Unifying Theories in Isabelle/HOL

Conference Paper · November 2010

DOI: 10.1007/978-3-642-16690-7_9 · Source: DBLP

Abderrahmane Feliachi
Université Paris-Sud 11

Marie-Claude Gaudel
Laboratoire de Recherche en Informatique

Burkhart Wolff
Laboratoire de Recherche en Informatique

Unifying Theories in Isabelle/HOL

Abderrahmane Feliachi, Marie-Claude Gaudel and Burkhart Wolff

[1] Univ Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
[2] CNRS, Orsay, F-91405, France
{Abderrahmane.Feliachi, Marie-Claude.Gaudel, Burkhart.Wolff}@lri.fr [*]

Abstract. In this paper, we present various extensions of Isabelle/HOL by theories that are essential for several formal methods. First, we explain how we have developed an Isabelle/HOL theory for a part of the Unifying Theories of Programming (UTP). It contains the theories of alphabetized relations and designs. Then we explain how we have encoded first the theory of reactive processes and then the UTP theory for CSP. Our work takes advantage of the rich existing logical core of HOL.
Our extension contains the proofs for most of the lemmas and theorems presented in the UTP book. Our goal is to propose a framework that will allow us to deal with formal methods that are semantically based, partly or totally, on UTP, for instance CSP and *Circus*. The theories presented here will allow us to make proofs about such specifications and to apply verified transformations on them, with the objective of assisting refinement and test generation.

Keywords: UTP, Theorem Proving, Isabelle/HOL, CSP, *Circus*

1 Introduction

The fundamental problem of the *combination* of programming paradigms has raised significant interest recently; a framework to combine different languages describing various facets and artifacts of software development in a seamless, logically consistent way is vital for its solution. Hoare & He gave one of the most significant approaches towards unification [10]. A relational theory between an initial and subsequent states or observations of computer devices is used to give meaning to specifications, designs, and programs. States are expressed as predicates over observational variables. They are constrained by invariants, called healthiness conditions, that characterize theories for imperative, communicating, or reactive processes and their designs.

The UTP framework has proved to be powerful enough for developing a theory of CSP processes, and more recently for giving semantics to languages like *Circus* [13] that combines CSP and Z features and enables states, concurrency and communications to be easily expressed in the same specification.

The motivation of this paper is to provide effective deductive support of UTP theories, in particular those related to *Circus*. Effective deduction is needed for

[*] This work was partially supported by the Digiteo Foundation.

practically useful transformations on *Circus* specifications, and for our objective of refinement support and automated test generation. Therefore, it is of major importance to find a semantic representation that has a small "representational distance" to the logic used in the implementing proof-environment: since *Circus* comprises typed sets, only frameworks for higher-order logics (HOL, Z, ...) are coming into consideration.

Textbook UTP presentations reveal a particular syntactic flavor of certain language aspects, a feature inherited from the Z tradition. The UTP framework is centered around the concept of *alphabetized* predicates, relations, etc, which were written $(\alpha P, P)$ where αP is intended to produce the *alphabet* of the predicate P implicitly associated to a superset of the free variables in it. In prior works based on *ProofPower*[2], providing a formal semantics theory for UTP in HOL [12], [14], [15], the authors observed the difficulty that "the name of a variable is used to refer both to the name itself and to its value". For instance, in the relation

$$(\{x, x'\}, x > 0 \wedge (x' = x + 1 \vee x' = x - 1)), \qquad (1)$$

the left-most x, x' indicates the names x resp. x', while the right-most x, x' stand for their value. Since Oliveira et al. [12] aimed at the proof of refinement laws, the authors saw no alternative to proving meta-theorems using a so-called *deep embedding*; thus, an explicit data type for abstract syntax and an explicit semantic interpretation function was defined that relates syntax and semantic domain. However, such a representation has a number of drawbacks, both conceptually as well as practically wrt. the goal of efficient deduction:

1. there are necessarily ad-hoc limitations of the cardinality of the semantic domain VAL (e.g. sets are limited to be *finite* in order to keep the recursive definition of the domain well-founded),
2. the alphabet uses an untyped presentation — there is no inherent link from names and their type, which must be established by additional explicit concepts adding a new layer of complexity, and
3. the reasoning over the explicit alphabet results in a large number of nasty side-conditions ("provisos") hampering deduction drastically. For example, the rule for the sequential composition $\_;_C\_$ in *Circus* reads as follows:

   $$\forall\, a, b, c : CA \mid \alpha a = \alpha b \wedge \alpha b = \alpha c \wedge\ a;_C (b;_C c) = (a;_C b);_C c \qquad (2)$$

   where CA abbreviates $CIRCUS\_ACTION$.

In contrast to this "deep embedding" approach we opt for a "shallow embedding". The characterizing feature for the latter is the following: if we represent an object-language expression E of type T into the meta-language by some expression E' of type T', then the mapping is injective for both E and T (provided that E was well-typed with T). In contrast, conventional representations are a surjective map from object-expressions to, say, a data type AST (abstract syntax tree) and therefore not shallow. Due to injective map on types, the types are implicit in a shallow representation, and thus reference to them in provisos in rules is unnecessary. It means that type-inference is used to perform a part of the deduction task beforehand, once and for all, as part of a parsing process prior to deduction.

