Isabelle/UTP: A mechanised theory engineering framework

Simon Foster, Frank Zeyda, and Jim Woodcock

University of York
{simon.foster,frank.zeyda,jim.woodcock}@york.ac.uk

Abstract. We introduce Isabelle/UTP, a novel mechanisation of Hoare and He's Unifying Theories of Programming (UTP) in Isabelle/HOL. UTP is a framework for the study, formalisation, and unification of formal semantics. Our contributions are, firstly, a deep semantic model of UTP's alphabetised predicates, supporting meta-logical reasoning that is parametric in the underlying notions of values and types. Secondly, integration of host-logic type checking that subsumes the need for typing proof obligations in the object-language. Thirdly, proof tactics that transfer results from well-supported mathematical structures in Isabelle to proofs about UTP theories. Additionally, our work provides novel insights towards reconciliation of shallow and deep language embeddings.

1 Introduction

Creation of a tool-chain for formal software development requires that the various components have a unified view of the problem domain. This issue is often tackled with *formal semantics*, so that the underlying program artifacts have unambiguous mathematical meanings. However, individual tools are often based on different semantic models: a simulator might implement an operational semantics, whilst a program verifier might implement an axiomatic Hoare calculus. Therefore to coordinate the tools in an integrated tool-chain, it is important to unify their different underlying semantic models. Otherwise the "formalisation gaps" may lead to incoherent analysis, and therefore ineffective verification.

A solution lies in the *Unifying Theories of Programming* [15] (UTP), a mathematical framework for the formalisation, study, and unification of such semantic models. UTP provides a layered approach to semantics that is illustrated in Figure 1, with different tools on top. If we desire an integrated tool-chain we need to (1) formalise semantic models for the tools, and (2) unify those models through an underlying *denotational semantics* that acts as the "gold standard" for comparing other semantic models. The order in which these elements are engineered can vary, but the end result is the same: an end-to-end verified tool-chain.

UTP further considers that denotational models can themselves be decomposed into constituent *UTP theories*. A UTP theory corresponds to a notable computational idea which can be isolated and studied independently. Though a large number of languages and notations exist, the paradigms that underlie them

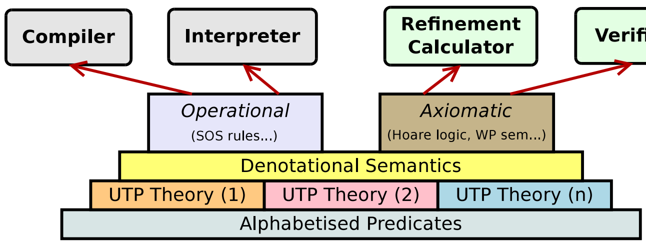

Fig. 1. UTP semantic stack

is much smaller, and therefore theoretical decomposition allows formal links to be established between similar formalisms. These links can either be built by common theoretical factors or alternatively by *Galois connections*, which formalise the best approximation of objects in one theory by objects in another. UTP therefore also enables a tool-chain that is semantically heterogeneous; consisting of a variety of notations that are formally linked [10].

UTP has been applied to theories from a wide area of computer science, including process calculi, object orientation, and real-time systems. It provides the basis for the *Circus* [25] specification language, which unifies Z, CSP, and refinement calculi, and has been used to formalise the semantics of a number of languages, such as Safety-Critical Java [6], Handel-C [21], and SysML [26], a multi-paradigm modelling notation.

In order to verify correctness of semantic models, we need mechanical support for formalising UTP theories, composing them to produce denotational models, and providing provably corresponding semantic bases. In a theorem prover like Isabelle [18], we can go even further and construct proof tactics and procedures for proving properties of theory objects in a particular semantic interface, such as a Hoare logic solver. This then means that we have an unbroken chain from proof of program correctness to justification in terms of high-level properties (like termination and feasibility) in the underlying denotational models and theories. Though mechanisations of UTP exist, they focus on either theory engineering [19,27,4,5] or program verification [8,9], but not both.

We have therefore created *Isabelle/UTP*, a novel mechanisation of UTP in Isabelle, which we believe can be applied to both tasks. Isabelle/UTP is a framework that allows the formation of theories, semi-automated proof of their properties, and theory combination to provide semantic models. It provides a deep model of alphabetised predicates that is also tightly integrated into the Isabelle type system and supported by high-level proof tactics. Isabelle is an ideal base due to its definitional nature that allows a natural and scalable representation of the different abstraction layers of Figure 1. For example when reasoning about a program using axiomatic semantics, we need not consider details of the underlying denotational model, but do retain confidence in its correspondence.

Our mechanisation is inspired by the prior embeddings ProofPower-Z UTP [19] and Isabelle/Circus [8,9]. We believe that Isabelle/UTP combines their advantages in having both the expressivity of a deep value and predicate model, but also taking advantage of host logic facilities, such as type inference and proof automation, typically only available in a shallow model. This combination is our main contribution, and substantiates the claim that Isabelle/UTP can be applied to both the tasks of theory engineering *and* program verification.

For theory engineering, we support proofs at a high-level of abstraction, aided by Isabelle's Isar proof scripting language. This is also important to closing the formalisation gaps between pen-and-paper proofs and their mechanisation. For example, the proof from the theory of designs, relating H1-H2 predicates and the design turnstile, can be written at a similar level of abstraction to book proofs:

```
theorem H1-H2-is-DesignD:
  assumes P is H1 and P is H2
  shows ‘P‘ = ‘$\neg P^f \vdash P^t$‘
proof –
  from assms have ‘P‘ = ‘$\$ok \Rightarrow H2(P)$‘
    by (utp-pred-tac)
  also have ... = ‘$\$ok \Rightarrow (P^f \vee (P^t \wedge \$ok'))$‘
    by (metis H2-split assms)
  also have ... = ‘$\$ok \wedge \neg P^f \Rightarrow \$ok' \wedge P^t$‘
    by (utp-pred-auto-tac)
  finally show ?thesis by (metis DesignD-def)
qed
```

It proceeds by (1) expanding H1 and H2 on P using our UTP predicate tactic, (2) applying the *H2-split* law to divide P into the two cases, (3) using the predicate tactic to rearrange, and finally (4) applying the definition of the design turnstile.

In the remainder we present our contributions. Section 2 gives a brief overview of the UTP. Section 3 gives preliminaries about Isabelle. Section 4 examines the existing UTP implementations and how Isabelle/UTP relates. Section 5 describes our deep predicate model. Section 6 describes the polymorphic expression model with host logic typing. Section 7 outlines our approach to automated proof tactics. Section 8 concludes the discussion and highlights future work.

2 Unifying Theories of Programming

UTP is a semantic framework facilitating study of the commonalities and differences between computational theories and paradigms. UTP is based on *programs as predicates*: every program can be written as a predicative binary relation between input variables, which are unprimed (x), and output variables, which are primed (x'). Predicates in UTP are also *alphabetised*, meaning they explicitly carry the set of variables to which they can refer. A key component of UTP is the *UTP theory*, which describes a particular aspect of a computational paradigm. For example, the *theory of designs* characterises specifications $P \vdash Q$, where P is an assumption and Q a commitment. The *theory of CSP* characterises processes from the CSP process calculus. A UTP theory consists of three parts:

- an *alphabet*: set of variables with which to observe theory objects;
- a *signature*: constructors for objects of the theory;
- *healthiness conditions*: functions that define theory membership.

