Isabelle/UTP Tutorial

Simon Foster Frank Zeyda

March 24, 2017

Contents

1 Isabelle/UTP Primer **1**
1.1 State-spaces and lenses 1
1.2 Predicate calculus 2
1.3 Meta-logical operators 3
1.4 Relations 4
1.5 Non-determinism and Complete Lattices 5
1.6 Laws of programming 7
1.7 Designs 9
1.8 Reactive Designs 9

1 Isabelle/UTP Primer

In this section, we will introduce Hoare and He's *Unifying Theories of Programming* [7] through a tutorial about our mechanisation, in Isabelle, called Isabelle/UTP [5, 3, 9]. The UTP is a framework for building and reasoning about heterogeneous semantics of programming and modelling languages. One of the core ideas of the UTP is that any program (or model) can be represented as a logical predicate over the program's state variables. The UTP thus begins from a higher-order logical core, and constructs a semantics for imperative relational programs, which can then be refined and extended with more complex language paradigms and theories. Isabelle/UTP mechanises this language of predicates and relations, and provides proof tactics for solving conjectures. For example, we can prove the following simple conjectures:

lemma $(true \wedge false) = false$
by *pred-auto*

lemma $(true \Rightarrow P \wedge P) = P$
by (*pred-auto*)

We discharge these using our predicate calculus tactic, *pred-auto*. It should be noted that *true*, *false*, and the conjunction operator are not simply the HOL operators; rather they act on on our UTP predicate type ($'\alpha$ *upred*).

1.1 State-spaces and lenses

Predicates in the UTP are alphabetised, meaning they specify beheaviours in terms of a collection of variables, the alphabet, which effectively gives a state-space for a particular program. Thus the type of UTP predicates $'\alpha$ *upred* is parametric in the alphabet $'\alpha$. In Isabelle/UTP we can create a particular state-space with the **alphabet** command:

alphabet *myst* =
  x :: *int*
  y :: *int*
  z :: *int set*

This command creates an alphabet with three variables, x, y, and z, each of which has a defined type. A new Isabelle type is created, *myst*, which can be then used as the parameter for our predicate model, e.g. *myst upred*. In the context of our mechanisation, such variables are represented using *lenses* [5, 4]. A lens, $X : V \Longrightarrow S$, is a pair of functions, $get :: V \to S$ and $put : S \to V \to S$, where S is the source type, and V is the view type. The source type represents a "larger" type that can in some sense be subdivided, and the view type a particular region of the source that can be observed and manipulated independently of the rest of the source.

In Isabelle/UTP, the source type is the state space, and the view type is the variable type. For instance, we here have that x has type $int \Longrightarrow myst$ and z has type $int\ set \Longrightarrow myst$. Thus, performing an assignment to x equates to application of the *put* function, and looking up the present valuation is application of the *get* function.

Since the different variable characterise different regions of the state space we can distinguish them using the independence predicate $x \bowtie z$. Two lenses are independent if they characterise disjoint regions of the source type. In this case we can prove that the two variables are different using the simplifier:

lemma $x \bowtie z$
  by *simp*

However, we cannot prove, for example, that $x \bowtie x$ of course since the same region of the state-space is characterised by both. Lenses thus provide us with a semantic characterisation of variables, rather than a syntactic notion. For more background on this use of lenses please see our recent paper [5].

1.2 Predicate calculus

We can now use this characterisation of variables to define predicates in Isabelle/UTP, for example $\&x >_u \&y$, which corresponds to all valuations of the state-space in which x is greater than y. Often we have to annotate our variables to help Isabelle understand that we are referring to UTP variables, and not, for example, HOL logical variables. In this case we have to decorate the names with an ampersand. Moreover, we often have to annotate operators with u subscripts to denote that they refer to the UTP version of the operator, and not the HOL version. We can now write down and prove a simple proof goal:

lemma ‘$(\&x =_u 8 \wedge \&y =_u 5) \Rightarrow \&x >_u \&y$‘
  by (*pred-auto*)

The backticks denote that we are writing an tautology. Effectively this goal tells us that $x = 8$ and $y = 5$ are valid valuations for the predicate. Conversely the following goal is not provable.

