UTPCalc — A calculator for UTP Predicates

Andrew Butterfield *

Trinity College Dublin

Abstract. We present the development of the UTP-Calculator: a tool, written in Haskell, that supports rapid prototyping of new theories in the Unifying Theories of Programming paradigm, by supporting an easy way to very quickly perform test calculations. The emphasis during the calculator development was keeping it simple but effective, and relying on the user to have the expertise to check its output. It is not intended to supplant existing theorem prover or language transformation technology. The tool is designed for someone who is both very familiar with UTP theory construction, and familiar enough with Haskell to be able to write pattern-matching code. In this paper we describe how this tool can be used to assist in theory development, by describing the key components of the calculator and how various aspects of such a theory might be encoded. We finish with a discussion of our experience in using the tool.

1 Introduction

1.1 Motivation

The development of a Unifying Theory of Programming (UTP) can involve a number of false starts, as alphabet variables are chosen and semantics and healthiness conditions are defined. Typically, some calculations done to check that everything works end up revealing problems with the theory. So an iteration occurs by revising the basic definitions, and attempting the calculations again.

We have recently started to explore using UTP to describe shared-variable concurrency, by adapting the work of the UTP semantics for Unifying Theories of Parallel Programming (UTPP) [20]. We have reworked it to use a system for generating unique labels, in order to give a slight improvement to the compositionality of the semantics. This we call a Unifying Theory of Concurrent Programming (UTCP)[6].

We illustrate the calculator here with, as a running example, the definition of the UTP semantics of an atomic action. We assume basic atomic actions A, B which modify the shared global state (program variables), represented by observation variables s, s'. The concurrent flow of control is managed by using labels associated with language constructs, which are added to and removed from a global label-set as execution proceeds. We represent this label-set using

* This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie)

observations ls, ls'. Our main change to the original UTPP theory is to provide a mechanism to create unique labels, to be associated with both the beginning and end of each language construct. This results in three static observables: a generator g; and two labels *in* and *out*. So our UTCP theory is based on a non-homogeneous relation with alphabet s, s', ls, ls', g, *in*, and *out*. See [6] for details.

Our running example is the need to calculate the outcome of sequentially composing (;;) two basic atomic actions (A, B), that are lifted $(\mathbf{A}(\_))$ to the full alphabet by adding control-flow, and are then *run* in order to see their dynamic behaviour:

$$run(\mathbf{A}(A) \;;;\; \mathbf{A}(B))$$

We hope that the final result would be

$$(A \; ; \; B) \land ls' = \{\ell_{g:}\}$$

We have the standard UTP sequential composition of A and B defined on s, s', and an assertion that the termination label $\ell_{g:}$ is the sole member of the final label-set.

The theory with its somewhat unusual arrangement of observation variables did *not* emerge as an immediate and obvious solution, but as a result of many trial calculations. These trial calculations exposed two problems: one was the difficulty in reading very long complex set-based expressions in order to assess their correctness. The second was the sheer length and drudgery of these calculations, often involving many repetitions of very similar steps.

To be specific, the calculator described in this paper is intended to be used for *calculation*, and not *theorem proving*. In particular, it is designed to help solve the problem just described above. Both the starting predicate and the final result have free variables and are not theorems. That means counter-example generators like Nitpick or Alloy are not helpful.

If we consider the reasoning processes used in the development and deployment of a theory, we can see a spectrum ranging from informal, through to fully mechanised: hand calculation; simulation; proof assistant; and automated theorem provers. The level of detail, complexity, and rigour rises as we proceed along the spectrum. The calculator described here is designed to assist with the exploratory hand-calculation phase early on, by making it easier to calculate, and to manually check the outcomes. It is not intended to provide the soundness guarantees that are quite rightly expected from the tools further along the spectrum.

1.2 Structure of this paper

In Sect. 2 we discuss related work to justify our decision to develop the calculator. The key design decisions and tool architecture is then described in Sect. 3. Three key components of the system are then discussed: Dictionaries (Sect. 4); Expressions (Sect. 5); and Predicates (Sect. 6). In Sect. 7 we describe how to encode laws and then conclude (Sect. 8).

2 Related Work

There are lots of tools for assisting with the kinds of calculations we are trying to perform, ranging from calculators [3], through rewrite/transformation systems (CIP-S[1], Stratego[16], ASF+DSF[15] Maude[7] HATS[19]) to full-blown theorem provers (Isabelle/HOL[12], CoQ[2], PVS[14]) including those that support equational reasoning (Isabelle/ISAR[18]).

Most of the above have a considerable body of work behind them, both in terms of theory and tool development, and provide very comprehensive coverage of their problem domain, be it rewriting, program transformation or theorem proving. However many are tied to specific languages, or have limited ability to allow the user to customise the target language. In particular, it is not clear in any of them, how to achieve the ability to do rapid calculation with a high degree of ease in proof-reading its output.

Within the UTP community, there has been considerable work using ProofPower-Z, such as the deep embedding into Z of an imperative language whose semantics were given using UTP [13] and re-working the mechanisation of UTP in order to better support the hierarchical nature of UTP theory building [21]. There is also work on embedding UTP into Isabelle/HOL[9]. This contains a considerable amount of infrastructure to support UTP's alphabetised predicates in a general way, with UTP forming a third sub-syntax in addition to Isabelle/HOL's inner and outer syntaxes. It continues to undergo continuous improvement[8].

Like all high-quality state-of-the-art tools, CoQ, Isabelle/HOL, ProofPower-Z and PVS all have in common that they work best when used in the manner for which they were designed—in none of these cases does this manner match the way we wish to work in UTP, as described in the introduction, without at least a steep learning curve.