At this point, we will already recklessly reveal the only essential idea of this paper to the knowledgeable reader: we will represent equation (1) by the λ-abstraction:

$$\lambda\, \sigma \bullet \sigma.x > 0 \wedge (\sigma.x' = \sigma.x + 1 \vee \sigma.x' = \sigma.x - 1) \tag{3}$$

having the record type $\langle x \rightsquigarrow \mathbb{Z}, x' \rightsquigarrow \mathbb{Z}, ...\rangle \Rightarrow \text{bool}$, which is, in other words, a set of records in HOL. The reader familiar with SML-like record-pattern-match notation may also recognize expression (3) as equivalent to:

$$\lambda\{x, x', ...\} \bullet x > 0 \wedge (x' = x + 1 \vee x' = x - 1)$$

Note that in record notation, the order of the names is insignificant (in contrast to, say, a representation by tuples). Further note that we use *extensible* records — the dots represent the possibility of their extensions, allowing to build up the UTP in an incremental way similar to Brucker and Wolff's approach [6].

Represented in the form of expression (3), the rule (2) above is practically an immediate consequence of rules of a HOL-library. The function αP becomes a meta-function (implemented in the meta language of the target HOL system, typically SML), and the notation $(\alpha P, P)$ is a particular pretty-print of P for sets of records.

Note that in a shallow embedding, the injective representation function must not be a one-to-one translation of operator symbols; rather, it can introduce coercions in E' on the basis of object-language types. For example, it can be necessary to coerce isomorphically a $\langle x \rightsquigarrow \mathbb{Z}, x' \rightsquigarrow \mathbb{Z}, ...\rangle$ set-predicate to a $(\langle x \rightsquigarrow \mathbb{Z}, ...\rangle \times \langle x \rightsquigarrow \mathbb{Z}, ...\rangle)$ set-relation in order to support the semantics of the UTP dash-notation x' in terms of relational composition. Similar compiler techniques are necessary to add or remove fields in extensible records, for example when entering and leaving the scope of a local variable declaration.

Another price we are ready to pay is that there may be *rules* in Textbook UTP, which must be implemented by a *rule scheme* in our representation. That is, there will be specific tactic support that implements a rule scheme, e. g., by inserting appropriate coercions in a more general rule and applying the result in a specific context. This technique has been used for the Z schema calculus by Brucker et al. [3].

As concrete implementation platform, we chose Isabelle/HOL [11] which is a well established tool for formal proof development. As member of the LCF-style prover family, it offers support for user-programmed extensions in a logically safe way. This choice is motivated by our wish to exploit, in a future step, the semantic *Circus* theory developed here with HOL-TestGen [4, 5], a powerful test-case generation system that has been built on top of the specification and theorem proving environment Isabelle/HOL.

The paper is organized as follows: Section 2 recalls briefly some useful aspects of various background concepts: UTP, Isabelle/HOL, some advanced aspects of HOL; Section 3 presents how we have expressed in HOL the part of UTP that is relevant for the *Circus* semantics; Section 4 introduces the *Circus* language and the theory we have developed in HOL from its denotational semantics; Section 5 gives a small example of a *Circus* specification defined in Isabelle/HOL; and the last section summarizes our current contributions and sketches our future work.

2 Background

2.1 Isabelle and Higher-order Logic

Higher-order logic (HOL) [9, 1] is a classical logic based on a simple type system. It provides the usual logical connectives like $\_ \wedge \_$, $\_ \to \_$, $\neg \_$ as well as the object-logical quantifiers $\forall\, x \bullet P\, x$ and $\exists\, x \bullet P\, x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f : : \alpha \Rightarrow \beta$. HOL is centered around extensional equality $\_ = \_ : : \alpha \Rightarrow \alpha \Rightarrow \text{bool}$. HOL is more expressive than first-order logic, since, e. g., induction schemes can be expressed inside the logic. Being based on some polymorphically typed λ-calculus, HOL can be viewed as a combination of a programming language like SML or Haskell and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is a logical embedding of HOL into the generic proof assistant Isabelle. The (original) simple-type system underlying HOL has been extended by Hindley/Milner style polymorphism with type-classes similar to Haskell. While Isabelle/HOL is usually seen as "proof assistant", systems like HOL-TestGen[4, 5] also use it as symbolic computation environment. Implementations on top of Isabelle/HOL can re-use existing powerful deduction mechanisms such as higher-order resolution, tableaux-based reasoners, rewriting procedures, Presburger Arithmetic, and via various integration mechanisms, also external provers such as Vampire and the SMT-solver Z3. Isabelle/HOL offers support for a particular methodology to extend given theories in a logically safe way: A theory-extension is *conservative* if the extended theory is consistent provided that the original theory was consistent. Conservative extensions can be *constant definitions*, *type definitions*, *datatype definitions*, *primitive recursive definitions* and *well-founded recursive definitions*.

For example, typed sets were built in the Isabelle libraries conservatively on top of the kernel of HOL as functions to bool; consequently, the constant definitions for membership is as follows:[3]

```
types  α set   = α  ⇒ bool
definition  Collect  ::   (α ⇒ bool) ⇒   α set  −−− set comprehension
where      " Collect  S ≡  S"
definition  member ::  α ⇒ bool                   −−− membership test
where      "member s S ≡S s"
```

[3] To increase readability, we use a slightly simplified presentation.

Isabelle's powerful syntax engine is instructed to accept the notation $\{x \bullet P\}$ for $\mathsf{Collect}\,(\lambda\,\mathsf{x}.\ \mathsf{P})$ and the notation $s \in S$ for $\mathsf{member\ s\ S}$. As can be inferred from the example, constant definitions are axioms that introduce a fresh constant symbol by some closed, non-recursive expressions; this type of axiom is logically safe since it works like an abbreviation. The syntactic side-conditions of this axiom are mechanically checked, of course. It is straight-forward to express the usual operations on sets like $\_\cup\_, \_\cap\_ :: \alpha\,\mathrm{set} \Rightarrow \alpha\,\mathrm{set} \Rightarrow \alpha\,\mathrm{set}$ as conservative extensions, too, while the rules of typed set-theory were derived by proofs from these definitions.