lemma ‘$(\&x =_u 5 \wedge \&y =_u 5) \Rightarrow \&x >_u \&y$‘
  apply (*pred-simp*) — Results in *False*
oops

We can similarly quantify over UTP variables as the following two examples illustrate.

lemma $(\exists\ x \cdot \&x >_u \&y) = true$
  by (*pred-simp*, *presburger*)

lemma $(\forall\ x \cdot \&x >_u \&y) = \mathit{false}$
by (*pred-auto*)

The first goal states that for any given valuation of y there is a valuation of x which is greater. Predicate calculus alone is insufficient to prove this and so we can also use Isabelle's *sledgehammer* tool [1] which attempts to solve the goal using an array of automated theorem provers and SMT solvers. In this case it finds that Isabelle's tactic for Presburger arithmetic can solve the goal. In this second case we have a goal which states that every valuation of x is greater than a given valuation of y. Of course, this isn't the case and so we can prove the goal is equivalent to *false*.

1.3 Meta-logical operators

In addition to predicate calculus operators, we also often need to assert meta-logical properties about a predicate, such as "variable x is not present in predicate P". In Isabelle/UTP we assert this property using the *unrestriction* operator, e.g. $x \sharp \mathit{true}$. Here are some examples of its use, including discharge using our tactic *unrest-tac*.

lemma $x \sharp \mathit{true}$
by (*unrest-tac*)

lemma $x \sharp (\&y >_u 6)$
by (*unrest-tac*)

lemma $x \sharp (\forall\ x \cdot \&x =_u \&y)$
by (*unrest-tac*)

The tactic attempts to prove the unrestriction using a set of built-in unrestriction laws that exist for every operator of the calculus. The final example is interesting, because it shows we are not dealing with a syntactic property but rather a semantic one. Typically, one would describe the (non-)presence of variables syntactically, by checking if the syntax tree of P refers to x. In this case we are actually checking whether the valuation of P depends on x or not. In other words, if we can rewrite P to a form where x is not present, but P is otherwise equivalent, then x is unrestricted – it can take any value. The following example illustrates this:

lemma $x \sharp (\&x <_u 5 \lor \&x =_u 5 \lor \&x >_u 5)$
by (*pred-auto*)

Of course, if x is either less than 5, equal to 5, or greater than 5 then x can take any value and the predicate will still be satisfied. Indeed this predicate is actually equal to *true* and thus x is unrestricted. We will often use unrestriction to encode necessary side conditions on algebraic laws of programming.

In addition to presence of variables, we will often want to substitute a variable for an expression. We write this using the familiar syntax $P[\![v/x]\!]$, and also $P[\![v_1,v_2,v_3/x_1,x_2,x_3]\!]$ for an arbitrary number of expressions and variables. We can evaluate substitutions using the tactic *subst-tac* as the following examples show:

lemma $(\&y =_u \&x)[\![2/x]\!] = (\&y =_u 2)$
by (*subst-tac*)

lemma $(\&y =_u \&x \land \&y \in_u \&z)[\![2/y]\!] = (2 =_u \&x \land 2 \in_u \&z)$
by (*subst-tac*)

lemma $(\exists\ \&x \cdot \&x \in_u \&z)[\![76/\&x]\!] = (\exists\ \&x \cdot \&x \in_u \&z)$
by (*subst-tac*)

lemma $true[\![1,2/\&x,\&y]\!] = true$
by (*subst-tac*)

We can also, of course, combine substitution and predicate calculus to prove conjectures containing substitutions.

lemma $(\&x =_u 1 \wedge \&y =_u \&x)[\![2/x]\!] = false$
apply (*subst-tac*)
apply (*pred-auto*)
done

So far, we have considered UTP predicates which contain only UTP variables. However it is possible to have another kind of variable – a logical HOL variable which is sometimes known as a "logical constant" [8]. Such variables are not program or model variables, but they simply exist to assert logical properties of a predicate. The next two examples compare UTP and HOL variables in a quantification.

lemma $(\forall\ x \cdot \&x =_u \&x) = true$
by (*pred-auto*)

lemma $(\forall\ x \cdot \ll x \gg =_u \ll x \gg) = true$
by (*pred-auto*)

The first quantification is a quantification of a UTP variable, which we've already encountered. The second is a quantifier over a HOL variable, denoted by the quantifier being bold. In addition we refer to HOL variables, not using the ampersand, but the quotes $\ll k \gg$. These quotes allow us to insert an arbitrary HOL term into a UTP expression, such as a logical variable.