We briefly considered using the $U \cdot (TP)^2$ theorem prover [4,5], which does support both equational reasoning, plus a mode in which calculations can be from a starting predicate, as we require. However, it would have required a lot of setup effort, in particular to build the support theories about sets and labels and generators. Also, it is currently not in an ideal state, due to difficulties installing the relevant third-party software libraries on more recent versions of Haskell.

However, as part of other ongoing work, we had developed a parser and some initial analysis tools in Haskell[11], and this software contained abstract syntax and support for general predicates. It became really obvious that this could be quickly adapted, to mechanise the checking calculations, that were being performed during each attempt. In particular, the key inspiration was the observation, that the pattern of each calculation was very uniform and similar. So a decision was taken to construct the calculator described in this paper. It also has the advantage that it runs on standard Haskell, and hence it is much easier to future-proof.

3 Design & Architecture

3.1 Key Design Decisions

Taking into account the repetitive nature of the calculations, and the need for shorthand notations we very rapidly converged on four initial design decisions:

1. All calculation objects are written directly in Haskell, to avoid having to implement a parser.
2. The expression and predicate datatype declarations would be very simple, with only equality being singled out.
3. Provide a good way to pretty-print long predicates that made it easy to see their overall structure.
4. Rely on a dictionary based system to make it easy to customise how specific constructs were to be handled.

From our experience with the $U \cdot (TP)^2$ theorem-prover we also decided the following regarding the calculation steps that would be supported:

- We would not support full propositional calculus or theories of numbers or sets. Instead we would support the use of hard-coded relevant laws, typically derived from a handwritten proof.
- We would avoid, at all costs, any use of quantifiers or binding constructs.
- The calculator user interface would be very simple, supporting a few high level commands such as "simplify" or "reduce". In particular, no facility would be provided for the user to identify the relevant sub-part of the current goal to which any operation should be applied. Instead the tool would use a systematic sweep through the predicate to find the first applicable calculation step of the requested kind, and apply that. Our subsequent experience with the calculator indicates that this was a good choice.

3.2 The Calculator REPL

The way the calculator is designed to be used is that a function implementing a calculator Read-Execute-Print-Loop (REPL) is given a dictionary and starting predicate as inputs. Calculator commands include an ability to undo previous steps ('`u`'), request help ('`?`'), and to signal an exit from the calculator ('`x`'). However, of most interest are the five calculation commands. The first is a global simplify command ('`s`'), that scans the entire predicate from the bottom-up looking for simplifiers for each composite and applying them. Simplifiers are captured as `eval` or `prsimp` components in dictionary entries.

The other four commands work by searching top-down, left-to-right for the first sub-component for which the relevant dictionary calculator function returns a changed result. Here is where we have a reduced degree of control, which simplifies the REPL dramatically, but has turned out to be effective in practice.

Here is a sample run obtained when calculating the effect of $run(\mathbf{A}(A) \;;; \mathbf{A}(B))$, from the introduction. For convenience we predefined the predicate $\mathbf{A}(A) \;;; \mathbf{A}(B)$ in Haskell as

```
athenb = pseq [patm (PVar "A"),patm (PVar "B")]
```

Here `PVar` is a constructor of the predicate datatype `Pred` (See §6), while `pseq` and `patm` are convenient functions we wrote to build instances of ;; and **A**(_) respectively. We invoked the calculator as follows, where `dictUTCP` is the dictionary for this theory:

```
calcREPL dictUTCP (run athenb)
```

We then proceeded to interact (he prompt " `?,d,r,l,s,c,u,x :-`" shows the available commands):

```
1  run(A(A) ;; A(B))
2   ?,d,r,l,s,c,u,x :- d
3   = "defn. of run.3"
4      (A(A) ;; A(B))[g::,lg,lg,lg:/g,in,ls,out]
5   ; ~ls(lg:) * (A(A) ;; A(B))[g::,lg,lg:/g,in,out]
6   ?,d,r,l,s,c,u,x :- d
7   = "defn. of ;;"
8      (A(A)[g:1,lg/g,out] \/
9      A(B)[g:2,lg/g,in])[g::,lg,lg,lg:/g,in,ls,out]
10  ; ~ls(lg:) * (A(A) ;; A(B))[g::,lg,lg:/g,in,out]
11  ?,d,r,l,s,c,u,x :- s
12  = "simplify"
13     A(A)[g:::1,lg,lg,lg::/g,in,ls,out] \/
14     A(B)[g:::2,lg::,lg,lg:/g,in,ls,out]
15  ; ~ls(lg:) * (A(A) ;; A(B))[g::,lg,lg:/g,in,out]
16 .... 10 more steps
17 A /\ ls' = {lg::} ; B /\ ls' = {lg:}
18  ?,d,r,l,s,c,u,x :- r
19  = "ls'-cleanup"
20 (A ; B) /\ ls' = {lg:}
```

Lines 2, 6, 11, and 18 show the user entering a single key command at the prompt. Lines 3, 7, and 19 show a short string identifying the relevant definition or law. Lines 1, 4–5, 8–10, 17 and 20 show various stages of the calculation.