The *alphabet* consists of variables which are used to encode structure in elements of the theory. For example, tr is a sequence of events representing the trace of interactions, and ok is a Boolean variable used to encode termination conditions. The *signature* consists of constructors that use these variables to encode syntax for the theory. In CSP, the signature consists of the usual CSP operators, like $a \to P$, $P \Box Q$, and $CHAOS$. For example, we could define $CHAOS \triangleq tr \leq tr'$, that is the predicate in which any observation is possible but the trace must only increase monotonically. Finally, the *healthiness conditions* are idempotent functions of which all the theory elements are fixed-points. In the theory of reactive processes, the condition $R1(P) \triangleq P \wedge tr \leq tr'$ states that in all observations the trace must get longer. Predicates which do not obey this constraint are outside the theory. Conversely, it trivially follows that $R1(CHAOS) = CHAOS$.

UTP also provides ways of combining and linking theories. The UTP book [15] shows how different process calculi can be constructed, beginning from a common base (reactive processes) and adding specific healthiness conditions for calculi like ACP and CSP. Moreover, the *Circus* specification language [25] is a combination of Z [22], CSP [14], and a refinement calculus. Aside from combination, theories can also be related using *Galois connections*, that formulate best approximations of UTP theory elements in a different theory. For example, there exists a Galois connection between the theory of CSP and the theory of designs, such that every CSP process can be written as a "reactive design" [20]. This therefore allows the import of results from the theory of designs into the theory of CSP, which provides a theory engineer with additional tools.

In order to mechanise theories we need a UTP implementation with sufficient expressivity to represent alphabets, theories, and theory links, all of which should be first-class entities in the language. This in turn means we need a flexible predicate model, whilst providing high-level proof facilities. Moreover the UTP implementation should be, to a significant extent, *language agnostic* inasmuch as issues concerning concrete implementation should be separated from high-level abstract concepts. This motivates a *modular* embedding supporting this separation of concerns. We consider these motivations in more detail in Section 4.

3 Isabelle/HOL

Isabelle/HOL [18] is a proof assistant for Higher-Order Logic (HOL). HOL consists of an ML-style functional programming language and a proof system to assert and prove properties of defined constructs. Functional programming concepts include recursive functions, algebraic datatypes, and records. HOL is also polymorphically typed, with type-parametric constructs defined using *type-classes*. For instance, we can declare a type-class for semigroups:

```
class semigroup =
  fixes add :: "'a ⇒ 'a ⇒ 'a" (infixr "⊕" 65)
  assumes add_assoc: "x ⊕ (y ⊕ z) = (x ⊕ y) ⊕ z"
```

This introduces a polymorphic infix operator "⊕", with precedence 65, over the type parameter `'a`, and requires that ⊕ be associative. Type classes can be instantiated for particular types, in which case the operators of the signature must be defined for that type, and the associated assumptions (here add_assoc) proved. Moreover type classes support a form of *inheritance*, such that they can be declared to inherit the signature and properties from one or more existing type classes. This, for instance, allows the specification of algebraic hierarchies. For example, we can create the monoid class by extending semigroup:

```
class monoid = semigroup +
  fixes one :: "'a" ("1")
  assumes l_one: "1 ⊕ x = x" and r_one: "x ⊕ 1 = x"
```

Type classes create a *local* theory context in which the assumptions are available as axioms to be used in proofs. These are not *global* axioms, and so even though it is possible to introduce inconsistent type class axioms, this does not have any effect on the global theory context. It simply means that the class cannot be instantiated; conversely if a class *can* be instantiated with a type, it means the axioms are consistent with respect to the global context.

Types in HOL can be created in several ways, such as through the algebraic datatype package. However, the core command for type definition is **typedef** which creates a new type based on a subset of values of an existing type. For instance, we may create the type of even numbers as a subset of the naturals:

```
typedef even = "{n :: nat. n mod 2 = 0}"
  by (rule_tac x="2" in exI, simp)
```

HOL types must contain at least one value and so the **typedef** requires a nonemptiness proof. We satisfy this by introducing 2 as a witness, and applying the simplifier to show that "$2 \,\mathsf{mod}\, 2 = 0$". New types come equipped with an abstraction and a representation function mapping between the old and new type, in this case Abs_even :: nat ⇒ even and Rep_even :: even ⇒ nat; they are mutual inverses on the characteristic set of even. Functions on the original type can also be lifted to the derived type using the lifting package [16], provided the function is closed over the characteristic set. For example, we can define even number addition by lifting natural addition, as the latter is closed over the even numbers. This must be proved as part of the definition (here we use auto):

```
lift_definition plus_even :: "even ⇒ even ⇒ even" is "plus" by auto
```

Proof in Isabelle is performed using tactics, which act on the proof state, transforming a matching goal into subgoals. This process is repeated until all subgoals are discharged. A proof can be entered as a sequence of commands (apply-style) or using the ISAR structured proof language, which supports readable proofs. Proofs are correct by construction with respect to a small core of axioms as part of the *LCF architecture* that ensures relative soundness of the system. Though a consistency proof for the HOL axioms is impossible (*à la* Gödel), they have been given a semantics in ZFC set theory [17] and therefore HOL is as consistent as the "foundation of mathematics". User theories should ideally be *definitional*: constructs are created in terms of existing constructs, rather than by axiomatisation that is discouraged and in fact seldom needed. This ensures that no inconsistency can be introduced by user-defined theories.

Along with elementary deduction tactics, like backward chaining, Isabelle has a large number of automated proof tactics. This includes the equational simplifier simp, the classical reasoner blast, and the combination tactic auto. Tactics can be augmented with additional rules by placing them in appropriate theorem sets. For instance, the set simp contains simplification laws and intro contains introduction laws. Isabelle also includes a number of high-level proof tools [1] such as sledgehammer, a certified integration of third-party automated theorem provers, and nitpick, a counterexample generator. We believe this combination sets Isabelle apart as an ideal platform on which to mechanise UTP.

4 Related work: five mechanisation criteria

In this section we consider existing mechanisations of the UTP and motivate features for Isabelle/UTP. Several prior UTP mechanisations exist, including an embedding in ProofPower-Z [19,27], a shallower embedding in Isabelle called Isabelle/Circus [8,9], and the specialised UTP prover $U{\cdot}(TP)^2$ [4,5]. To aid comparison we consider five desirable mechanisation criteria:

1. *consistency* – it should not be possible to derive contradictions;
2. *expressivity* – laws can be formulated with sufficient granularity;
3. *automated proof* – reasoning can be performed at a suitably high-level;
4. *well-formedness by construction* – so proof effort is minimised;
5. *modularity* – to enable language-independent reasoning.

Though by no means exhaustive, this list allow us to discriminate between the current mechanisations and motivate the features for Isabelle/UTP that will allow both theory engineering and program verification. The summarised comparison is shown in Table 1 which we now discuss in detail.

Consistency. Mechanising the semantics and proof system of a language is typically performed through a *semantic embedding* [2]: the constructs of the "client-logic" are assigned denotations in a semantic model defined in a suitably expressive host-logic. This has the advantage that consistency of the client-logic can be argued from the consistency of the host-logic, such as HOL.