2.2 Advanced concepts of the HOL-Language

Similarly, a logical compiler is invoked for the following statements introducing the types option and list:

datatype α option = None | Some α
datatype α list = Nil | Cons a l

Here, [] or $a\#l$ are an alternative syntax for Nil or Cons $a\ l$; moreover, $[a, b, c]$ is defined as alternative syntax for $a\#b\#c\#[]$. These (recursive) statements were internally represented in by internal type- and constant definitions. Besides the *constructors* None, Some, there are match-operations like:

case x of None $\Rightarrow F$ | Some $a \Rightarrow G\ a$.

Finally, there is a compiler for primitive and well-founded recursive function definitions.

Isabelle/HOL also provides a rich collection of library theories like sets, pairs, relations, partial functions lists, multi-sets, orderings, and various arithmetic theories which only contain rules derived from conservative definitions. Setups for the automated proof procedures like simp, auto, and the arithmetic types such as int have been done.

Isabelle/HOL's support for *extensible records* is of particular importance for this work. Record types are denoted, for example, by:

record T = a :: T_1
b :: T_2

which implicitly introduces the record constructor ⦇a :=e1,b :=e2⦈ and the update of record r in field a, written as r⦇a := x⦈. Extensible records are represented internally by cartesian products with an implicit free component δ, i.e. in this case by a triple of the type $\mathsf{T}_1 \times \mathsf{T}_2 \times \delta$. Thus, the record T can be extended later on using the syntax:

record ET = T +
c :: T_3

The key point is that theorems can be established, once and for all, on T types, even if future parts of the record are not yet known, and reused in the later definition and proofs over ET-values. Thus, we can model the effect of defining the alphabet of UTP processes incrementally while maintaining a fully typed shallow embedding with full flexibility on the types T_1, T_2 and T_3. In other words, extensible records give us the means to implement the dots in the representation type of the alphabetized predicate (3):

$$\langle x \rightsquigarrow \mathbb{Z}, x' \rightsquigarrow \mathbb{Z}, ...\rangle \mathrm{set}$$

shown in the introduction.

3 Representing UTP in HOL

3.1 Core UTP

In this section, we present the most general features of UTP: the concept of alphabetized predicates, and sub-concepts such as alphabetized relations. As already unveiled in the introduction, we semantically represent alphabetized predicates by sets of extensible records, and the latter by sets of pairs of extensible records; for the latter, there is already the theory Relation .thy in the Isabelle library that provides a collection of derived rules for sequential relational composition _ o _ or operators for least and greatest fixpoints (lfp , gfp).

In order to support a maximum of common UTP look-and-feel, we implement on the SML level implementing Isabelle a function that computes for a term denoting an alphabetized predicate (a cterm in Isabelle terminology) the alphabet of a theorem, be it in the format of an alphabetized predicate or an alphabetized relation. This function is suitably integrated into the command language ISAR of Isabelle such that we can define and query on the ISAR shell:

```
define_pred sample "({x:: int ,x ':: int ,   ...},   x = x' + 1) "
alpha       sample
inalpha     sample
outalpha    sample
```

The first statement will be expanded internally into definitional constructions of an alphabetized predicate; the latter three statements make Isabelle execute the common alphabet projection functions α sample, the input alphabet **in**α sample and the output alphabet outα sample (which are $\{x, x', ...\}$,$\{x, ...\}$ and $\{x', ...\}$, respectively). In more detail, the alphabetized predicate mechanism expands internally the **define_pred**-command as follows:

```
record      sample_type =  x :: int ,   x ':: int
definition  sample "sample ≡{A::sample_type.A.x = A.x' + 1}"
```

where the latter introduces per default a *constant* sample with the right type and a *theorem* sample_def containing the constant definition for sample. Note that leaving out the dots "..." in the **define_pred**-declaration leads to non-extensible records (the internal δ type-variable representing future extensions is instantiated with the trivial unit-type); the query-functions will reflect this in the output accordingly.

Alphabetized Predicates. We introduce the abbreviation α alphabet as a syntactic marker to highlight types that we use for alphabetized elements; on this basis, alphabetized predicates as sets of records are defined as follows:

```
types α alphabet  = "α"
types α predicate  = "α alphabet ⇒ bool"
```

The standard logical connectives on predicates are simply introduced as abbreviations:

```
abbreviation  true :: "α predicate"
where "true = λA. True"

abbreviation  false :: "α predicate"
where "false = λA. False"

abbreviation  not :: "α predicate ⇒ α predicate" ("¬ _")
where "¬ P  = λA. ¬ (P A)"

abbreviation  conj :: "[α predicate, α predicate] ⇒ α predicate" ( infix "∧")
where "P ∧ Q = λA. (P A) ∧ (Q A)"

abbreviation  disj :: "[α predicate, α predicate] ⇒ α predicate" ("_ ∨ _")
where "P ∨ Q = λA. (P A) ∨ (Q A)"

abbreviation impl :: "[α predicate, α predicate] ⇒ α predicate" ("_ ⟶ _")
where "P ⟶ Q = λA. (P A) ⟶ (Q A)"
```

Note that our typing requires that all arguments range over the *same* alphabet. This is a significant restriction compared to textbook UTP, where all alphabets were merged (by the union of the underlying sets), pretty much in the style of Z. Thus, there are implicit coercions between sub-expressions in UTP alphabetized predicates that have to be made explicit in suitable coercion functions. For example, if we have the additional alphabetized predicate:

```
define_pred sample2 "({y:: int,  ...},  y = 5) "
```

an expression like sample ⟶ sample2 is simply ill-typed since they are both built over different alphabets. In order to make this work, it is necessary to insert suitable coercion functions (whose definition will be shown below):

$$(Inj_{\alpha\mathrm{sample}\mapsto\alpha\mathrm{sample}\cup\alpha\mathrm{sample2}}\ \mathrm{sample}) \longrightarrow (Inj_{\alpha\mathrm{sample2}\mapsto\alpha\mathrm{sample}\cup\alpha\mathrm{sample2}}\ \mathrm{sample2})$$

The insertion of such coercion functions can be done automatically (based on an SML- computation of the alphabet of each sub-expression) and in an optimized form (only in cases where the alphabets are not just inclusion, only at the "leaves", i.e. around constants denoting alphabetized predicates). The details of such an automated coercion inference are out of the scope of this paper; the technique, however, has already been applied elsewhere [3].