3.3 Pretty-Printing

For the calculator output, it is very important that it be readable, as many of the predicates get very large, particularly at intermediate points of the calculation. For this reason, a lot of effort was put into the development of both good pretty-printing, and ways to highlight old and new parts of predicates as changes are made. The key principle was to ensure that whenever a predicate had to split over multiple lines, that the breaks are always around the top-most operator or composition symbol, with sub-components indented in, both after and *before*. An example of such pretty-printing in action is shown in Fig.1 The top-level

```
   D(out)
\/ (      ~ls(out)
      /\ (D(lg) \/ A(in,lg,a,in,lg,lg) \/ D(out) \/ A(lg,out,b,lg,out,out))
    ; W(D(lg) \/ A(in,lg,a,in,lg,lg) \/ D(out) \/ A(lg,out,b,lg,out,out)))
```

Fig. 1. Pretty-printer output

structure of this is $D(out) \vee ((\neg ls(out) \wedge \dots); W(\dots))$ where the precedence ordering from tightest to loosest is $\wedge, ;, \vee$.
The pretty printing support can be found in `PrettyPrint.lhs`, which was written from scratch, but inspired by writings of Hughes[10] and Wadler[17] on the subject. In addition to the layout management aspects of pretty-printing there is also a need for a support for shorthand notations. We illustrate this in Sect.5.

Display Convention In the rest of this paper, code that is part of the underlying calculator infrastructure is shown as a simple verbatim display, thus:

```
underlying UTPCalc code
```

while code supplied by the user to set it up for a particular theory under investigation is shown enclosed in horizontal lines:

```
user-supplied theory customisation code
```

4 Dictionaries

The approach taken is to provide a dictionary that maps names to entries that supply extra information. The names can be those of expression or predicate composites, or correspond to variables, and a few other features of note. All of the main calculator functions are driven by this dictionary, and the correct definition of dictionary entries is the primary way for users to set up calculations. The dictionary datatype (`Dict s`), parameterised with a generic type `s`, is critical to the functioning of the calculator.

```
type Dict s = M.Map String (Entry s)
-- M is the renamed import of Data.Map
```

It is the basic way in which the user of the calculator describes the alphabet, definitions and laws associated with their theory.

The dictionary uses the Haskell `String` datatype for keys, and contains four different kinds of entries: alphabets, expressions, predicates and laws.

```
data Entry s = AlfEntry .. | ExprEntry .. | PredEntry .. | LawEntry ..
```

For simplicity, there is only one namespace involved, and some names are reserved for special purposes. These are listed in Fig. 2. There are ten names that

```
Alf Obs Obs' Mdl Mdl' Scr Scr' Dyn Dyn' Stc laws
```

Fig. 2. Reserved Dictionary Names

```
data Expr s
  = St s | B Bool | Z Int | Var String
  | App String [Expr s] | Sub (Expr s) (Substn s) | Undef
  deriving (Eq, Ord, Show)
type Substn s = [(String,Expr s)]
```

Fig. 3. Expression and Substitution Datatypes (`CalcTypes.lhs`)

describe different (overlapping) parts of the theory alphabet (Fig. 2). While it is possible to define these individually, this can be quite error-prone, so a function is provided to construct all these entries from three basic pieces of information: program variable names (‘script’, `Scr`); auxiliary variable names (‘model’,`Mdl`), e.g. variables like *ls* that don’t represent variable values, but instead some other observable program property of interest); and static parameter variable names (`Stc`).

```
stdAlfDictGen :: [String] -> [String] -> [String] -> Dict s
```

All lists contain undashed names, with dashes added when required by the function. So, the alphabet entries for the UTCP theory are defined as follows:

```
dictAlpha = stdAlfDictGen ["s"] ["ls"] ["g","in","out"]
```

All of these entries will be of kind `AlfEntry`, i.e, just lists of the relevant variables.

```
   AlfEntry { avars  :: [String]}
```

There are two important utility functions, one that builds dictionaries from lists of string/entry pairs, and another that merges two dictionaries, resolving conflicts by merging entries if possible, otherwise favouring the second dictionary:

```
makeDict :: [(String, Entry s)] -> Dict s
dictMrg :: Dict s -> Dict s -> Dict s
```

5 Expressions

In Fig. 3 we show the Haskell declarations of the datatypes used to represent expressions and substitution. Both types are parameterised on a generic state type `s`, which allows us to be able to reason independently of any particular notion of state. We provide booleans (`B`), integers (`Z`), values of the generic state type (`St`), and named function application (`App`). We also have substitution (`Sub`), which pairs an expression with a substitution (`Substn`), which is a list of variable/expression pairs. The `deriving` clause for `Expr` enables the Haskell default notions of equality, ordering and display for the type.

5.1 Set Expressions

We shall explore the use of the Expr datatype by indicating how the notions of sets and some basic operators could be defined with the calculator. We shall represent sets as instances of App with the name "set", and the subset relation as an App with name "subset", so the set $\{1, 2\}$ and predicate $S \subseteq T$ would be represented by App "set" [Z 1,Z 2], and App "subset" [Var "S",Var "T"] respectively. In practice, we would define constructor functions to build these:

```
set es = App "set" es
subset s1 s2 = App "subset" [s1,s2]
```

There is a standard interface for defining expression simplifiers: define a function with the following type:

```
Dict s -> [Expr s] -> (String, Expr s)
```

The first argument, of type Dict, is the dictionary currently in use. The second argument is the list of sub-expressions of the App construct for which the simplifier is intended. The result is a pair consisting of a string and an expression. If the simplification succeeds, then the string is non-empty and gives some indication for the user of what was simplified. In this case the expression component is the simplified result. If the simplification has no effect, then the string is empty, and the expression returned is not defined.