It follows, therefore, that Isabelle/Circus [9] is inherently consistent, as it is mechanised within the LCF architecture and does not rely on additional axioms. ProofPower-Z UTP [19] is also consistent, in theory, as ProofPower-Z follows LCF. However it is also possible to disable consistency checks for Z schemas, which can introduce problems. $U{\cdot}(TP)^2$ [4] cannot guarantee consistency as it is a fresh implementation and, as such, soundness of its axioms has not been established.

	ProofPower-Z	Isabelle/Circus	$U{\cdot}(TP)^2$	Isabelle/UTP
(1) Consistency	✓/?	✓	?	✓
(2) Expressivity	✓	✗	✓	✓
(3) Proof Auto	✗	✓	✗	✓
(4) Well-formedness	✗	✓	✓	✓
(5) Modularity	✗	?	✓	✓

Table 1. UTP mechanisations contrasted

Therefore, like Isabelle/Circus, we embed our mechanisation in Isabelle/HOL and take a purely definitional approach. Moreover our model will be formed using Isabelle **typedef** and lifting, thus ensuring non-vacuity.

Expressivity. *U·(TP)*^{2} is a UTP implementation purpose-built from first principles. It therefore allows flexible description of UTP laws and theories with a suitable level of expressivity, and sets a good benchmark for other implementations which are predominantly semantic embeddings. In a semantic embedding, expressivity is principally determined by depth. Deeper embeddings introduce concepts such as variables and typing by custom definitions, whilst shallower embeddings reuse equivalent concepts in the host-logic. Hence a deeper embedding provides greater potential for expressivity, though at the cost of extra work. UTP is based on the alphabetised predicate calculus, and so its mechanisation requires a semantic model in which the predicate operators can be denoted. Moreover to allow generic theory composition, it must be possible to compose predicates with different alphabets.

A common predicate model used since the earliest embeddings of Z in HOL is the *binding set* [3], where a binding is a mapping from variables to values. Each predicate is modelled by the set of bindings that satisfy it, e.g. $[\![x > 1]\!] = \{x \mapsto 2, x \mapsto 3, \cdots\}$. This approach is used by both ProofPower-Z UTP and Isabelle/Circus. The major difference between these embeddings is how the bindings themselves are represented. This difference determines both the potential for automation and expressivity of the resulting predicate model.

Isabelle/Circus, a shallower model, encodes bindings as HOL record types with each field corresponding to a variable assignment. This limits expressivity: laws that rely on meta-logical properties, such as variable freshness or comparison, cannot generally be expressed. Moreover since variables and thus alphabets are not explicitly modelled, alphabets cannot be arbitrarily decomposed. This limits the extent to which UTP theories with different alphabets can be composed.

In contrast, ProofPower-Z UTP explicitly defines types for variables ($NAME$) and values (VAL): a binding then is a function $BINDING \triangleq NAME \to VAL$ and a predicate is a binding set $PRED \triangleq \mathbb{P}\, BINDING$. This allows greater expressivity than Isabelle/Circus since we can compare variables, and similarly encode alphabets as variable sets. We therefore choose a deep model for Isabelle/UTP so we can achieve a faithful UTP encoding; see Section 5.

Proof Automation. To allow effective reasoning about UTP theories, laws of programming, and eventually programs themselves, it is necessary to provide high-level proof tactics. *U·(TP)*^{2}, as a fresh implementation, cannot take advantage of proof automation as provided by more mature proof assistants. The ProofPower-Z UTP model, as a deep embedding, also does not have well developed proof tactics for automated reasoning. Moreover automation at the level of ProofPower-HOL itself is weak in areas such as automated deductive reasoning and simplification. This adds to the proof burden making reasoning a challenge, particularly for non-experts.

Isabelle, by contrast, has powerful tactics for reasoning about different theories, including sets, relations, and lattices. Indeed, Isabelle/Circus makes much use of these, and simple properties like associativity can be discharged easily by auto. Moreover, the shallowness of this model means that other proof tools, such as sledgehammer and nitpick, are directly available. This means the full weight of Isabelle's proof facilities can be brought to bear, making Isabelle/Circus ideal for verification. However, a deeper embedding needs more machinery to enable access to these. Hence for Isabelle/UTP we engineer tactics that formalise links between UTP theories and established HOL theories, such as relations. This allows transfer of results from HOL into UTP, thus maximising potential for proof automation whilst retaining the expressivity of a deep model; see Section 7.

Well-formedness. While ProofPower-Z UTP is expressive, it exhibits problems with well-formedness and typing. Bindings, and thus predicates, must be proved well-formed through closure conditions. Also, typing in UTP predicates is orthogonal to the ProofPower-Z type system. Therefore, it is possible to form expressions that would normally be rejected by the type system, such as "1+**true**". We are therefore burdened with discharging closure and typing proof obligations.

In contrast, Isabelle/Circus provides direct access to the HOL type-system in its binding model. Since a binding is a HOL type, inconsistently typed predicates are rejected: there is no need to separately establish well-formedness. This has several advantages, most notably that program verification is simplified by harnessing host-logic type checking. However the type system can also be a barrier to composition of predicates with different alphabets and so should only be enforced when needed. Therefore we retain the intrinsic "deep nature" of Isabelle/UTP and add an additional "shallow layer" on top of the core predicate model that imposes host logic typing. This allows us to reason at different levels of type abstraction depending on the activity. We enable proof of correspondence between types within the underlying model type system and types in the HOL type system. We then engineer a polymorphic value model that uses the HOL type system to expose this information; see Section 6.

Modularity. UTP does not specify a particular value or type system, and hence it should be possible to reason about UTP predicates and theories independently of the underlying value model. This is important so that laws can be proved generically and then imported into different programming languages. In this way a programming language can be given a semantics as a composition of UTP theories. ProofPower-Z UTP fixes a particular notion of value and is therefore not easily adaptable to different languages. In contrast Isabelle/UTP formalises a *parametric* value model that allows the user to precisely specify a type system and definedness predicate for a particular language, thus enabling language-independent reasoning. Moreover we use the type class system to impose constraints on the value model in polymorphic predicate constructions. Thus, UTP theories can demand particular types be available in a model for it to be instantiable in that theory.

In the remainder we will describe in more detail how each of these five criteria is met in Isabelle/UTP. We begin in the next section with the predicate model.

5 Predicate Model

In this section, we expound the predicate model for Isabelle/UTP. The core model is based on [19,27], but uses **typedef** and function lifting [16] to ensure well-formedness of predicates by construction. The contributions are:

1. A polymorphic value model enabling modular reasoning about model types.
2. A model of bindings that is well-formed by construction.
3. A layered predicate model consisting of:
   – core predicates and expressions;
   – alphabetised predicates (alphabet + core predicate).
4. The standard imperative programming predicate transformers.

Our contributions satisfy the criteria of expressivity **(2)**, modularity **(5)** and partially satisfy well-formedness **(4)**. We have also implemented an extensible parser for UTP predicates using Isabelle *syntax translations*. This means that UTP predicates can be presented in Isabelle as written in this paper using a back-tick notation, e.g. '$P \wedge Q \sqsubseteq R$'. We elide these details in the remainder, but note that this feature makes Isabelle/UTP particularly accessible and helps to further close the formalisation gap by avoiding an unfamiliar syntax.