It remains to define universal and existential quantifications in terms of HOL quantifications.

```
abbreviation ex::"'β ⇒ ['β ⇒'α predicate] ⇒ 'α predicate" ("∃ _ _")
where "∃ x P ≡λA. ∃ x. (P x) A"

abbreviation all ::"'β ⇒ ['β ⇒'α predicate] ⇒ 'α predicate" ("∀ _ _")
where "∀ x P ≡λA. ∀ x. (P x) A"
```

Alphabetized Relations. the alphabetized relations type is defined as a HOL relation over $in\alpha P$ and $out\alpha P$. Some programming constructs are then defined over relations, for example the *conditional expression*. The condition expression is represented as a predicate over $in\alpha P$, the symbols are kept as defined in the UTP book.

```
types α relation  = "(α alphabet ×α alphabet) set"
types α condition  = "α ⇒bool"

abbreviation cond::"[α relation ,αcondition,α relation ]⇒ α relation"  ("_◁ _ ▷_")
where "(P ◁ b ▷ Q) = λ(A, A'). (b A ∧P (A, A')) ∨
                                 (¬ (b A) ∧Q (A, A'))"
```

The second definition concerns the *sequential composition*; we use a predefined HOL relation operator, which is the relation composition. This operator corresponds exactly to the definition of sequential composition of alphabetized relations.

```
abbreviation comp::"α relation ⇒ α relation ⇒ α relation" ("_ ;; _")
where "(P ;; Q) = P o Q"
```

Since the alphabet is defined as an extensible record, an update function is generated automatically for every field. For example, let a be a field in some record, then there is the function r⦇a := x⦈, or represented internally _update_ name a x. We use this internal representation to define by a syntactic paraphrasing the update relation family defined as {(A,A'). A'= ⦇a := E A⦈}.

The syntactic transformation of the assignment to the update function is instrumented as follows:

```
syntax
"_Assign"        :: "[idt , α ⇒ β] ⇒ α relation"      ("_ :== _")
translations
"x :== E"        => "{(A,A'). A' = _update_ name x (E A)}"
```

A last construct is the *Skip* relation, which keeps all the variable values as they were. We use an equality over $in\alpha P$ and $out\alpha P$ to represent this. By using the record type for the alphabet, this equality is considered as values equality.

```
abbreviation skip_r ::"α relation" ("Π_r")
where "Π_r = λ(A, A'). (A' = A)"
```

The notion of *refinement* is equivalent to the universal implication of predicates, it is defined using the universal closure used in the UTP.

```
abbreviation closure :: "α predicate ⇒ bool" ("[_]")
where "[ P ] = ∀ A. P A"

abbreviation refinement :: "[αpredicate,α predicate] ⇒ bool" ("_⊑_")
where "P ⊑Q = [ Q ⟶P ]"
```

Coercions. As mentioned earlier, it is crucial for our approach to generate coercions in order to make our overall approach work. While it is impossible to *define* coercion function once and for all for an arbitrary $\alpha\,P$ inside HOL, it is however possible for any concrete alphabet, say $\{x :: int, x' :: int, ...\}$, a coercion, and compute this concrete alphabet for each UTP theory context outside the logic in suitable parsing functions.

More concretely, we have:

1. $Inj_{A\mapsto B}\,P$ which embeds pointwise. Elements of the P-set with alphabet αP are mapped to elements with identical field content if field $a \in \alpha P$, and with arbitrary values if $a \in A$. For example we consider the case $Inj_{\alpha\text{sample}\mapsto\alpha\text{sample}\cup\alpha\text{sample2}}$ which is just: $Inj_{\{x,x',...\}\mapsto\{x,x',y...\}}$. Then we define it by:

   ```
   λ P.  {σ . P ⦇x:=x σ , x' := x' σ , ... ⦈}
   ```

2. $Proj_A\,P$ projects pointwise. Elements of the P-set with alphabet αP are mapped to elements with identical field content if field $a \in \alpha P$; the fields were ommited otherwise.
3. $Inj_{A\mapsto(B\times C)}\,P$ is a version of $Inj_{A\mapsto B}\,P$ that splits into pairs (useful for the transition between predicates and relations). Example: $Inj_{\alpha\text{sample}\mapsto\alpha in\text{sample}\times\alpha out\text{sample}}$ or concretely:

   ```
   λ P.  {(σ ,σ ') . P ⦇x:=x σ , x' := x σ ', ... ⦈}
   ```

4. $Proj_{A\times B\mapsto A\cup B}\,P$ is the inverse of the latter.
5. etc.

3.2 Designs theory

The *Designs* theory is centered around a new concept which is captured by the extra name ok. Thus, we consider alphabets that contain at least this variable. This fits well to our representation of alphabets in extensible records: any theorem that we prove once and for all in the *Designs* theory will hold in future theories, too.

The name ok. For short, the definition proceeds straightforwardly:

```
define_pred alpha_d "({ok::bool,  ...},  true) "
```

However, it is worthwhile to look at the internal definitions generated here:

```
record alpha_d        = ok::bool
types 'α alphabet_d  = "'α alpha_d_scheme alphabet"
types 'α  relation_d  = "'α alphabet_d  relation "
```

In this construction, we use the internal type synonym alpha_d_scheme which Isabelle introduces internally for the cartesian product format where δ captures the possible type extension.