The following code defines a simplifier for "subset", which expects it to have precisely two set components:

```
evalSubset d [App "set" s1,App "set" s2] = dosubset d s1 s2
evalSubset _ _ = none -- predefined shorthand for ("",Undef)
```

The two underscores in the second line are pattern matching wildcards, so this catches all other possibilities. It makes use of the following helper, which gets the two lists of expressions associated with each set:

```
dosubset d es1 es2 -- is es1 a subset of es2 ?
  | null (es1 \\ es2)  =  ( "subset", B True )
  | all (isGround d) ((es1 \\ es2) ++ es2)
                       =  ( "subset", B False )
  | otherwise          =  none
```

If the result of removing es2 from es1 is null it then returns true. If not, then if all elements remaining are "ground", i.e., contain no variables, it returns false. Otherwise, we cannot infer anything, so return none.

5.2 Rendering Expressions

The UTCP theory definitions and calculations involve a lot of reasoning about sets, leading to quite complicated expressions. To avoid complex set expressions that are hard to parse visually, a number of simplifying notations are desirable, so that a singleton set $\{x\}$ is rendered as x and the very common idiom $S \subseteq ls$ is rendered as $ls(S)$, so that for example, $ls(in)$ is short for $\{in\} \subseteq ls$. This shrinks the expressions to a much more readable form, mainly by reducing the number of infix operators and set brackets.

When rendering expressions, if an `App` construct is found, then its name is looked up in the dictionary. If an `ExprEntry` is not found, then the default rendering is used, in which `App "f" [e1,e2,..,en]` is converted into `f(e1,e2,..,en)`. Otherwise, a function of type `Dict s -> [Expr s] -> String`, in that entry, is used to render the construct.

As far as expressions are concerned, they become strings, and so are viewed as atomic by the predicate pretty-printer (see Sect. 6). So, we could show singleton sets without enclosing braces by defining:

```
showSet d [elm] = edshow d elm    -- drop {,} from singleton
showSet d elms = "{" ++ dlshow d "," elms ++ "}"
```

Here `edshow` (expression-dict-show) displays its `elm` argument, while `dlshow` (dictionary-list-show) displays the expressions in `elms` separated by the `","` string. Similar tricks are used to code a very compact rendering of a mechanism that involves unique label generator expressions that involve very deep nesting, such as:

$$\pi_2(new(\pi_1(new(\pi_2(split(\pi_1(new(g))))))))$$

This can be displayed as `lg:2:`, using a very compact shorthand described in [6] which we do not explain here.

5.3 Expression Equality

In contrast to the way that the subset predicate is captured as an expression above, the notion of expression equality is hardwired in, as part of the predicate abstract syntax (see Sect. 6). The simplifier will look at the two expression arguments of that construct, and if they are both instances of `App` with the same name, will do a dictionary lookup, to see if there is an entry, from which an equality checking function can be obtained (`isEqual` component). This has the following signature:

```
Dict s -> [Expr s] -> [Expr s] -> Maybe Bool
```

The `Maybe` type constructor is standard Haskell, defined as

```
data Maybe t = Nothing | Just t
```

It converts a type `t` into one which is now “optional”, or equivalently has a undefined value added.

The equality testing function takes a dictionary and the two expression lists from the two `App` instances and either returns `Nothing`, if it cannot establish the truth or falsity of the equality, or `Just` the appropriate result. Suitable code for `"set"` is the following

```
eqSet d es1 es2
 = let ns1 = nub (sort es1)
       ns2 = nub (sort es2)
   in if all (isGround d) (ns1++ns2)
      then Just (ns1==ns2) else Nothing
```

The standard function nub removes duplicates, which we do after we sort. If both lists are ground we just do an equality comparison and return Just it. Otherwise, we return Nothing.

5.4 The Expression Entry

The dictionary entry for expressions has the following form:

```
ExprEntry
    { ecansub :: [String]
    , eprint :: Dict s -> [Expr s] -> String
    , eval :: Dict s -> [Expr s] -> (String, Expr s)
    , isEqual :: Dict s -> [Expr s] -> [Expr s] -> Maybe Bool}
```

One big win in using a functional language like Haskell, in which functions are first class data values, is that we can easily define datatypes that contain function-valued components. We make full use of this in three of the entry kinds, for expressions, predicates and laws.

The eprint, eval and isEqual components correspond to the various examples we have seen above. The ecansub component indicates those variables occurring in the App expression list for which it is safe to replace in substitutions.

To understand the need for ecansub, consider the following shorthand definition for an expression:

$$D(L) \mathrel{\widehat{=}} L \subseteq ls$$

in a context where we know that L is a set expression defined only over variables g, in and out. The variable ls is not free in the lhs, but does occur in the rhs. A substitution of the form $[E/ls]$ say, would leave the lhs unchanged, but alter the rhs to $L \subseteq E$. For this reason the entry for D would need to disallow substitution for ls. The ecansub entry lists the variables for which substitution is safe with the expression as-is. With the definition above, the value of this entry should be ["g","in","out"]. If we want to state that any substitution is safe, then we use the “wildcard” form: ["*"]. We choose to list the substitutable variables rather than those that are non-substitutable, because the former is always easy to determine, whereas the latter can be very open ended.

Given all of the above, we can define dictionary entries for set and subset as

```
setUTCPDict = makeDict
 [ ("set",(ExprEntry ["*"] showSet evalSet eqSet))
 , ("subset",(ExprEntry ["*"] showSubSet evalSubset noEq)) ]
```

Here noEq is an equality test function that always returns Nothing.

```
data Pred s = T | F | PVar String | Equal (Expr s) (Expr s) | Atm (Expr s)
            | Comp String [Pred s] | PSub (Pred s) (Substn s)
```

Fig. 4. Predicate Datatype (`CalcTypes.lhs`)

6 Predicates

In Fig. 4 we show the Haskell declarations of the datatypes to represent predicates. Similar to expressions we have basic values such as true (`T`) and false (`F`), with predicate-valued variables (`PVar`), and composite predicates (`Comp`) which are the predicate equivalent of `App` (see Sect. 5). We also have two ways to turn expressions into predicates. One (`Atm`) lifts an expression, which should be boolean-valued into an (atomic) predicate, while the other is an explicit representation (`Equal`) for expression equality. We can also substitute over predicates (`PSub`).