Different from [19] our layered predicate model splits predicate types into "core" and "alphabetised" variants. Alphabets can complicate proof, since equality proofs about alphabetised predicates must ensure the alphabet remains constant at each step. The layered model gives the ability to ignore the alphabet in intermediate proof steps, as proofs about alphabetised predicates can be decomposed into an alphabet proof and core predicate proof. This gives flexibility in proof, contributing toward satisfaction of criterion **(3)**. It also means that proof automation need only act on core predicates; this will be seen in Section 7.

The remainder of this section is heavy in technical content; a casual reader may wish to skip straight to Section 6.

5.1 Parametric Value Model

The *value model* provides the basis for the predicate model by constraining the well-typed values we can construct and assign to variables. Various languages with differing notions of value and type exist, and so since we desire a general framework, we opt for a customisable value model. This allows us to separate general UTP predicate proofs from those that apply to a particular language, providing modularity **(5)**. Moreover, an embedded type system enables expression of advanced types, like dependent products, that are useful in constructs like CSP actions which compose a typed channel with an appropriate value.

We implement value models by using type classes to polymorphically introduce a constraint hierarchy on the value model, which can be refined as required for a particular UTP theory, as illustrated in Figure 2. At the base of the hierarchy is `VALUE`, which specifies a basic notion of value model. Below this are sorts also axiomatising the existence of different values and types in the model, such as Booleans and sets. Such classes should be non-vacuous to ensure local

$$\begin{tikzcd}
 & \text{VALUE} & & \\
\text{BOOL\_SORT} \arrow[ur] & \text{EVENT\_SORT} \arrow[u] & \text{LIST\_SORT} \arrow[ul] & \text{SET\_SORT} \arrow[ull] \\
\text{DESIGN\_SORT} \arrow[u] & & & \text{REACTIVE\_SORT} \arrow[ulll] \arrow[ull] \arrow[ul] \arrow[u]
\end{tikzcd}$$

Fig. 2. Value model class hierarchy

consistency. Finally the leaf nodes define classes for particular UTP theories, including designs which needs a Boolean for *ok*, and reactive processes, which additionally requires lists and sets of events to represent traces and refuals, respectively. Hence a UTP theory constrains the value model only as it needs. Our predicate model is therefore parametric over a value model type-parameter `'a`.

Minimally, a value model must introduce a value type `'a`, a typing relation for `'a`, and a definedness predicate $\mathcal{D}$ that determines whether a value is defined. Formalising definedness at this level allows generic reasoning about variables containing undefined values. We have used this, for example, in mechanising a value model for CML with its unique notion of undefinedness [26]. The base of our parametric value model is therefore specified via the following type classes:

```
class DEFINED =
  fixes Defined :: "'a ⇒ bool" ("𝒟")

class VALUE = DEFINED +
  fixes utype_rel :: "'a ⇒ nat ⇒ bool" (infix ":_u" 50)
  assumes utype_nonempty: "∃ t v. v :_u t ∧ 𝒟 v"
```

The `VALUE` class introduces primitives to represent definedness ($\mathcal{D}$) and the typing relation for values ($:_u$). This provides our UTP predicate model with a set of fundamental axioms on which to build, manifested through polymorphism. As type-classes permit only a single type parameter for reasons of decidability, we use a natural number to represent types; we assume types are purely syntactic and thus countable. Additionally we require, by assumption `utype_nonempty`, that the type relation exhibits at least one well-typed and defined value. We then create a HOL type for UTP types, called `'a utype`, which is parametric over the value model and isomorphic to the set of numbered types for which there is at least one defined value. We also create the derived typing relation "_ : _ :: `'a ⇒ 'a utype ⇒ bool`" that only accepts feasible types. Additional sort classes like `BOOL_SORT` and `LIST_SORT` are then declared by extending `VALUE` with injections for their respective sort types. We can then, for instance, use `BOOL_SORT` to constrain constructs requiring the presence of Booleans, and use the injections to produce correctly typed model values. A concrete model will usually consist of datatypes for the value and type spaces, and functions for the typing relation and definedness predicate, which instantiate the value model.

As in [19], we model variables explicitly, and with [27] also encode type data so bindings can impose well-typedness. We encode type `NAME`, consisting of a string label and decorations, and then type `'a uvar` as a record with fields `name :: NAME`, `vtype :: 'a utype` and `aux :: bool`, that denotes whether the variable is an auxiliary or program variable. Auxiliary variables are used to encode UTP theory variables and we require they be defined in every binding to avoid unnecessarily reasoning about undefinedness. In contrast, program variables can have undefined values. We write D_n, for $n \in \mathbb{N}$, for the set of variables with n dashes. We also introduce the syntax $v \rhd x$, meaning that value v is compatible with variable x: it is correctly typed and if x is auxiliary then v is defined.

5.2 Well-formed Bindings

Bindings are represented using a **typedef** as the set of functions from variables to values, such that each variable is associated with a value of the correct type. In contrast to [19] the HOL type-system ensures well-formednes by construction.

```
typedef 'a binding = "{b :: 'a uvar ⇒ 'a . ∀ x. b x ▷ x}"
```

Effectively a binding is a dependent function: the type of value each variable points to depends on that variable. Our assumption that there exists at least one value per type (`utype_nonempty`) is crucial in creating the binding type, as this in turn ensures that at least one binding exists. We also define a number of functions necessary for the definition of the predicate operators. Binding override, written $b_1 \oplus_b b_2$ **on** vs, replaces the values of variable set vs in binding b_1 with those in b_2. It is defined by lifting the function override operator in HOL that is closed under well-formed bindings since we are substituting well-typed assignments:

```
lift_definition binding_override_on ::
  "'a binding ⇒ 'a binding ⇒ 'a uvar set ⇒ 'a binding"
  is "override_on" by (auto simp add:override_on_def)
```

Binding update, $b(x :=_b v)$, replaces the value of x in b with v, under the assumption that $v \rhd x$. If v is not well-typed, an arbitrary well-typed value is inserted. Operators written in terms of binding override are therefore subject to typing proof obligations to ensure injectivity. Proofs about large predicates could require the discharge of many such obligations, an onerous task, but this problem will be dealt with in Section 6. We adopt the notation $\langle b \rangle_b(x)$ to refer to the value assigned to x in b. This function derives well-typedness: $\langle b \rangle_b(x) \rhd x$.

We also formalise binding permutations, which are used to implement alpha conversion. Permutations, borrowed from Nominal Logic [24], are type-preserving total bijections on variables. We define permutation composition, written $\pi \bullet b$, which precomposes the binding function of b with the permutation π, yielding a binding where each variable takes on the value of the variable it was renamed from. Due to the type-preserving nature of permutations, which we also encode using a **typedef**, such a composition produces a well-formed binding.