Since the definition of alphabets and relations uses a polymorphic type, we declare a new alphabet and relation type by instantiating this type to an extensible alpha_d. All the expressions defined for the first, more general type, will be directly applicable to this new specific type.

Designs. Designs are a subclass of relations than can be expressed in the form:

$$(\mathsf{ok} \wedge \mathsf{P}) \rightarrow (\mathsf{ok}' \wedge \mathsf{Q})$$

which means that if a program starts with its precondition P satisfied, it will finish and satisfy its post condition Q. The definition of designs uses the previous definitions of relations and expressions.

```
definition  design :: "[α  relation_d ,  α  relation_d ]  ⇒ α  relation_d "  ("( _ ⊢ _)")
where "  (P ⊢  Q) ≡λ(A, A'). (ok A ∧P (A, A'))  ⟶(ok A' ∧ Q (A ,A'))"
```

As seen above, ok is an automatically generated function over the record type alpha_d, it returns the value of field ok

Once given the definition of designs, new definitions for *skip* are stated as follows:

```
abbreviation skip_d :: "α  relation_d"  ("Π_d")
where "Π_d ≡(true ⊢  Π_r)"
```

Our definitions make it possible to lead some proofs using Isabelle/HOL, as for the *true*−; *left zero* lemma. More details about proofs are given in Sect. 3.5.

3.3 Reactive processes

As for designs, reactive processes require more observational variables to be defined. They are used for modeling the interaction of a process with its environment. Proceeding like we did with ok, we extend the alphabet with the variables wait, tr and ref. The corresponding extended alphabet and the definition of reactive processes are given in our *Reactive Process* theory. This kind of alphabet is called a *reactive alphabet*.

The names wait, tr and ref. The variable wait expresses whether a process has terminated or is waiting for an interaction with its environment. The variable tr records the trace of events (interactions) the process has already performed. The ref variable is an event set, that encodes the events (interactions) that the process may refuse to perform at this state.

The new alphabet is an extension of the alphabet of designs, using the same construct: extensible records. The traces are defined as polymorphic events lists, and the refusals as polymorphic events sets.

```
datatype α event     = ev α
types    α trace     = "(α event) list"
types    α refusals  = "(α event) set"

define_pred alpha_rp
          "(alpha_d ∪ {wait :: bool, tr :: α trace, ref :: α refusals ,...}, true)"
```

and we add the handy type abbreviation:

```
types (α, δ) relation_rp = "(α, δ) alpha_rp_scheme relation"
```

Again, the δ is used to make the record-extensions explicit.

Reactive Processes. Reactive processes are characterised by three healthiness conditions. The first healthiness condition **R1** states that a reactive process cannot change the history of performed event.

$$\mathbf{R1}\ P = P \wedge (tr \leq tr')$$

This healthiness condition is encoded as a relation, it uses a function $\leq$ on traces, which is defined in our theory.

```
abbreviation R1::"(α, δ) relation_rp"
where "R1 (P) ≡ λ (A, A'). P (A, A') ∧ (tr A ≤ tr A')"
```

To express the second healthiness condition **R2**, we use the formulation proposed by Cavalcanti and Woodcock [7].

$$\mathbf{R2}\ (P(tr, tr')) = P(<>, tr - tr')$$

It states that a process description should not rely on what took place before its activation, and should restrict only the new events to be recorded since the last observation. These are the events in $tr - tr'$.

```
abbreviation R2::"(α, δ) relation_rp"
where "R2 (P) ≡ λ (A, A'). P (A(|tr:=[]|),A'(|tr:= (tr A ' − tr A) |))"
```

The last healthiness condition for reactive processes, **R3**, states that a process should not start if invoked in a waiting state.

$$\mathbf{R3}\ (P) = \mathit{\Pi} \triangleleft wait \triangleright P$$

A definition is given to $\mathit{\Pi}$ (Skip process), and the healthiness condition is expressed as a conditional expression over predicates.

```
abbreviation R3::"(α, δ)  relation_rp "
where "R3 (P) ≡(Π_rp ◁(wait  o  fst ) ▷ P)"
```

We can now define a reactive process as a relation over a reactive alphabet that satisfies these three healthiness conditions. This condition can be expressed as a functional composition of the three conditions.

```
definition  R::"(α, δ)  relation_rp "
where "R ≡R3 o R2 o R1"
```

3.4 CSP Processes

As for reactive processes, a theory *CSP Process* corresponds to the CSP processes healthiness conditions. In UTP, a reactive process is a CSP process if it satisfies two additional healthiness conditions **CSP1** and **CSP2**.

```
definition  CSP1::"(α, δ)  relation_rp "
where "CSP1 (P) ≡ λ (A, A'). (P (A, A')) ∨ (¬ ok A ∧tr  A ≤tr  A')"

definition  J_csp ::"(α, δ)  relation_rp "
where "J_csp  ≡  λ (A, A'). ok A ⟶ok A' ∧ tr  A = tr A' ∧ wait  A = wait A'
                                               ∧ ref  A = ref A' ∧ more A = more A' "

definition  CSP2::"(α, δ)  relation_rp "
where "CSP2 (P) ≡ P  ;;  J_csp"
```

CSP basic processes and operators can be encoded using their definitions as reactive designs. Isabelle can be used to prove that these reactive designs are CSP healthy. This could be an extension of our theory, which contains only the definitions of the two CSP healthiness conditions above. There are three other CSP healthiness conditions that we don't mention here. However, they will be considered in the *Circus* theory since they are required for *Circus* processes.

