧ ¬ ost IN states ⇒
  REF_Op
  ((((STATE p) := (CST st)) || ((VAL p) := (val MEM))) IF ((val(STATE p)) =* (CST ost)))
  ((A_STATE := (REPLACE_B_P(val A_STATE)(CST ost)(CST st))) || (A_VAL := ((val A_MEM) INSERT_P (val A_VAL))))
  (λ e1 e2. e2 = C2Af states e1)
```

The application of the parallel-sequential transformation of independent assignments (section 3.6.1) yields the following subgoal:

```
% Goal %
REF_Op
 ((((STATE p) := (CST st)) || ((VAL p) := (val MEM))) IF ((val(STATE p)) =* (CST ost)))
 (Seq (A_STATE := (REPLACE_B_P(val A_STATE)(CST ost)(CST st))) (A_VAL := ((val A_MEM) INSERT_P (val A_VAL))))
 (And (Pr2_Phi_X(λ x y. y = C2Af states x)A_VAL) (Pr2_Phi_X(λ x y. y = C2Af states x)A_STATE))"

% Hypothesis %
  2  ["st IN states" ]
  1  ["¬ ost IN states" ]
```

Then, the application of the tactic associated to the joint sequential refinement theorem of section 4.3.2 splits the current goal into four subgoals:

```
% Subgoal 4 %
REF_Op
 ((((STATE p) := (CST st)) || ((VAL p) := (val MEM))) IF ((val(STATE p)) =* (CST ost)))
 (A_VAL := ((val A_MEM) INSERT_P (val A_VAL)))
 (Pr2_Phi_X(λ x y. y = C2Af states x)A_STATE)
  2  ["st IN states" ]
  1  ["¬ ost IN states" ]

% Subgoal 3 %
REF_Op
 ((((STATE p) := (CST st)) || ((VAL p) := (val MEM))) IF ((val(STATE p)) =* (CST ost)))
 (A_STATE := (REPLACE_B_P(val A_STATE)(CST ost)(CST st)))
 (Pr2_Phi_X(λ x y. y = C2Af states x)A_VAL)
  2  ["st IN states" ]
  1  ["¬ ost IN states" ]

% Subgoal 2 %
stable
 (Pr2_Phi_X(λ x y. y = C2Af states x)A_STATE)
 (A_STATE := (REPLACE_B_P(val A_STATE)(CST ost)(CST st)))
  2  ["st IN states" ]
  1  ["¬ ost IN states" ]

% Subgoal 1%
stable
 (Pr2_Phi_X(λ x y. y = C2Af states x)A_VAL)
 (A_VAL := ((val A_MEM) INSERT_P (val A_VAL)))
  2  ["st IN states" ]
  1  ["¬ ost IN states" ]
```

6 Conclusion

The start point of this paper is the proposal of a new state representation. We believe that it should reduce the gap between the syntactic and the semantic levels. Proofs do not need any new information with respect to the syntactic specification of the program, such as type informations. Within this framework, we have restated and validated the usual programming constructs. Another important aspect of our work has been concerned with the refinement of sequential and parallel constructs. All these results have been validated by their embedding in the HOL theorem prover.

The next step of our work is to use the results presented here to validate consistency cache protocols. We are especially interested in generic models and their relations.

This paper was mainly concerned with local properties of programs. A further study would be to consider state sequences for reasoning about behavioral program properties.

References

[APP93] F. Andersen, K. D. Petersen, and J.S. Pettersson. Program verification using HOL-UNITY. In *Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[Arn92] A. Arnold. *Systèmes de transitions finis et sémantiques des processus communicants*. Etudes et recherches en informatique. MASSON, 1992.

[Bac92] R. J. R. Back. Refinement calculus, lattices, and higher order logic. In *Program design calculi*, volume 118 of *NATO ASI Series F. Computer and systems sciences*, pages 53–72. Springer Verlag, 1992.

[BC84] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. volume 197 of *Lecture Notes in Computer Science*, pages 389–448, Berlin, Germany, 1984. Springer-Verlag.

[BFR94] J.-P. Bodeveix, M. Filali, and P. Roché. Towards a HOL theory of memory. In *Higher Order Logic Theorem Proving and its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 49–64. Springer-Verlag, sep 1994.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.

[CT94] C. Ching-Tsun. Mechanical verification of distributed algorithms in higher order logic. In *Higher Order Logic Theorem Proving and its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 158–176. Springer-Verlag, 1994.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Englewood Cliffs New Jersey: Prentice Hall, 1976.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Comp. Science*, 50:1–102, 1987.

[GM94] M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, 1994.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[LT87] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM symposium on principles of distributed computing*, pages 137–151, aug 1987.

[Nip92] T. Nipkow. Formal verification of data type refinement - theory and practice. In *Stepwise refinement of distributed systems*, volume 430 of *Lecture Notes in Computer Science*, pages 561–591. Springer Verlag, 1992.

[Tre92] G. Tredoux. Mechanizing execution sequence semantics in HOL. *South African Computer Journal*, (7), July 1992.

[vW90] J. von Wright. *A lattice-theoretical basis fro program refinement*. PhD thesis, Abo Akademi Finland, 1990.