5.3 Core Predicates

Next, we introduce a HOL type to represent core predicates:

```
typedef 'a upred = "UNIV :: 'a binding set set" ..
```

Our predicate type is isomorphic to the universe set of well-formed bindings. A predicate is then simply the set of bindings that satisfy it. Unlike [9] we

$[\![\mathsf{true}]\!]_p = \mathcal{UNIV}$	$[\![P \sqsubseteq Q]\!]_p = [\![[Q \Rightarrow P]]\!]_p$
$[\![\neg P]\!]_p = -[\![P]\!]_p$	$[\![e = f]\!]_p = \{b \mid [\![e]\!]_e(b) = [\![f]\!]_e(b)\}$
$[\![P \vee Q]\!]_p = [\![P]\!]_p \cup [\![Q]\!]_p$	$[\![P[e/x]]\!]_p = \{b \mid b(x :=_b [\![e]\!]_e(b)) \in [\![P]\!]_p\}$
$[\![P \Rightarrow Q]\!]_p = [\![\neg P \vee Q]\!]_p$	$[\![\pi \bullet P]\!]_p = \{\pi \bullet b \mid b \in [\![P]\!]_p\}$
$[\![\exists vs.\ P]\!]_p = \{b \oplus_b b'\ \mathbf{on}\ vs \mid b \in [\![P]\!]_p\}$	$[\![\mathsf{LitE}\ v]\!]_e = \lambda b.v$
$[\![\forall vs.\ P]\!]_p = [\![\neg\exists vs.\neg P]\!]_p$	$[\![\mathsf{Op1E}\ f\ x]\!]_e = \lambda b.f([\![x]\!]_e(b))$
$[\![[P]]\!]_p = [\![\forall \mathcal{UNIV}.P]\!]_p$	$[\![\$x]\!]_e = \lambda b.\langle b\rangle_b(x)$

Table 2. A selection of predicate and expression operator definitions

do not encode type information about the binding type in predicates and so can arbitrarily compose predicates that reference disjoint variable sets. Next we introduce a type to represent core expressions:

```
typedef 'a uexpr = "{f:: 'a binding ⇒ 'a. ∃t. ∀b. f b : t}"
```

Expressions, unlike in [19], are defined purely semantically; syntax is introduced definitionally and is therefore extensible. An expression, semantically, is a function from a binding to the value that the binding produces. For example $[\![2 * x]\!] = \{\{x \mapsto 1\} \mapsto 2, \{x \mapsto 2\} \mapsto 4 \cdots\}$. We also require that there exists a type that every evaluation of the expression has. This ensures that predicates containing expressions can be consistently typed.

We now define the predicate operators, as illustrated in Table 2. We introduce operators denotationally by lifted definition; no formal syntax tree is necessary. The predicate semantic function $[\![\_]\!]_p$ corresponds to the `upred` representation function, and similarly $[\![\_]\!]_e$ for expressions. Hence the predicate operators are all certified closed under well-defined predicates by the type-system. Operators that involve expressions, such as substitution, depend on type correctness of the expressions to ensure well-formedness.

The lattice operators map to their set equivalents, in common with both [19] and [9]. The existential quantifier follows [19] and is defined as binding override comprehension, consistent with our deep encoding of variables. It overrides the variables quantified (vs) with all possible arbitrary bindings (b'), thus hiding their values. The universal quantifier is then defined using the De Morgan equivalence. Definitions for universal closure and the refinement relation follow without difficulty. Application of a permutation to a predicate simply applies it to every possible binding.

We also denote some basic expression operators using binding functions. LitE, a literal, returns a value of the model type `'a`, Op1E applies a unary HOL function $f :: $ `'a` $\Rightarrow$ `'a` to an expression, and $\$x$, returns the value of a variable. These expression constructs, unlike [9] are not subject to HOL typing, and so associated laws must prove well-typedness. Hence at the user level the polymorphic expression model can be used instead (see Section 6).

$\mathsf{true}[e/x] = \mathsf{true}$	$(\exists vs.P)[e/x] = \exists vs.(P[e/x])$ [when $x \notin vs$]
$(P \vee Q)[e/x] = P[e/x] \vee Q[e/x]$	$(\$x)[e/x] = e$ [when $e \rhd_e x$]

Table 3. Substitution laws

$\frac{-}{xs \sharp \mathsf{true}}$	$\frac{xs \sharp P \qquad xs \sharp Q}{xs \sharp (P \vee Q)}$	$\frac{xs \sharp P}{xs \sharp (\neg P)}$	$\frac{ys \sharp P}{xs \cup ys \sharp (\exists xs. P)}$	$\frac{\{x\} \sharp P}{P[v/x] = P}$

Table 4. Unrestricted Variables

Substitution, $P[e/x]$, is normally implemented as a function on syntax, but since we do not have syntax it has a purely semantic account. This has the advantage that substitution laws are introduced incrementally by proof and are thus correct by construction. Substitution is defined as the set of bindings, such that updating the value of x in b with the evaluation of e under b yields a binding that is in P. We then prove the usual substitution laws; a selection is shown in Table 3. Likewise, many of these laws are subject to typing obligations, for instance to substitute e for x the types must match (written $e \rhd_e x$).

5.4 Alphabetised Predicates

We now extend our core predicate model with alphabets: the maximal set of variables to which a predicate can refer. We first formalise "unrestricted variables" to give a semantic account of fresh variables using the `UNREST` predicate, adapted from [19]. A variable x is unrestricted in predicate P, written $\{x\} \sharp P$, if the satisfiability of P is independent of x. Unrestriction therefore corresponds to the fresh variables of a predicate's normal form. For instance it follows that $\{x\} \sharp (x = 1 \vee \mathsf{true})$, even though x is syntactically present. This is a sufficient notion, as core predicates are equivalent to their normal form. We have likewise proved many laws about unrestricted variables, a selection of which is shown in Table 4. These laws are used to satisfy provisos in laws that depend on freshness of particular variables, and provide predicate alphabet theorems. With unrestriction formalised we can construct the model of alphabetised predicates:

```
typedef 'a apred = "{(a, P :: 'a upred). - a ♯ P ∧ finite a}"
```

An alphabetised predicate is a core predicate P together with a finite set of variables a (the alphabet), such that P restricts no more variables that those in a. If a variable is not in the alphabet it is certainly unrestricted, though variables in the alphabet may also be unrestricted. Alphabets give a more syntactic account of fresh variables: it is sufficient to mention a variable for it to be in the alphabet. Explicit alphabets are a major feature of deep embeddings in contrast to [9] that can represent alphabets only implicitly as HOL types. As for the core predicate model, we can use lifting to define the alphabetised predicate operators:

```
lift_definition OrA :: "'a apred ⇒ 'a apred ⇒ 'a apred"
is "λ (a, p) (b, q). (a ∪ b, p ∨ₚ q)"
  by (auto intro: UNREST_subset UNREST_OrP)
```

The definition of `OrA` takes the union of the two alphabets and uses $\vee_p$ to compose the underlying core predicates. This requires a proof that $p \vee q$ is unrestricted on $-(a \cup b)$ and that $a \cup b$ is finite. The former follows by the unrestriction subset law and the latter from the fact that the union of finite sets is finite. We use lifting to build equivalent alphabetised operators for each of the core predicate operators in a similar fashion with lifting. This ensures that the alphabetised predicate operators are closed without need for further proof.