3.5 Proofs

As mentioned above, the theories contains also proofs for some theorems and lemmas. In the relations theory, 100 lemmas are proved using 250 lines of proof, and in the designs theory 26 lemmas are proved using 120 lines of proof. Since our definitions are close to the library definitions of Isabelle/HOL, we can exploit the power of the standard Isabelle proof procedures. For example, we consider the proof of the *true*−; *left zero* lemma. There are almost the same proof steps as those used in the textbook proof.

```
lemma t_comp_lz: "(true;;(P ⊢ Q)) = true"
  apply (auto simp: expand_fun_eq  design_def  rel_comp_def_raw  mem_def)
  apply ( rule_tac  x="b(|ok:=False|)"  in exI)
  by (simp add: mem_def)
```

In the previous proof we first apply some simplifications using the operators definitions (eg. *design_def*), then we fix the ok value to false and finally some simplifications will finish the proof.

4 *Circus*

4.1 A Brief Introduction into the *Circus* Language

Circus is a formal specification and development approach providing a combination of process algebra and model-based abstract data types, with an integrated notion of refinement. As a language, it combines CSP, Z and refinement.

$$\begin{array}{l}
\textbf{channel}\ out : \mathbb{N} \\
\\
\textbf{process}\ Fib \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ FibState == [\, x, y : \mathbb{N} \,] \\
\quad InitFibState == [\, FibState' \mid x' = y' = 1 \,] \\
\quad InitFib \mathrel{\widehat=} out\,!1 \rightarrow out\,!1 \rightarrow InitFibState \\
\quad OutFibState == [\, \Delta FibState;\ next! : \mathbb{N} \mid next! = y' = x + y \land x' = y \,] \\
\quad OutFib \mathrel{\widehat=} \mu\ X \bullet (\,\textbf{var}\ next : \mathbb{N} \bullet OutFibState \,;\, out\,!next \rightarrow X\,) \\
\quad \bullet\ InitFib \ ;\ OutFib \\
\textbf{end}
\end{array}$$

Fig. 1. The Fibonacci suite in *Circus*

Syntactically, a *Circus* specification is a sequence of paragraphs, just like in Z or Isabelle/ISAR, with the possibility to declare schemas, channels and processes. In the example of Fig. 1, there is first a paragraph where a channel is declared, namely $out : \mathbb{N}$. Then comes the definition of the *Fib* process as a sequence of (1) a state definition, which is just a couple of natural numbers, (2) an initialization operation *InitFibState* on the state defined by a Z schema, (3) a *Circus* action named *InitFib* defined as a CSP-like process with the specificity that the *InitFibState* operation appears as an event, (4) a normal operation on the state *OutFibState*, defined by a Z schema and (5) a recursive action *OutFib* defined by CSP-like constructs where *OutFibState* operation appears as an event. Finally, the main action of the *Fib* process is given by just the sequential composition of the two actions above.

This example just shows the description of a process with an encapsulated state, where the behavior combines CSP-like external interactions and Z-like internal state operations. The small example gives only a flavor of *Circus*, which comprises a combined semantics for features like parallelism, internal choices, encapsulated complex data types, imperative statements, and refinements.

4.2 The *Circus* Theory

The denotational semantics of *Circus* was defined by Oliveira et al. [13], based on UTP. *Circus* actions are defined as CSP healthy reactive processes. The Circus_Actions theory contains the definition of the type Action, which restricts the relations to the subset of CSP healthy relations.

```
typedef(Action)
  (α,δ)  action  = "{p::(α,δ) relation_rp . is_CSP_process p}"
proof −
   have "true ∈ {p ::( α,δ) relation_rp . is_CSP_process p}"
        by(auto simp add: Collect_def mem_def Healthy_def)
   thus ?thesis by auto
qed
```

We assume here the predicate is_CSP_process capturing the known healthiness conditions of CSP (not shown here). Isabelle methodology imposes that type definitions should be non-empty. In the action type definition, the first part declares the actions as subset of CSP healthy relations, and the second part is the proof that this subset is not empty.

Every *Circus* operator is defined as an alphabetized predicate. the first definitions concern the basic processes Stop, Skip and Chaos. Some other examples of operators are also shown in the sequel of the paper.

Basic Processes. Stop is defined as a reactive design, with a precondition true and a postcondition stating that the system deadlocks and the traces are not evolving.

```
definition
      Stop :: "(α,δ) action"
where
    "Stop ≡ Abs_Action (R (true ⊢ λ (A, A'). tr A' = tr A ∧wait A'))"
```

Skip is defined as a reactive design, with a precondition *true* and a postcondition stating that the system terminates and all the variables of the state are not changed.

```
definition
      Skip :: "(α,δ) action"
where
    "Skip ≡ Abs_Action (R (true ⊢ λ (A, A'). tr A' = tr A ∧¬ wait A'
                                                   ∧ more A = more A'))"
```

The Chaos process is defined as a reactive design with false as precondition and true as postcondition.

Communications. The prefixed actions definition is based on the definition of a special predicate do_C. In the *Circus* denotational semantics, different forms of prefixing were defined, we define in our theory one general form, and the other notations can be defined using this form.

```
abbreviation
  do_C :: "[α event, α event set] ⇒ (α,δ) relation_rp ''
where
 "do_C x S ≡ (λ (A, A'). (tr A = tr A') ∧ (S ∩ (ref A') = {}))
              ◁ wait ▷
              (λ (A, A'). ∃ e. e ∈ S ∧ (tr A') = (tr A)@[e] ∧ x = e)"
```

The definition of do_C is different from Oliveira et al.'s definition [13], because we want our definition to be more general. The prefixing action can then be defined, using the same denotational semantics definition.

```
definition
     Prefix :: "[α event set, α event ⇒ (α,δ) action] ⇒ (α,δ) action"
where
 "Prefix S P ≡Abs_Action((∃ e.  R (true ⊢ (λ (A,A'). ((do_C e S)(A,A')
                                         ∧ more A' = more A )))) ;; P e)"
```

Different types of communication are considered below. The channels are defined as functions over communicated values. We distinguish three types of communications:

- Inputs: the set of communications contains all possible values.
- Outputs: the set of communications contains only one value.
- Synchronizations: the set is empty, there is just a channel name.