1.4 Relations

Relations, $('\alpha, '\beta)\ rel$, are a class of predicate in which the state space is a product – i.e. $('\alpha, '\beta)\ rel$ – and divides the variables into input or "before" variables and output or "after" variables. In Isabelle/UTP we can write down a relational variable using the dollar notation, as illustrated below:

term $(\$x' =_u 1 \wedge \$y' =_u \$y \wedge \$z' =_u \$z) :: myst\ hrel$

Type $'\alpha\ hrel$ is the type of homogeneous relations, which have the same before and after state. This example relation can be intuitively thought of as the relation which sets x to 1 and leaves the other two variables unchanged. We would normally refer to this as an assignment of course, and for convenience we can write such a predicate using a more convenient syntax, $x := 1$, which is equivalent:

lemma $(x := 1) = ((\$x' =_u 1 \wedge \$y' =_u \$y \wedge \$z' =_u \$z) :: myst\ hrel)$
by (*rel-auto*)

Since we are now in the world of relations, we have an additional tactic called *rel-auto* that solves conjectures in relational calculus. We can use relational variables to write to loose specifications for programs, and then prove that a given program is a refinement. Refinement is an order on programs that allows us to assert that a program refines a given specification, for example:

lemma $(\$x' >_u \$y) \sqsubseteq (x,\ y := \&y + 3,\ 5)$
by (*rel-auto*)

This tells us that the specification that the after value of x must be greater than the initial value of y, is refined by the program which adds 3 to y and assigns this to x, and simultaneously assigns 5 to y. Of course, this is not the only refinement, but an interesting one. A refinement conjecture $P \sqsubseteq Q$ in general asserts that Q is more deterministic than P. In addition to assignments, we can also construct relational specifications and programs using sequential (or relational) composition:

lemma $(x := 1 \;;; x := \&x + 1) = (x := 2)$
  by (*rel-auto*)

Internally, what is happening here is quite subtle, so we can also prove this law in the Isar proof scripting language which allows us to further expose the details of the argument. In this proof we will make use of both the tactic and already proven laws of programming from Isabelle/UTP.

lemma $(x := 1 \;;; x := \&x + 1) = (x := 2)$
proof −
  — We first show that a relational composition of an assignment and some program P corresponds to substitution of the assignment into P, which is proved using the law *assigns-r-comp*.
  have $(x := 1 \;;; x := \&x + 1) = (x := \&x + 1)[\![1/\$x]\!]$
    by (*simp add*: *assigns-r-comp alpha*)
  — Next we execute the substitution using the relational calculus tactic.
  also have ... $= x := 1 + 1$
    by (*rel-auto*)
  — Finally by evaluation of the expression, we obtain the desired result of 2.
  also have ... $= x := 2$
    by (*simp*)
  finally show *?thesis* **.**
qed

UTP also gives us an if-then-else conditional construct, written $P \lhd b \rhd Q$, which is a more concise way of writing **if** b **then** P **else** Q. It also allows the expression of while loops, which gives us a simple imperative programming language.

lemma $(x := 1 \;;; (y := 7 \lhd \$x >_u 0 \rhd y := 8)) = (x,y := 1,7)$
  by (*rel-auto*)

Below is an illustration of how we can express a simple while loop in Isabelle/UTP.

term $(x,y := 3,1 \;;; \mathit{while}\ \&x >_u 0\ \mathit{do}\ x := \&x - 1 \;;; y := \&y * 2\ \mathit{od})$

1.5 Non-determinism and Complete Lattices

So far we have considered only deterministic programming operators. However, one of the key feature of the UTP is that it allows non-deterministic specifications. Determinism is ordered by the refinement order $P \sqsubseteq Q$, which states that P is more deterministic that Q, or alternatively that Q makes fewer commitments than P. The refinement order $P \sqsubseteq Q$ corresponds to a universally closed implication $Q \Rightarrow P$. The most deterministic specification is *false*, which also corresponds to a miraculous program, and the least is *true*, as the following theorems demonstrate.