$$
\begin{array}{rcl@{\qquad}rcl}
[\![\mathrm{II}_{xs}]\!]_p & \triangleq & \{b \mid \forall x \in xs. \langle b \rangle_b(x') = \langle b \rangle_b(x)\} & P^{\smile} & \triangleq & SS \bullet P \\
P \lhd b \rhd Q & \triangleq & (b \wedge P) \vee (\neg b \wedge Q) & x :=_{xs} e & \triangleq & (x' = e) \wedge \mathrm{II}_{xs - \{x\}} \\
P \,;\, Q & \triangleq & (\exists D_2 . (SS1 \bullet P) \wedge (SS2 \bullet Q)) & b * P & \triangleq & (b \wedge P)^{\star} \wedge \neg b' \\
p\{Q\}r & \triangleq & (p \Rightarrow r') \sqsubseteq Q & Q \ \mathsf{wp}\ r & \triangleq & \neg(Q \,;\, \neg r)
\end{array}
$$

Table 5. Relational Operators

5.5 Relations and Imperative Programming

We conclude the section with the relational calculus operators for core predicates, as shown in Table 5, some of which use permuations. Permutation SS swaps dashed and undashed variables. SS1 swaps dashed and doubly dashed variables. SS2 swap undashed and doubly dashed variables. We use doubly dashed variables to represent "intermediate" variables in a sequential composition.

Skip (II_{xs}) consists of the set of bindings where the dashed and undashed variables in xs correspond. Sequential composition is defined using existential quantification and conjunction. It synchronises the outputs from P with the inputs from Q by renaming them using SS1 and SS2, respectively, so they correspond. This set of variables is then hidden so that only the inputs from P and outputs from Q remain visible. Similarly, permuations are used for relational converse, $P^{\smile}$, defined as the application of SS, which effectively flips the direction. Assignment, $x :=_{xs} e$, is implemented as a conjunction of an equality on the post-state of x and a skip on all other variables. We implement a while loop ($b * P$) via a link to Kleene algebra [11]. Thus the algebraic hiearchies from Isabelle can be used to import properties into the UTP predicate model.

In this section we mechanised a predicate model retaining the expressivity of [19], whilst also being modular in its handling of value models, and partly well-formed by construction. In the following Sections we will solve the problems of typing obligations by integrating type inference in expressions, and automated proof, that will also make the infrastructure practically applicable.

6 Polymorphic expressions

In this section we explore typing in UTP predicates and expressions. In Section 5.1 we declared the type-class `VALUE` that introduces a model type, which we now refer to as `'m`, and an associated typing relation. This typing relation is *orthogonal* to HOL's own type system and therefore it is necessary to manually prove well-typedness of values, variables and expressions. This is unavoidable as we need the expressivity of a deep model to express generic laws involving values and types. Nevertheless it means that in the core expression model type coercisions must be inserted manually and that the proof burden is greatly increased in comparison to Isabelle/Circus, which directly uses HOL typing. So if we desire both expressivity and host-logic typing, we need to overcome this problem.

Our solution is to bind the UTP value model typing relation to the HOL typing relation. This means that the Isabelle type inference system can effectively be harnessed to discharge UTP typing obligations, thus satisfying criterion **(4)**, whilst also aiding criterion **(3)**. We do this by linking specific types in the

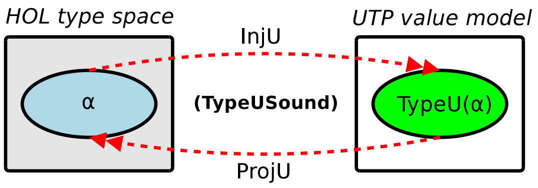

Fig. 3. Mapping HOL types to UTP types

underlying UTP model with types in HOL. This is implemented through two polymorphic functions, `InjU` and `ProjU`, that convert between these two levels and are inverses; this is illustrated in Figure 3 for a type α.

This allows us to form a polymorphic expression type `('a, 'm) pexpr` that carries, along with value model `'m`, the expression type `'a`. For example, we can enter expression "$\$x + 5$" and HOL will infer that it has the type `(int, 'm :: INT_SORT) pexpr`, and that x is a variable of type `int`. Conversley "$\{1\} + 5$" is correctly rejected. We also implement *type erasure*, using the injection functions, which converts a polymorphic expression into a core expression. It disgards the `'a` parameter meaning that differently typed expressions can be compared. The polymorphic expression model can then be grounded in the core predicate model, the former simply acting as an additional layer.

Implementation of the type injection and projection functions is non-trivial as HOL type-classes can range over only a single type parameter. However our injection functions need to range over two parameters: the model type (`'m`) and injectable type (`'a`). Therefore we use Isabelle's *polymorphic constants* to create these functions. Polymorphic constants work in a similar way to type-class signature functions but can range over multiple type parameters.

```
consts TypeU :: "'a itself ⇒ ('m :: VALUE) utype"
       InjU  :: "'a ⇒ 'm :: VALUE"
       ProjU :: "'m :: VALUE ⇒ 'a"
```

`TypeU` effectively maps HOL types to UTP types. The "itself" type is a special HOL singleton type used to refer to types on the value level, and can be constructed by application of the special `TYPE` function to a HOL type name. `InjU` and `ProjU` define the injection and projection functions from a type `'a` into model type `'m`, which must be a member of the `VALUE` class. `InjU` can alternatively be viewed as a type erasure operator, since it converts from a value containing type data, to one without in the underlying value model. Indeed it is `InjU` upon which we base type erasure in polymorphic expressions.

However, just declaring these constants will not suffice, as unlike for type-classes they are not subject to laws, and we want to reason about injectable types generically. We could declare axioms, but that would not be definitional. Instead we create the following definition to act as a pseudo two-parameter type-class.

```
definition TypeUSound :: "'a itself ⇒ 'm itself ⇒ bool" where
"TypeUSound a m ⟷ (∀ x::'a. (InjU x :: 'm) : TypeU a)
                   ∧ (∀ x::'a. 𝒟 x ⟶ 𝒟 (InjU x :: 'm))
                   ∧ (∀ x::'m. x :! TypeU a ⟶ 𝒟 (ProjU x :: 'a))
                   ∧ (∀ x::'a. ProjU (InjU x :: 'm) = x)
                   ∧ (∀ x :! TypeU a. (InjU (ProjU x :: 'a) :: 'm) = x)"
```

TypeUSound asserts that an injectable type 'a can be used soundly in value model 'm. The five properties within show correspondance between typing in the UTP value model, and typing in the equivalent HOL type, as in Figure 3. Note that x :! a means that x is both typed by a and also defined.