Below, we define these three communications forms

```
definition
 read      :: "[α ⇒ β event,α set, α ⇒ (β, δ) action] ⇒ (β, δ) action"
"read c S P  ≡ Prefix (c ‘ S) (P o (inv c))"
 write     :: "[α ⇒ β event, α, (β, δ) action] ⇒ (β, δ) action"
"write c a P ≡ Prefix {c a} (λ x. P)"
 write0    :: "[β event, (β, δ) action] ⇒ (β, δ) action"
"write0 a P  ≡ Prefix {a} (λ x. P)"
```

and configure the Isabelle syntax-engine such that it parses the usual communication primitives:

```
syntax
"_read"   :: "[id, pttrn, (α,δ) action] ⇒ (α, δ) action" ("_‘?‘_→_")
"_readS"  :: "[id, pttrn, β⇒bool, (α,δ)action] ⇒ (α,δ)action"("_‘?‘_‘:‘_→_")
"_write"  :: "[id, β,(α,δ)action] ⇒  (α,δ) action"  ("_‘!‘_ → _")
"_writeS" :: "[α,(α,δ)action] ⇒ (α,δ)action"  ("_ → _")
translations
"c ‘?‘ p → P"          ≡"CONST read c CONST UNIV (λp. P)"
"c ‘?‘ p ‘:‘ b → P"    ≡"CONST read c {p. b} (λp. P)"
"c ‘!‘ p → P"          ≡"CONST write c p P"
"a → P"                ≡"CONST write0 a P"
```

Guarded Actions. A guarded action is defined with a condition and an action, we define a special function Spec that fixes the values of wait and ok' for a given predicate.

```
abbreviation
  Spec
where
 "Spec b b' P ≡λ (A,A'). P (A(|wait := b'|), A'(|ok := b|))"

definition
  Guard :: "[(α,δ) relation_rp ,(α,δ)action] ⇒ (α,δ)action" ("_ & _")
where
 "g & P ≡ Abs_Action(R ((g → ¬ Spec False False P) ⊢
                          ((g∧Spec True False P) ∨
                           (¬ g ∧ λ (A,A').( tr A'=trA∧ waitA')))))"
```

Sequencing Actions may be composed sequentially using the squential composition operator. The definition is based on the UTP relation composition.

```
definition
  Seq :: "[(α,δ)action ,(α,δ)action] ⇒ (α,δ)action"  ("_‘;‘_")
where
  "P ‘;‘ Q ≡ Abs_Action (Rep_Action P ;; Rep_Action Q)"
```

The complete *Circus* theory contains the definition of all actions operators, constructs, healthiness conditions and the proofs of some theorems over them.

Circus **Processes.** Finally, the *Circus* process definition contains the alphabet declaration, schema expressions and actions. The alphabet is defined by extending the alpha_rp record with the process variables. The normalized schema expressions are defined separately as *relations* over the defined alphabet. The actions are defined as *Circus actions* over the alphabet.

The next section gives an example of how a *Circus* process is written using the previous theories.

5 Example: Using Isabelle/*Circus*

To illustrate the *use* of the *Circus* theory we come back to our example in Fig. 1 of a process that calculates and outputs the Fibonacci suite. The process uses only one channel *out* that communicates natural numbers. The process state is defined by two natural variables x and y. The process contains two schema expressions *InitFibState* and *OutFibState*, and two actions *InitFib* and *OutFib*. *InitFibState* initializes the state variables to the value 1. *InitFib* action outputs twice the value 1 over the channel *out*, then calls *InitFibState*. *OutFibState* performs the Fibonacci suite step, and returns a value *next*. The *OutFib* action recursively calls *OutFibState* and outputs the value of *next*. The main action of the process performs initialization with *InitFib*, then generates the fibonacci suite with *OutFib*.

In the following, we will encode this example in our *Circus* theory. Note that we deliberately refrain from a front-end here that hides the Isabelle/*Circus* internals from the user (such a front-end consisting of the existing CZT-parser and type-checker for Circus will be integrated in the future); the purpose of this section is to have a glance at our *Circus* semantics “at work”.

5.1 Channels and alphabet

We first define the channels, types and alphabets. The state definition corresponds to the extension part of the defined alphabet.

```
datatype channel = out nat

record  fib_state  = "channel alpha_rp" +
                      x   :: nat
                      y   :: nat

types my_alpha = " fib_state  alphabet"
types my_pred = "my_alpha relation"

types my_action = "(channel, ⦇x :: nat,  y :: nat⦈)  action"
```

5.2 Schema expressions and actions

Normalized schema expressions are defined as reactive processes. The predicate value corresponds to the schema formula, and the input/output variables are passed as parameters (eg. **next**). The actions are defined as *Circus actions*.

```
definition
      InitFibState :: "my_pred"
where
    " InitFibState  ≡R (λ(A, A'). (x  A' = 1 ∧y  A' = 1))"

definition
      InitFib   ::  "my_action"
where
    " InitFib  ≡(out '!' 1 → (out '!' 1 → Abs_Action  InitFibState ))"

definition
      OutFibState::"nat ⇒ my_pred"
where
    "OutFibState next ≡R (λ(A, A'). x  A' = y A
                                    ∧ y  A' = x A + y A ∧next = x A + y A)"
```

```
definition
  OutFib :: "my_action"
where
  "OutFib ≡μ X. Abs_Action (λ A. ∃ next.
               ((OutFibState next) ;; (Rep_Action (out ‘!‘ next → X))) A)"
```