In many ways, we define our predicates of interest in much the same was as done for expressions. Basic logic features such as negation, conjunction, etc., are not built in, but have to be implemented using `Comp`. A collection of these are pre-defined as part of the calculator, in the Haskell module `StdPredicates`.

There are a few ways in which the treatment of predicates differ from expressions:

- The simplifier and some of the infrastructure for handling laws treats `PVar` in a special way. It is possible to associate an `AlfEntry` in the dictionary with a `PVar`, so defining its alphabet. This can be useful when reasoning about atomic state-change actions which only depend on s and s'. Such entries will be looked up when certain side-conditions are being checked.
- We distinguish between having a definition/expansion associated with a `Comp`, and having a way to simplify one.
- Rendering predicates involves the pretty printer so the interface is more complex. We explain this below.

6.1 Coding Atomic Semantics

Formally, using our shorthand notations, we define atomic behaviour as in UTCP as:

$$\mathbf{A}(A) \mathrel{\widehat{=}} ls(in) \land A \land ls' = ls \ominus (in, out)$$

where A and $\mathbf{A}(\_)$ are as in the introduction, and $S \ominus (T, V)$ is notation from [20] that stands for $(S \setminus T) \cup V$.

Coding a Definition We want to define a composite, called "A" (representing **A**). We define a function that takes a single predicate argument and applies **A** to it

```
patm pr = Comp "A" [pr] -- we assume pr has only s, s' free
```

We can now code up its definition, which takes a dictionary, and a list of its sub-components and returns a string/predicate pair, interpreted in the same manner as the string/expression pair returned by the expression simplifier.

One way to code this is as follows. First define our variables and expressions, because these get used in a variety of places.

```
ls = Var "ls" ; ls' = Var "ls'"
inp = Var "in" -- 'in' is a Haskell keyword
out = Var "out"
lsinout = App "sswap" [ls,inp,out]
```

Here, `"sswap"` is our name for $\ominus$, and note that Haskell identifiers can contain the prime (`'`) character. We then define our atomic predicates ($ls(in)$ and $ls' = ls \ominus (in, out)$)

```
lsin = Atm (App "subset" [inp,ls])
ls'eqlsinout = Equal ls' lsinout
```

Finally we can define $\mathbf{A}(a)$ as their conjunction, where `mkAnd` is a smart constructor for `Comp "And"`, defined in `StdPredicates.lhs`.

```
defnAtomic d [a] = Just ("A",mkAnd [lsin,a,ls'eqlsinout],True)
```

Coding for Pretty Printing For rendering `Comp` predicates, we are going to generate an instance of the pretty-printer type `PP`, using a dictionary and list of sub-predicates, with two additional arguments: one of type `SubCompPrint` which is a function to render sub-components, and one of type `Int` which gives a precedence level. The type signature is

```
SubCompPrint s -> Dict s -> Int -> [Pred s] -> PP
```

The function type is

```
type SubCompPrint s  =  Int -> Int -> Pred s -> PP
```

It takes two integer arguments to begin. The first is the precedence level to be used to render the sub-component, while the second should denote the position of the sub-component in the sub-component list, counting from 1. The third argument is the sub-predicate to be printed. To render our atomic construct we can define the pretty-printer as follows:

```
ppPAtm sCP d p [pr]
       = pplist [ ppa "A" , ppbracket "(" (sCP 0 1 pr) ")"]
```

The functions `pplist`, `ppa` and `ppbracket` build instances of `PP` that respectively represent lists of `PP`, atomic strings, and an occurrence of `PP` surrounded by the designated brackets. Note that the `SubCompPrint` argument (`sCP`) is applied to `pr`, with the precedence set to zero as it is bracketed, and the sub-component number set to one as `pr` is the first (and only) sub-component. We will show how the pretty-printing for sequential composition (;;) in UTCP is defined, to illustrate the support for infix notation.

```
ppPSeq sCP d  p [pr1,pr2]
 = paren p precPSeq
     ( ppopen  (pad ";;") [ sCP precPSeq 1 pr1
                          , sCP precPSeq 2 pr2 ] )
```

Here `pad` puts spaces around its argument, and so its use here is equivalent to `ppa " ;; "`, while `ppopen` uses its first argument as a separator between all the elements of its second list argument. The `paren` function takes two precedence values, and a `PP` value, and puts parentheses around it if the first precedence number is greater than the second. The variable `precPSeq` is the precedence level of sequential composition, here defined to be tighter than disjunction, but looser than conjunction, as defined in module `StdPrecedences`. Note, once more, the use of `sCP`, and how the 2nd integer argument corresponds to the position of the sub-predicate involved.

The Predicate Entry

The dictionary entry for predicates has the following form:

```
PredEntry
 { pcansub :: [String]
 , pprint  :: SubCompPrint s -> Dict s -> Int -> [Pred s] -> PP
 , alfa :: [String], pdefn   :: Rewrite s, prsimp  :: Rewrite s}
type Rewrite s   =  Dict s -> [Pred s] -> RWResult s
type RWResult s  =  Maybe ( String, Pred s, Bool )
```

Fields `pcansub` and `prsimp` are the predicate analogues of `ecansub` and `eval` in the expression entry. Here `pprint` plays the same role as `eprint`, but is oriented towards pretty printing. The `alfa` component allows a specific alphabet to be associated with a composite —if empty then the dictionary alphabet entries apply.