Therefore if a given HOL type 'a is TypeUSound for a model 'm, then laws that would otherwise require proof of type correctness can be applied through HOL typing. This greatly reduces the proof burden: laws that rely on conversions between HOL value and UTP values simply need a TypeUSound assertion as an assumption. This is much simpler than performing a traversal of the equivalent core expression to prove type-correctness. Using this approach, we construct a typed version of variables, for variable type 'a and model type 'm:

```
typedef ('a, 'm) pvar = "UNIV :: (NAME * bool) set" ..

definition PVAR_VAR :: "('a, 'm) pvar ⇒ 'm uvar"
where "PVAR_VAR v = MkVar (pvname v) (TypeU TYPE('a)) (pvaux v)"
```

A pvar is simply a pairing of a NAME and a Boolean auxiliary flag. The type of the variable is carried by the additional type variable, which is not referred to internally. Type erasure is then implemented via the function PVAR_VAR, which constructs an untyped UTP variable using the name, auxiliary flag and equivalent model type using TypeU. This, and other type erasure operators, are given the overloaded syntax $x\downarrow$. Similarly we also implement polymorphic expressions.

```
typedef ('a, 'm) pexpr = "UNIV :: ('m binding ⇒ 'a) set"
morphisms DestPExpr MkPExpr ..
```

A polymorphic expression is a function from a binding to a value of the correct type. We can use InjU and ProjU to marshal between UTP values in the binding and HOL typed functions. For instance, we implement variable lookup as follows:

```
definition PVarPE :: "('a, 'm) pvar ⇒ ('a, 'm) pexpr" where
"PVarPE x ≡ MkPExpr (λ b. ProjU (⟨b⟩_b (x↓)))"
```

This constructs a polymorphic expression which (1) erases the incoming polymorphic variable, (2) looks up the value associated with this variable in the binding, and (3) projects it to a HOL value of the correct type. We can also construct combinators which are typed by HOL. The following definition applies a HOL unary function to a correctly typed expression:

```
definition
  Op1PE :: "('a ⇒ 'b) ⇒ ('a, 'm) pexpr ⇒ ('b, 'm) pexpr"
where "Op1PE f v = MkPExpr (λ b. f (⟦v⟧_*b))"
```

Using this definition the input types must match the function type, ensuring type-correctness by construction. Moreover, we can prove a useful type-erasure law: (Op1PE f v)$\downarrow$ = Op1E (ProjU $\circ$ f $\circ$ InjU) (v$\downarrow$), which allows manipulation of the expression using laws establised in the core predicate model.

We now have a predicate model that is well-formed by construction down the level of expressions through the imposition of HOL typing. Proofs need not handle typing obligations, since the TypeUSound "class" ensures that a well-typed polymorphic expression is also a well-typed core expression. Nevertheless, the predicate model is still deep, and we retain all the meta-logical operators from Section 5, thus partially reconciling deep and shallow.

7 Automating Proof by Transfer

In this section we look at proof automation in Isabelle/UTP, thus satisfying criterion **(3)**. Equality of UTP predicates is implied by equality of the underlying binding sets (the denotation), thus an equality proof can be performed by decomposing a predicate into its constituent binding functions and showing their equivalence. However this kind of proof is of too fine a granularity and the predicate structure is lost. This is evidenced by the complexity of proofs in ProofPower-Z UTP [19] like that of sequential composition associativity. In comparison Isabelle/Circus can discharge such goals instantly by application of the auto tactic due to Isabelle's large mechanised theory library for HOL relations. Although not identical to UTP relations they are very similar, and so use of the associated laws in UTP relational proofs ought to be possible.

Our approach makes the laws applicable by binding UTP relations to HOL relations via suitable injections and projections, similar to our approach to typing in Section 6. Specifically, we link a UTP theory to an existing HOL theory and prove that results can be transferred. This greatly improves proof automation in Isabelle/UTP by enabling reuse of HOL theory libraries. So far we have developed tactics for theories like predicates, relations, and expressions. Conceptually, a UTP proof tactic consists of three parts, which together form the link:

1. *interpretation function* – maps elements of a UTP theory to elements of a target domain, in which we desire to perform the proof;
2. *transfer rules* – demonstrate how results in the target domain map to results in the UTP theory;
3. *congruence rules* – map operators from the signature of the UTP theory to operators in the target domain.

Such tactics are partial: they apply only to the operators of the theory, leaving unrecognised operators uninterpreted. For instance our UTP predicate tactic, utp-pred-tac, only solves problems built using the predicate operators and our relational tactic, utp-rel-tac, only solves problems in relational calculus. Therefore, proof automation is all about picking the correct level of abstraction. The algorithm for performing proof using these tactics is typically of this form:

Interpretation Function
EvalR :: α upred $\Rightarrow$ (α WF-REL-BINDING) rel ($[\![\_]\!]\mathcal{R}$)

Transfer Theorems
$(P = Q) \longleftrightarrow ([\![P]\!]\mathcal{R} = [\![Q]\!]\mathcal{R})$
$(P \sqsubseteq Q) \longleftrightarrow ([\![P]\!]\mathcal{R} \supseteq [\![Q]\!]\mathcal{R})$

Congruence Rules (Selection)
$[\![\mathsf{false}]\!]\mathcal{R} = \emptyset$
$[\![P \vee_p Q]\!]\mathcal{R} = [\![P]\!]\mathcal{R} \cup [\![Q]\!]\mathcal{R}$
$[\![\mathrm{II}]\!]\mathcal{R} = \mathsf{Id}$
$[\![P \mathbin{;} Q]\!]\mathcal{R} = [\![P]\!]\mathcal{R} \circ [\![Q]\!]\mathcal{R}$
$[\![P']\!]\mathcal{R} = ([\![P]\!]\mathcal{R})^{-1}$
$[\![P^\star]\!]\mathcal{R} = ([\![P]\!]\mathcal{R})^*$

Fig. 4. utp-rel-tac: Transfer function and rules

1. **Apply a transfer rule to the proof goal** (for a suitable target model);
2. **Apply the associated congruence rules**, to interpret the UTP theory operators into operators of the target domain;
3. **Apply a builtin HOL tactic**, for instance simp, auto or sledgehammer.

We exemplify this with our relational tactic, utp-rel-tac; the main definitions and theorems are in Figure 4. UTP and HOL relations, though similar, are mismatched as the former consists of a set of bindings with undashed and dashed variables, whilst the latter consists of a set of pairs. We resolve this with the function `EvalR`, written $[\![P]\!]\mathcal{R}$, which converts a UTP predicate into a HOL relation over a special binding type, `WF_REL_BINDING`, which has only undashed variables, the dashedness replaced by the pairs. We then prove two transfer theorems that allow equality and refinement conjectures to be transferred to equality and subset conjectures in HOL. Finally we prove a collection of congruence properties for suitable operators, including sequential composition, skip and Kleene star.

If we apply utp-rel-tac to goal "$p \operatorname{;} \mathbf{false} \operatorname{;} q = \mathbf{false}$", the following steps occur:

(1) Transfer:	$[\![p \operatorname{;} \mathbf{false} \operatorname{;} q]\!]\mathcal{R} = [\![\mathbf{false}]\!]\mathcal{R}$	$\longleftrightarrow$	
(2) Map operators:	$[\![p]\!]\mathcal{R} \circ \emptyset \circ [\![q]\!]\mathcal{R} = \emptyset$	$\longleftrightarrow$	
(3) HOL tactic:	$\emptyset = \emptyset$	$\longleftrightarrow$	True $\Box$

Thus, this tactic can prove many relational conjectures automatically, providing a similar level of proof automation to Isabelle/Circus. For example, we can both express and automatically prove the law of commutativity of assignments:

```
theorem AssignR_commute:
  fixes x y :: "('a, 'm) pvar"
  assumes
    "x↓ ∈ D₀" "y↓ ∈ D₀" "TYPEUSOUND('a, 'm)" "¬ aux (x↓)" "¬ aux (y↓)"
    "D₁ ♯ e" "D₁ ♯ f" "{x↓} ♯ f" "{y↓} ♯ e" "x↓ ≠ y↓"
  shows "‘(x := e); (y := f)‘ = ‘(y := f); (x := e)‘"
  using assms by (utp_prel_auto_tac)
```

We use a variant of utp-rel-tac called utp-prel-tac, that also handles polymorphic expressions, and compose it with auto. The law is subject to static typing: `e` must match the type of `x` and likewise for `f` and `y`. This shows how the deep model, host-logic typing, and proof tactics combine to provide a mechanisation which satisfies all the criteria in Section 4. We have used our tactics to develop a large library of algebraic laws (> 300 at time of writing), which when combined with sledgehammer further improves automation. This library includes most of the standard predicate and relation laws from [15] and a proof that UTP relations form a *Relation Algebra* [23], thus also partly validating our model.