5.3 Main action

The main action is also defined as a *Circus Action*, by a sequential composition of the defined actions InitFib and OutFib.

```
definition
  Fib :: "my_action"
where
  "Fib ≡ InitFib ‘;‘ OutFib"
```

6 Conclusions and Future Work

This paper introduces the Isabelle/*Circus* proof environment. It is conceived as a shallow embedding into Isabelle/HOL and aims for effective deductive support of those UTP theories related to *Circus* and to *Circus* itself. This is work in progress: while the foundation of the UTP is done and most textbook proofs have been formalized, there is at present still very little automated proof support with respect to our ultimate goal, the development of efficient deductive support for verification and test generation for *Circus*.

Our choice of Isabelle as a foundation is justified by the fact that this proof environment comes with a rich set of deduction machinery, and the powerful binding mechanism and type inference system of HOL that we can re-use. However, our main motivation is that we plan to use the HOL-TestGen system [4, 5], that is developed on the top of Isabelle/HOL, for developing well-founded test generation strategies from *Circus* specifications, on similar formal bases as those presented by Cavalcanti and Gaudel [8] for CSP.

When developing these theories, it turned out that using HOL extensible records for representing alphabetized predicates is extremely convenient and allows an incremental encoding that remains very close to the original UTP definitions.

Moreover, a significant advantage is that we do not encode the alphabet in our key formalization of the "alphabetized predicate": after a pre-processing, we do everything in the semantic representation with the *type* (as in HOL-Z). This means

- that P is not of type boolean ("predicate"), but of form $\alpha => bool$, which is equivalent to $\alpha\ set$.

- for α, we uses the fact that the fields of the extensible records correspond to the elements of the alphabet,
- the function $\alpha\ P$ becomes a meta-function in ML.

The price to pay are a number of coercions that we have currently to add by hand (and that might be generated by a future front-end using the CZT-Parser, in the way the "Encoder" works in HOL-Z).

From prior experiments, for instance the HOL-Z system mentioned above, we expect this approach to lead to the deductive efficiency required to support proofs about *Circus* specifications, to apply verified transformations on them, with the objective of assisting refinement and test generation.

References

1. Andrews, P.B.: Introduction to Mathematical Logic and Type Theory: To Truth through Proof. 2nd edn. (2002)
2. Arthan, R.: The ProofPower homepage (2009), http://www.lemma-one.com/ProofPower/index/
3. Brucker, A.D., Rittinger, F., Wolff, B.: Hol-z 2.0: A proof environment for z-specifications. Journal of Universal Computer Science 9(2), 152–172 (Feb 2003)
4. Brucker, A.D., Wolff, B.: Symbolic test case generation for primitive recursive functions. In: Grabowski, J., Nielsen, B. (eds.) FATES. Lecture Notes in Computer Science, vol. 3395, pp. 16–32. Springer (2004)
5. Brucker, A.D., Wolff, B.: Test-sequence generation with hol-testgen with an application to firewall testing. In: Gurevich, Y., Meyer, B. (eds.) TAP. Lecture Notes in Computer Science, vol. 4454, pp. 149–168. Springer (2007)
6. Brucker, A.D., Wolff, B.: An extensible encoding of object-oriented data models in hol with an application to imp++. Journal of Automated Reasoning (JAR) 41(3–4), 219–249 (2008), serge Autexier, Heiko Mantel, Stephan Merz, and Tobias Nipkow (eds)
7. Cavalcanti, A.L.C., Woodcock, J.C.P.: A Tutorial Introduction to CSP in Unifying Theories of Programming. In: Refinement Techniques in Software Engineering. Lecture Notes in Computer Science, vol. 3167, pp. 220 – 268. Springer-Verlag (2006)
8. Cavalcanti, A., Gaudel, M.C.: A note on traces refinement and the *conf* relation in the Unifying Theories of Programming. In: Butterfield, A. (ed.) Unifying Theories of Programming, Second International Symposium, UTP 2008, Trinity College, Dublin, Ireland, September 8-10, 2008, Revised Selected Papers. Lecture Notes in Computer Science, vol. 5713, pp. 42–61. Springer (2008)
9. Church, A.: A formulation of the simple theory of types 5(2), 56–68 (Jun 1940)
10. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice Hall International Series in Computer Science (1998)
11. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/**hol!**—A Proof Assistant for Higher-Order Logic, vol. 2283 (2002)
12. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: Unifying theories in ProofPower-Z. In: Dunne, S., Stoddart, B. (eds.) UTP 2006: First International Symposium on Unifying Theories of Programming. LNCS, vol. **4010**, pp. 123–140. Springer-Verlag (2006)

13. Oliveira, M., Cavalcanti, A., Woodcock, J.: A denotational semantics for Circus. Electron. Notes Theor. Comput. Sci. 187, 107–123 (2007)
14. Zeyda, F., Cavalcanti, A.: Encoding Circus programs in ProofPowerZ. In: Unifying Theories of Programming, Second International Symposium, UTP 2008, Trinity College, Dublin, Ireland, September 8-10, 2008, Revised Selected Papers. Lecture Notes in Computer Science, vol. 5713. Springer-Verlag (2009), http://www.cs.york.ac.uk/circus/publications/docs/zc09b.pdf
15. Zeyda, F., Cavalcanti, A.: Mechanical reasoning about families of UTP theories. Science of Computer Programming (2010), in Press, Corrected Proof, Available online 17 March 2010