theorem *false-greatest*: $P \sqsubseteq \mathit{false}$
  by (*rel-auto*)

theorem *true-least*: $\mathit{true} \sqsubseteq P$
  by (*rel-auto*)

In this context *true* corresponds to a programmer error, such as an aborting or non-terminating program (the theory of relations does not distinguish these). We can similarly specify a non-deterministic choice between P and Q with $P \sqcap Q$, or alternatively $\bigsqcap A$ where A is a set of possible behaviours. Predicate $P \sqcap Q$ encapsulates the behaviours of both P and Q, and is thus refined by both. We can also prove a variety of theorems about non-deterministic choice.

theorem *Choice-equiv*:
fixes $P\ Q :: {'\alpha}\ upred$
shows $\bigsqcap \{P, Q\} = P \sqcap Q$
by *simp*

Theorem *Choice-equiv* shows the relationship between the big choice operator and its binary equivalent. The latter is simply a choice over a set with two elements.

theorem *Choice-refine*:
fixes $A\ B :: {'\alpha}\ upred\ set$
assumes $B \subseteq A$
shows $\bigsqcap A \sqsubseteq \bigsqcap B$
by (*simp add*: *Sup-subset-mono assms*)

The intuition of theorem *Choice-refine* is that a specification with more options is refined by one with less options. We can also prove a number of theorems about the binary version of the operator.

theorem *choice-thms*:
fixes $P\ Q :: {'\alpha}\ upred$
shows
$P \sqcap P = P$
$P \sqcap Q = Q \sqcap P$
$(P \sqcap Q) \sqcap R = P \sqcap (Q \sqcap R)$
$P \sqcap true = true$
$P \sqcap false = P$
$P \sqcap Q \sqsubseteq P$
by (*simp-all add*: *lattice-class.inf-sup-aci true-upred-def false-upred-def*)

Non-deterministic choice is idempotent, meaning that a choice between P and P is no choice. It is also commutative and associative. If we make a choice between P and *true* then the erroneous behaviour signified by the latter is always chosen. Thus our operator is a so-called "demonic choice" since the worst possibility is always picked. Similarly, if a choice is made between P and a miracle (*false*) then P is always chosen in order to avoid miracles. Finally, the choice between P and Q can always be refined by removing one of the possibilities.

Since predicates form a complete lattice, then by the Knaster-Tarski theorem the set of fixed points of a monotone function F is also a complete lattice. In particular, this complete lattice has a weakest and strongest element which can be calculated using the notations *gfp F* and *lfp F*, respectively. Such fixed point constructions are of particular use for expressing recursive and iterative constructions. Isabelle/HOL provides a number of laws for reasoning about fixed points, a few of which are detailed below.

theorem *mu-id*: $(\mu\ X \cdot X) = true$
by (*simp add*: *mu-id*)

theorem *nu-id*: $(\nu\ X \cdot X) = false$
by (*simp add*: *nu-id*)

theorem *mu-unfold*: $mono\ F \Longrightarrow (\mu\ X \cdot F(X)) = F(\mu\ X \cdot F(X))$
by (*simp add*: *def-gfp-unfold*)

theorem *nu-unfold*: *mono* $F \Longrightarrow (\nu\ X \cdot F(X)) = F(\nu\ X \cdot F(X))$
by (*simp add*: *def-lfp-unfold*)

Perhaps of most interest are the unfold laws, also known as the "copy rule", that allows the function body F of the fixed point equation to be expanded once. These state that, provided that the body of the fixed point is a monotone function, then the body can be copied to the outside. These can be used to prove equivalent laws for operators like the while loop.