8 Conclusion

We introduced Isabelle/UTP, a novel mechanisation of UTP in Isabelle. We described its modular value model, predicate model, and associated operators. We showed how to adapt HOL type checking to discharge typing obligations, reducing the proof burden. Finally, we introduced our approach to proof, which gives a similar level of automation to that of shallower models. All in all we have shown that Isabelle/UTP combines many advantages of previous the embeddings in a sound and usable framework. This provides a powerful platform on which to realise the UTP semantic framework which we hope to apply to both theory engineering and verification. It remains to be seen what effect the semantic overhead imposed by our embedding has on proof in larger examples when compared to, for instance, Isabelle/Circus. Isabelle/UTP can be obtained from our website.[1]

Along with the work presented in the paper we have also made substantial progress in mechanising UTP theories, including the theory of designs (total correctness) for which we have mechanised most laws in the UTP book, the theory of reactive processes, and the theory of CSP. Indeed, we have found the combination of Isabelle's Isar proof language, the sledgehammer tool [1], and our own tactics provides a very pleasing approach to creating automated readable proofs, as demonstrated in our UTP tutorial [12] using an early version of Isabelle/UTP. Isabelle/UTP also provides the basis for a mechanised semantics for CML [26], a modelling language for *Systems of Systems*, in the COMPASS project[2]. It is also used by the theorem prover plugin of the associated *Symphony*[3] tool-chain to discharge proof obligations in CML specifications [7]. We believe this goes much of the way to validating its applicability for verification tasks.

In future work, we will develop more proof tactics based on additional theory links. For example *Design Algebra* [13] provides an interesting and potentially exploitable link between the theory of designs and Kleene algebra over a matrix. We are also developing additional theories in Isabelle/UTP, for example, the theory of operational semantics, and links to various process algebras. Although this paper did not formally tie down the notion of UTP theory and their combination, Isabelle/UTP does provide the basis for this when aided by Isabelle's HOL-Algebra library which we will fully exploit in the future. Our long-term goal is to mechanise, and thus firmly validate, all the laws from the UTP book and associated publications. This is an important step toward a firmly grounded mechanised repository of UTP theory for engineering semantics.

Acknowledgements: This work is supported by EU FP7 project COMPASS (grant 287829) and EPSRC grant EP/H017461/1.

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. R. Boulton, A. Gordon, M. Gordon, J. Harrison, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In *Proc. IFIP Intl. Conf. on Theorem Provers in Circuit Design*, pages 129–156, 1993.
3. J. Bowen and M. Gordon. Z and HOL. In J.P. Bowen and J.A. Hall, editors, *Z User Workshop*, pages 141–167. Springer, June 1994. Cambridge.

[1] http://cs.york.ac.uk/~simonf/utp-isabelle/
[2] For more information please see http://www.compass-research.eu/
[3] Symphony can be obtained from http://symphonytool.org/

4. Andrew Butterfield. Saoithín: A theorem prover for UTP. In *UTP 2010*, volume 6445 of *LNCS*, pages 137–156. Springer, 2010.
5. Andrew Butterfield. The logic of *U·(TP)*^{2}. In *UTP 2012*, volume 7681 of *LNCS*, pages 124–143. Springer, 2012.
6. A. Cavalcanti, A. Wellings, and J. Woodcock. The Safety-Critical Java memory model formalised. *Formal Aspects of Computing*, 25(1):37–57, 2013.
7. L. Couto, S. Foster, and R. Payne. Towards verification of constituent systems through automated proof. In *Proc. Workshop on Engineering Dependable Systems of Systems (EDSoS)*. ACM CoRR, 2014.
8. 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.
9. A. Feliachi, M.-C. Gaudel, and B. Wolff. Isabelle/Circus: a process specification and verification environment. In *VSTTE 2012*, volume 7152 of *LNCS*, pages 243–260. Springer, 2012.
10. S. Foster, A. Miyazawa, J. Woodcock, A. Cavalcanti, J. Fitzgerald, and P. Larsen. An approach for managing semantic heterogeneity in systems of systems engineering. In *Proc. 9th Intl. Conf. on Systems of Systems Engineering*. IEEE, 2014.
11. S. Foster, G. Struth, and T. Weber. Automated engineering of relational and algebraic methods in Isabelle/HOL. In H. Swart, editor, *RAMICS 2011*, volume 6663 of *LNCS*, pages 52–67. Springer, 2011.
12. S. Foster and J. Woodcock. Unifying theories of programming in Isabelle. In *ICTAC 2013 School on Software Engineering*, volume 8050 of *LNCS*, pages 109–155. Springer, 2013.
13. W. Guttmann and B. Möller. Normal design algebra. *The Journal of Logic and Algebraic Programming*, 79(2):144–173, 2010.
14. T. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
15. T. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
16. B. Huffman and O. Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *3rd Intl. Conf. on Certified Programs and Proofs*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013.
17. M. Iancu and F. Rabe. Formalising foundations of mathematics. *Mathematical Structures in Computer Science*, 21:883–911, 8 2011.
18. T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
19. M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying theories in ProofPower-Z. In *UTP 2006*, volume 4010 of *LNCS*, pages 123–140. Springer, 2007.
20. M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for Circus. *Formal Aspects of Computing*, 21:3–32, 2009.
21. J. Perna and J. Woodcock. UTP Semantics for Handel-C. In Andrew Butterfield, editor, *UTP 2010*, volume 5713 of *LNCS*, pages 142–160. Springer, 2010.
22. J. Spivey. *The Z-Notation - A Reference Manual*. Prentice Hall, 1989.
23. A. Tarski. On the calculus of relations. *J. Symbolic Logic*, 6(3):73–89, 1941.
24. C. Urban. Nominal techniques in Isabelle/HOL. In *Proc. 20th Intl. Conf. on Automated Deduction*, volume 3632 of *LNCS*, pages 38–53. Springer, 2005.
25. J. Woodcock and A. Cavalcanti. The Semantics of Circus. In *ZB 2002*, volume 2272 of *LNCS*, pages 184–203. Springer, 2002.
26. J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry. Features of CML: a Formal Modelling Language for Systems of Systems. In *Proc. 7th Intl. Conference on System of Systems Engineering*. IEEE, July 2012.
27. F. Zeyda and A. Cavalcanti. Mechanical reasoning about families of UTP theories. In *SBMF*, volume 240 of *ENTCS*, pages 239 – 257, 2008.