The `pdefn` component, of the same type as `prsimp`, is used when the user invokes the Definition Expansion command from the REPL. The calculator searches top-down, left-right for the first `Comp` whose `pdefn` function returns a changed outcome.

A `RWResult` can be `Nothing`, in which case this definition expansion or simplifier was unable to make any changes. If it was able to change its target then it returns `Just(reason,newPred,isTopLevel)`. The string `reason` is used to display the justification for the calculation step to the user. The `isTopLevel` flag is a hint to the change-highlighting facilities of the pretty-printer infrastructure.

The dictionary entry for our atomic semantics is then:

```
patmEntry=("A",PredEntry [] ppPAtm [] defnAtomic (pNoChg "A"))
```

The function `pNoChg` creates a simplifier that returns `Nothing`.

7 Laws

In addition to the global simplifier and definition expansion facility, we have three broad classes of laws that can be invoked from the REPL: Reduce; Conditional Reduce; and Loop Unroll.

The way the latter three laws are applied is somewhat different to the behaviour of either the simplifier or definition expansion. Instead the reserved dictionary key `"laws"` is used to lookup a special dictionary entry

```
 LawEntry { reduce  :: [RWFun s]
          , creduce :: [CRWFun s], unroll  :: [String -> RWFun s] }
```

7.1 Reduce

The `reduce` component of the `LawEntry` is a list of `RWFun`, which are defined as follows:

```
type RWFun s = Dict s -> Pred s -> Pred s -> RWResult s
```

The first predicate argument is used to supply an invariant assertion for those reduction rules that require one. It is a recent new feature of the calculator, not required for this UTCP theory, and its use is beyond the scope of this paper.

When asked to do a reduce, the calculator then does a top-down, left-to-right search, where at each point it tries all the laws in its reduce list, in order, with the current composite being passed in as the second predicate argument. It terminates at the point of first success (a non-`Nothing` outcome). A reduce law is an equation of the form $P = Q$, where we search for instances of P and replace them with the corresponding instance of Q. The idea is that we pattern-match on predicate syntax with the second predicate argument, to see if a law is applicable (we have its lefthand side), and if so, we then build an appropriate instance of the righthand-side. The plan is that we gather all these pattern/outcome pairs in one function definition, which will try them in order. This is in direct correspondence with Haskell pattern-matching. So for UTCP we have a function called `reduceUTCP`, structured as follows:

```
reduceUTCP d inv (...1st law pattern...) = 1st-outcome
reduceUTCP d inv (...2nd law pattern...) = 2nd-outcome
...
reduceUTCP _ _ _ = Nothing  -- catch-all at end, no change
```

A simple example of such a pattern is the following encoding of $II ; P = P$:

```
reduceUTCP d inv (Comp "Seq" [(Comp "Skip" []), pr])
   =  Just ( ";-lunit", pr, True )
```

The pattern matches a composite called “Seq”, with a argument list containing two predicates. The first predicate pattern matches a “Skip” composite with no further sub-arguments. The second argument pattern matches an arbitrary predicate (P in the law above). The righthand side constructs a `RWResult` return value, with the string being a justification note that says a reduction-step using a law called ";-lunit" was applied, and noting that the top-level composite (the "Seq") was modified.

7.2 Conditional Reduce

A `CRWResult` is a `RWResult` that has been adapted, so that instead of returning one result if successful, it returns a list of possible results, each paired with a side-condition predicate.

```
type CRWResult s  =  Maybe ( String, [( Pred s, Pred s, Bool)] )
type CRWFun s     =  Dict s -> Pred s -> CRWResult s
```

A conditional reduce law is an equation as per reduce, but with conditional outcomes, e.g. $P = Q_1 \lhd C \rhd Q_2$. Matching an instance of P will return a list of two pairs, the first being (C, Q_1), the second $(\neg C, Q_2)$. No attempt is made to evaluate C, but instead the REPL asks the user to choose. This is a key design decision for the calculator. A general purpose predicate evaluator requires implementing lots of theories about numbers, sets, lists, and whatever else might be present. Given the scope and purpose of this calculator it is much more effective to let the user choose.

For an example, here is one pattern of the conditional reduce function for UTCP. Given $\vec{x}$ a list of unique variables, and $\vec{e}$ a list of the same length of expressions, with $\vec{x} \subseteq \{s, ls\}$ we have:

$$\begin{aligned} c[\vec{e}/\vec{x}] &\implies (c * P)[\vec{e}/\vec{x}] = P[\vec{e}/\vec{x}]; c * P \\ \neg c[\vec{e}/\vec{x}] &\implies (c * P)[\vec{e}/\vec{x}] = II[\vec{e}/\vec{x}] \end{aligned}$$

```
creduceUTCP d (PSub w@(Comp "Iter" [c,p]) sub)
 | isCondition c && beforeSub d sub
 = Just( "loop-substn", [ctrue,cfalse] )
 where
   csub = PSub c sub
   ctrue  = (        csub, mkSeq (PSub p sub) w, diff )
   cfalse = ( mkNot csub, PSub mkSkip sub,       diff )
```

Here `mkSeq`, `mkNot` and `mkSkip` build sequential composition, negations and standard UTP skip (II) respectively. Both `isCondition` and `beforeSub` ensure that their arguments contain no dashed variables.