1.6 Laws of programming

Although we have some primitive tactics for proving conjectures in the predicate and relational calculi, in order to build verification tools for programs we need a set of algebraic "laws of programming" [6] that describe important theoretical properties of the operators. Isabelle/UTP contains several hundred examples of such laws, and we here outline a few of them.

theorem *seq-assoc*: $(P\ ;;\ Q)\ ;;\ R = P\ ;;\ (Q\ ;;\ R)$
by (*rel-auto*)

theorem *seq-unit*:
$P\ ;;\ II = P$
$II\ ;;\ P = P$
by (*rel-auto*)+

theorem *seq-zero*:
$P\ ;;\ false = false$
$false\ ;;\ P = false$
by (*rel-auto*)+

Sequential composition is associative, has the operator II as its left and right unit, and *false* as its left and right zeros. The II operator is a form of assignment which simply identifies all the variables between the before and after state, as the following example demonstrates.

lemma $x := \&x = II$
by (*rel-auto*)

In the context of relations, *false* denotes the empty relation, and is usually used to represent a miraculous program. This is intuition of it being a left and right zero: if a miracle occurred then the whole of the program collapses. The conditional $P \lhd b \rhd Q$ also has a number of algebraic laws that we can prove.

theorem *cond-true*: $P \lhd true \rhd Q = P$
by (*rel-auto*)

theorem *cond-false*: $P \lhd false \rhd Q = Q$
by (*rel-auto*)

theorem *cond-commute*: $(P \lhd \neg\ b \rhd Q) = (Q \lhd b \rhd P)$
by (*rel-auto*)

theorem *cond-shadow*: $(P \lhd b \rhd Q) \lhd b \rhd R = P \lhd b \rhd R$
by (*rel-auto*)

A conditional with *true* or *false* as its condition presents no choice. A conditional can also be commuted by negating the condition. Finally, a conditional within a conditional over the same

condition, b, presents and unreachable branch. Thus the inner branch can be pruned away. We next prove some useful laws about assignment:

theorem *assign-commute*:
assumes $x \bowtie y$ $y \sharp e$ $x \sharp f$
shows $x := e;; y := f = y := f ;; x := e$
using *assms* **by** (*rel-auto*)

theorem *assign-twice*:
shows $x := \ll e \gg;; x := \ll f \gg = x := \ll f \gg$
by (*rel-auto*)

theorem *assign-null*:
assumes $x \bowtie y$
shows $(x, y := e, \&y) = x := e$
using *assms* **by** (*rel-auto*)

Assignments can commute provided that the two variables are independent, and the expressions being assigned do not depend on the variable of the other assignment. A sequence of assignments to the same variable is equal to the second assignment, provided that the two expressions are both literals, i.e. $\ll e \gg$. Finally, in a multiple assignment, if one of the variables is assigned to itself then this can be hidden, provided the two variables are independent.

Since alphabetised relations form a complete lattice, we can denote iterative constructions like the while loop which is defined as $while_\bot\ b\ do\ P\ od = (\mu\ X \cdot (P ;; X) \lhd b \rhd_r II)$. We can then prove some common laws about iteration.

theorem *while-false*: $while_\bot\ false\ do\ P\ od = II$
by (*simp add*: *while-bot-false*)

theorem *while-unfold*: $while_\bot\ b\ do\ P\ od = (P ;; while_\bot\ b\ do\ P\ od) \lhd b \rhd_r II$
using *while-bot-unfold* **by** *blast*

As we have seen, the predicate *true* represents the erroneous program. For loops, we have it that a non-terminating program equates to *true*, as the following example demonstrates.

lemma $while_\bot\ true\ do\ x := \&x + 1\ od = true$
by (*simp add*: *assigns-r-feasible while-infinite*)

A program should not be able to recover from non-termination, of course, and therefore it ought to be the case that *true* is a left zero for sequential composition: $true ;; P = true$. However this is not the case as the following examples illustrate:

lemma $true ;; P = true$
apply (*rel-simp*)
nitpick — Counterexample found
oops

lemma $(true ;; x,y,z := \ll c_1 \gg,\ll c_2 \gg,\ll c_3 \gg) = ((x,y,z := \ll c_1 \gg,\ll c_2 \gg,\ll c_3 \gg) :: myst\ hrel)$
by (*rel-auto*)

The latter gives an example of a relation for which *true* is actually a left unit rather than a left zero. The assignment $\langle[\&x \mapsto_s \ll c_1 \gg, \&y \mapsto_s \ll c_2 \gg, \&z \mapsto_s \ll c_3 \gg]\rangle_a$ does not depend on any before variables, and thus it is insensitive to a non-terminating program preceding it. Thus we can see that the theory of relations alone is insufficient to handle non-termination.

