UTPCalc — A calculator for UTP Predicates

Andrew Butterfield [*]

Trinity College Dublin

Abstract. The developments of a UTP Theory can involve a number of false starts, as alphabet variables are chosen and semantics and healthiness conditions are defined. Typically some calculations done just to check that everything is OK in fact reveal problems with the theory. So we iterate by revising the basic definitions, and attempting the calculations again. Hopefully, these eventually converge to what becomes a sound and useful UTP Theory. In a recent bout of such theory revision and re-calculation, which required 5 iterations in total, the author noted that common patterns of proof-steps kept occurring in each iteration. This inspired the development of the UTP-Calculator: a tool, written in Haskell, that supports rapid prototyping of new theories by supporting an easy way to very quickly perform calculations. 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 walking through how such a theory might be encoded.

1 Introduction

1.1 Research Context

We have recently started to explore using UTP to do the formal modelling of a language, called "Process Modelling Language" (PML), designed to describe software development and similar business processes [1]. The main objective is to give PML a formal semantics, as the basis for a number of analysis tools that could be made available to process designers and users—with one key application area being the modelling of clinical healthcare pathways. It quickly became apparent that PML and similar (business) process notations essentially involve concurrency with global shared mutable state.

There has been work using UTP to model concurrent programs with shared mutable state, most notably with an encoding into actions systems having been done by Woodcock and Hughes[11]. We have been looking at adapting this work to provide a UTP semantics for PML.

So we started by taking the action-system UTP semantics for Unifying Theories of Parallel Programming (UTPP) [11], and reworking it to use a system

[*] 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)

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) and details of this are, at the time of writing, currently under review[5]. We give a very brief overview here of this theory. The language assumes atomic actions (A) that modify a generic state, and four composition operators: sequential ($;;$), parallel ($\|$), non-deterministic choice ($\lhd\!\rhd$) and non-deterministic iterations ($^\omega$)

$$P ::= A \mid P \mathbin{;;} P \mid P \parallel P \mid P \lhd\!\rhd P \mid P^\omega$$

Note that this is essentially the same as the baseline "Command" language in the Views paper[6]. In UTCP we have two distinct groups of observations:

- Variables that track changes during execution (dynamic observations), modelling the generic state (s, s') and the global label-set (ls, ls') used to manage control-flow.
- Variables that record static parameters such the start (in) and stop (out) labels for a construct, as well as a label generator (g) associated with each construct.

So our UTCP theory is a non-homogeneous relation with alphabet s, s', ls, ls', g, in, and out.

1.2 Theory Construction Difficulties

The theory with its somewhat unusual arrangement of observation variables did *not* emerge as an immediate and obvious solution. The theory design started with an homogeneous alphabet with g', in' and out' being present. Semantic definitions were proposed for all constructs, and then a series of calculations were undertaken in order to check the validity of those definitions. The first such calculation was that of the semantics of a single atomic action (A), and this always seemed to work out fine. The second calculation was of the sequential compositions of two atomic actions ($A \mathbin{;;} B$), and this is where problems arose.

The first problem was relatively minor —a lot of the semantics talks about the presence in or absence of various labels from the global label set, and of modifications to that set. This resulted in very wide expressions, so some shorthand notations were introduced: using ℓ to denote singleton set $\{\ell\}$; or writing $ls(L)$ instead of $L \subseteq ls$, for example.