7.3 Loop Unroll

Iteration is typically defined in UTP as the least fixed point w.r.t. the refinement ordering that also involves sequential composition, which itself is defined using existential quantification, and II.

$$\begin{aligned} c * P &\mathrel{\widehat{=}} \mu L \bullet (P; L) \lhd c \rhd II \\ P; Q &\mathrel{\widehat{=}} \exists s_m, ls_m \bullet P[s_m, ls_m/s', ls'] \land Q[s_m, ls_m/s, ls] \\ II &\mathrel{\widehat{=}} s' = s \land ls' = ls \end{aligned}$$

We do not want to explicitly handle quantifiers, or fixed-points. Instead we prefer to use the loop unrolling law, as this is much more useful for the kinds of calculations we encounter.

$$c * P = (P \mathbin{;} c * P) \lhd c \rhd II$$

Even more useful are ones that split the conditional and unroll a number of times (; binds tighter than $\lor$ but looser than $\land$):

$$\begin{aligned} c * P &= \neg c \land II \lor c \land P \mathbin{;} c * P \\ &= \neg c \land II \lor c \land P \mathbin{;} c \land II \lor c \land P \mathbin{;} c * P \\ &= \ldots \end{aligned}$$

The loop unroll functions are like those for reduce but have an extra string argument: `unroll  :: [String -> RWFun s]`. When the user enters a command of the form `"lsss"`, the loop unroll facility is activated, and the string `"sss"` is passed as the first argument to the functions above. It is up to the user to decide how to interpret these strings—but the most useful is to treat them as specifying the number of unrollings to do. We won't give an example here of the use of unrolling.

7.4 Bringing it all together

We make these two reduction functions "known" to the calculator by adding them into a dictionary.

```
lawsUTCPDict
 = makeDict [("laws", LawEntry [reduceUTCP] [creduceUTCP] [])]
```

We then can take a number of partial dictionaries and use various dictionary functions, defined in `CalcPredicates`, to merge them together.

```
dictUTCP = foldl1 dictMrg [ alfUTCPDict , ... , lawsUTCPDict]
```

The main method of working with dictionaries is to construct small ones focussed on some specific area of interest. These can then be combined in different ways to provide a number of complete dictionaries that can vary in the order in which things are tried.

8 Conclusions

We have presented a description of a calculator written in Haskell, that allows the encoding of an UTP theory under development, in order to be able to rapidly perform test calculations. This helps to check that predictions of the theory match expectations. The tool was not designed to be a complete and sound theory development system, but instead to act as a rapid-prototype tool to help smoke out problems with a developing theory. This approach relies on the developer to be checking and scrutinising everything.

8.1 Costs vs. Benefits

As far as the development of the UTCP theory is concerned, the costs of developing and customising the calculator have been well compensated for by the benefits we encountered. This also applies to ongoing work to develop a fully compositional UTP theory of shared-state concurrency that does not require *run*. We note a few observations based on our experience using the calculator.

The "first-come, first-served" approach used by the calculator is surprisingly effective. We support a system of equational reasoning where reductions and definitions replace predicates with ones that are equal. In effect this means that the order in which most of these steps take place is immaterial. Some care needs to be taken when several rules apply to one construct, but this can be managed by re-arranging the order in which various patterns and their side-conditions can be checked.

The main idea in using the calculator is to find a suitable collection of patterns, in the right order, to be most effective in performing calculations. The best way to determine this is to start with none, run the calculator and when it stalls (no change is happening for any command), see what law would help make progress, and encode it. This leads to an unexpected side-effect of this calculator, in that we learnt what laws we needed, rather than what we thought we would need.

Effective use of the calculator results in an inexorable push towards algebras. By this we do not mean the Kleene algebras, or similar, that might characterise the language being formalised. Rather we mean that the most effective use of the calculator results when we define predicate functions that encapsulate some simple behaviour, and demonstrate, by proofs done without the calculator, some laws they obey, particularly with respect to sequential composition. In fact, one of the 'algebras' under development for the fully compositional theory, is so effective, that many of the test calculations can actually be done manually. However some, most notably involving parallel composition, still require the calculator in order to be feasible.

8.2 Correctness

An issue that can be raised, given the customisation and lack of soundness guarantees, is how well has the calculator been tested? The answer is basically that the process of using it ensures that the whole system is comprehensively tested. This is because calculations fail repeatedly. Such failures lead to a post-mortem to identify the reason. Early in the calculator development, the reason would be traced to a bug in the calculator infrastructure. The next phase has failures that can be attributed to bugs in the encoding of laws in Haskell, or poor ordering in the dictionary. What makes the above tolerable is that the time taken to identify and fix each code problem is relatively short, often a matter of five to ten minutes. The final phase is where calculation failures arise because of errors in the proposed theory—this is the real payback, as this is the intended purpose of the tool. The outcome of all of this iterative development is a high degree of confidence in the end result. In the author's experience, the cost of all the above failures is considerably outweighed by the cost of trying to do the check calculations manually.

There are no guarantees of soundness. But working on any theory by hand faces exactly the same issues — a proof or calculation by hand always raises the issue of the correctness of a law, or the validity of a "proof-step" that is really a number of simpler steps all rolled into one. In either case, by hand or by calculator, the theory developer has a responsibility to carefully check every line. This is one reason why so much effort was put into pretty-printing and marking. The calculator's real benefit, and *main design purpose*, is the ease with which it can produce a calculation and transcript.

In effect, this UTP Calculator is a tool that assists with the validation of UTP semantic definitions, and is designed for use by someone with expertise in UTP theory building, and a good working knowledge of Haskell.