1.7 Designs

Though we now have a theory of UTP relations with which can form simple programs, as we have seen this theory experiences some problems. A UTP design, $P \vdash_r Q$, is a relational specification in terms of assumption P and commitment Q. Such a construction states that, if P holds and the program is allowed to execute, then the program will terminate and satisfy its commitment Q. If P is not satisfied then the program will abort yielding the predicate *true*. For example the design $(\$x \neq_u 0) \vdash_r y := \&y \ div \ \&x$ represents a program which, assuming that $x \neq 0$ assigns y divided by x to y.

lemma *dex1*: $true \vdash_r x,y := 2,6 \ ;; \ (\$x \neq_u 0) \vdash_r y := \&y \ div \ \&x = true \vdash_r x,y := 2,3$
by (*rel-auto*, *fastforce+*)

lemma *dex2*: $true \vdash_r x,y := 0,4 \ ;; \ (\$x \neq_u 0) \vdash_r y := \&y \ div \ \&x = true$
by (*rel-blast*)

The first example shows the result of pre-composing this design with another design that has a *true* assumption, and assigns 2 and 6 to x and y respectively. Since x satisfies $x \neq 0$, then the design executes and changes y to 3. In the second example 0 is assigned to x, which leads to the design aborting. Unlike with relations, designs do have *true* as a left zero:

theorem *design-left-zero*: $true \ ;; \ (P \vdash_r Q) = true$
by (*simp add*: *H1-left-zero H1-rdesign Healthy-def*)

Thus designs allow us to properly handle programmer error, such as non-termination.

The design turnstile is defined using two observational variables $ok, ok' : \mathbb{B}$, which are used to represent whether a program has been started (ok) and whether it has terminated (ok'). Specifically, a design $P \vdash Q$ is defined as $(ok \wedge P) \Rightarrow (ok' \wedge Q)$. This means that if the program was started (ok) and satisfied its assumption (P), then it will terminate (ok') and satisfy its commitment (Q). For more on the theory of designs please see the associated tutorial [2].

1.8 Reactive Designs

A reactive design, $\mathbf{R}_s \ (P \vdash Q)$, is a specialised form of design which is reactive in nature. Whereas designs represents programs that start and terminate, reactive designs also have intermediate "waiting" states. In such a state the reactive design is waiting for something external to occur before it can continue, such as receiving a message or waiting for sufficient time to pass as measured by a clock. When waiting, a reactive design has not terminated, but neither is it an infinite loop or some other error state.

Reactive designs have two additional pair of observational variables:

- $wait, wait' : \mathbb{B}$ – denote whether the predecessor is in a waiting state, and whether the current program is a waiting state;
- $tr, tr' : \mathcal{T}$ – denotes the interaction history using a suitable trace type $\mathcal{T}$.

For more details on reactive designs please see the associated tutorial [2].

References

[1] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In *FroCoS*, volume 6989 of *LNCS*, pages 12–27. Springer, 2011.

[2] A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in unifying theories of programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.

[3] A. Feliachi, M.-C. Gaudel, and B. Wolff. Unifying theories in Isabelle/HOL. In *UTP 2010*, volume 6445 of *LNCS*, pages 188–206. Springer, 2010.

[4] J. Foster. *Bidirectional programming languages.* PhD thesis, University of Pennsylvania, 2009.

[5] S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In *Proc. 13th Intl. Conf. on Theoretical Aspects of Computing (ICTAC)*, volume 9965 of *LNCS*. Springer, 2016.

[6] T. Hoare, I. Hayes, J. He, C. Morgan, A. Roscoe, J. Sanders, I. Sørensen, J. Spivey, and B. Sufrin. The laws of programming. *Communications of the ACM*, 30(8):672–687, August 1987.

[7] T. Hoare and J. He. *Unifying Theories of Programming.* Prentice-Hall, 1998.

[8] C. Morgan. *Programming from Specifications.* Prentice Hall, 1990.

[9] F. Zeyda, S. Foster, and L. Freitas. An axiomatic value model for Isabelle/UTP. In *Proc. 6th Intl. Symp. on Unifying Theories of Programming*, volume 10134 of *LNCS*, pages 155–175. Springer, 2016.