8.3 Future Work

We plan a formal release of this calculator as a Haskell package. A key part of this would be comprehensive user documentation of the key parts of the calculator API, the standard built-in dictionaries, as well as a complete worked example of a theory encoding. There are many enhancements that are also being considered, that include better transcript rendering options (e.g. LaTeX) or ways to customise the REPL (e.g. always do a simplify step after any other REPL command). Also of interest would be finding a way of connecting the calculator to either the $U{\cdot}(TP)^2$ theorem-prover[4] or the Isabelle/UTP encoding[9] in order to be able to validate the dictionary entries. All the code described here is available online at
`https://bitbucket.org/andrewbutterfield/utp-calculator.git` as Literate Haskell Script files (`.lhs`) in the `src` sub-directory.

References

1. Bauer, F.L., Ehler, H., Horsch, A., Möller, B., Partsch, H., Paukner, O., Pepper, P.: The Munich Project CIP, Volume II: The Program Transformation System CIP-S, Lecture Notes in Computer Science, vol. 292. Springer (1987), http://dx.doi.org/10.1007/3-540-18779-0
2. Bertot, Y., Castéran, P.P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Texts in theoretical computer science, Springer Verlag (2004)
3. Bird, R.: Thinking Functionally with Haskell. Cambridge Univ. Press (Dec 2014)

4. Butterfield, A.: Saoithín: A theorem prover for UTP. In: Unifying Theories of Programming - Third International Symposium, UTP 2010, Shanghai, China, November 15-16, 2010. Proceedings. pp. 137–156 (2010), http://dx.doi.org/10.1007/978-3-642-16690-7_6
5. Butterfield, A.: The logic of $U \cdot (TP)^2$. In: Unifying Theories of Programming, 4th International Symposium, UTP 2012, Paris, France, August 27-28, 2012, Revised Selected Papers. pp. 124–143 (2012), http://dx.doi.org/10.1007/978-3-642-35705-3_6
6. Butterfield, A., Mjeda, A., Noll, J.: UTP Semantics for Shared-State, Concurrent, Context-Sensitive Process Models. In: Bonsangue, M., Deng, Y. (eds.) TASE 2016 10th International Symposium on Theoretical Aspects of Software Engineering. pp. 93–100. IEEE (Jul 2016)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: The maude 2.0 system. In: Nieuwenhuis, R. (ed.) Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2706, pp. 76–87. Springer (2003), http://dx.doi.org/10.1007/3-540-44881-0_7
8. Foster, S., Woodcock, J.: Mechanised theory engineering in isabelle. In: Irlbeck, M., Peled, D.A., Pretschner, A. (eds.) Dependable Software Systems Engineering, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 40, pp. 246–287. IOS Press (2015), http://dx.doi.org/10.3233/978-1-61499-495-4-246
9. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: A mechanised theory engineering framework. In: Naumann, D. (ed.) Unifying Theories of Programming - 5th International Symposium, UTP 2014, Singapore, May 13, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8963, pp. 21–41. Springer (2014), http://dx.doi.org/10.1007/978-3-319-14806-9_2
10. Hughes, J.: The design of a pretty-printing library. In: Jeuring, J., Meijer, E. (eds.) Advanced Functional Programming, LNCS, vol. 925. Springer Verlag (1995), http://www.cs.chalmers.se/~rjmh/Papers/pretty.ps
11. Marlow, S. (ed.): Haskell 2010 Language Report. Haskell Community (2010), https://www.haskell.org/definition/haskell2010.pdf
12. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002), http://link.springer.de/link/service/series/0558/tocs/t2283.htm
13. Nuka, G., Woodcock, J.: Mechanising a unifying theory. In: Dunne, S., Stoddart, B. (eds.) Unifying Theories of Programming, First International Symposium, UTP 2006. LNCS, vol. 4010, pp. 217–235. Springer (2006), http://dx.doi.org/10.1007/11768173_13
14. Shankar, N.: PVS: Combining specification, proof checking, and model checking. In: Srivas, M.K., Camilleri, A.J. (eds.) Formal Methods in Computer-Aided Design, First International Conference, FMCAD '96. LNCS, vol. 1166, pp. 257–264. Springer (1996)
15. Van Den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: the ASF+SDF compiler. ACM Transactions on Programming Languages and Systems 24(4), 334–368 (Jul 2002)
16. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. In: Middeldorp, A. (ed.) Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2051, pp. 357–362. Springer (2001), http://dx.doi.org/10.1007/3-540-45127-7_27

17. Wadler, P.: A prettier printer. In: Gibbons, J., de Moor, O. (eds.) The Fun of Programming (Cornerstones of Computing), chap. 11, pp. 223–244. Palgrave - Macmillan (Mar 2003)
18. Wenzel, M.: The Isabelle/Isar Reference Manual (June 2010), `http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/isar-ref.pdf`
19. Winter, V.L., Beranek, J.: Program transformation using HATS 1.84. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE. Lecture Notes in Computer Science, vol. 4143, pp. 378–396. Springer (2005), `http://dx.doi.org/10.1007/11877028_15`
20. Woodcock, J., Hughes, A.P.: Unifying theories of parallel programming. In: George, C., Miao, H. (eds.) Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21-25, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2495, pp. 24–37. Springer (2002), `http://dx.doi.org/10.1007/3-540-36103-0_5`
21. Zeyda, F., Cavalcanti, A.: Mechanical reasoning about families of UTP theories. Electr. Notes Theor. Comput. Sci 240, 239–257 (2009), `http://dx.doi.org/10.1016/j.entcs.2009.05.055`