The second problem was more serious. The calculation to take $A \mathbin{;;} B$, expand it using the semantics of $;;$, and reduce it down to a simple predicate that described the fairly obvious correct behaviour, was very lengthy. Often it would give the right result, but then checking $A \parallel B$ (an even longer calculation) would give an incorrect answer. In every case the fault was traced back to sequential composition[1] , where either the definition of $;;$ was wrong, or, as proved crucial, the behaviour of "standard" UTP sequential composition was at fault. The

[1] In the authors experience, when building semantic theories involving concurrency or parallelism, it is always sequential composition that causes the most difficulty

calculations for sequential and parallel composition required typically about 400 lines of LaTeX and resulted in 7 full pages of output, an extract of which is shown in Fig. 1.

$$
\begin{array}{ll}
 & \bigvee_{i\in 1,2} (\ P[\{\ell_{g::2:}, \ell_{g::1:}\}, \rho_i/ls, \rho]; ((\ell_{g:} \notin ls) * P)\ ) \\
= & \text{“unabbrev. } P\text{”} \\
 & \bigvee_{i\in 1,2} (\ (\ D(\ell_g, \{\ell_{g::1}, \ell_{g::2}\}, ls) \lor D(\{\ell_{g::1:}, \ell_{g::2:}\}, \ell_{g:}, ls) \\
 & \qquad \lor G_{\ell_{g::1}, \ell_{g::1:}, ls}(\rho' = \rho \oplus \{x \mapsto [\![x+y]\!]_\rho\}) \\
 & \qquad \lor G_{\ell_{g::2}, \ell_{g::2:}, ls}(\rho' = \rho \oplus \{y \mapsto [\![y-x]\!]_\rho\})\ ) \\
 & \qquad [\{\ell_{g::2:}, \ell_{g::1:}\}, \rho_i/ls, \rho]; ((\ell_{g:} \notin ls) * P)\ ) \\
= & \text{“substn for } ls'\text{”} \\
 & \bigvee_{i\in 1,2} (\ (\ D(\ell_g, \{\ell_{g::1}, \ell_{g::2}\}, \{\ell_{g::2:}, \ell_{g::1:}\}) \lor D(\{\ell_{g::1:}, \ell_{g::2:}\}, \ell_{g:}, \{\ell_{g::2:}, \ell_{g::1:}\}) \\
 & \qquad \lor G_{\ell_{g::1}, \ell_{g::1:}, \{\ell_{g::2:}, \ell_{g::1:}\}}(\rho' = \rho \oplus \{x \mapsto [\![x+y]\!]_\rho\}) \\
 & \qquad \lor G_{\ell_{g::2}, \ell_{g::2:}, \{\ell_{g::2:}, \ell_{g::1:}\}}(\rho' = \rho \oplus \{y \mapsto [\![y-x]\!]_\rho\})\ ) \\
 & \qquad [\rho_i/\rho]; ((\ell_{g:} \notin ls) * P)\ )
\end{array}
$$

Fig. 1. Extract from 4th attempt to calculate $x := x + y \parallel y := y - x$

With choice and iteration to be covered it was becoming very clear that a better way was required for checking semantic outcomes.

1.3 A Plan

We briefly considered using the $U \cdot (TP)^2$ theorem prover [3,4], but it would have required a lot of setup effort, and it is currently not in an ideal state[2] . However, as part of our work on the formal semantics of PML, we had developed a parser and some initial analysis tools in Haskell[10], 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, over all of those calculation attempts (5 in all!) that the pattern of each calculation was very uniform and similar. So a decision was taken to construct the calculator described in this paper.

1.4 Structure of this paper

In Sect. 2 we describe key design decisions that underly the calculator's features, and which are responsible for its success. Then, in Sect. 3 we present a development of a theory using the calculator, as a way of describing its features, from a users perspective. This is followed by a short example of the calculator in use

[2] The issue has to do with difficulties installing the relevant software libraries on more recent versions of Haskell.

(Sect. 4). We then discuss this calculator, and related work and draw some conclusions in Sect. 5, before finishing up describing future plans for the calculator (Section 5.1).

2 Design & Architecture

2.1 Development Process

A very early decision was made to adhere to Agile Software development principles [8] in developing this calculator (to the extent possible given that the roles of Software Engineer, Scrum Manager and Customer were all rolled into one). In particular we stuck close to the YAGNI ("Ya Ain't Gonna Need It") principle[3] which requires us to only write software for a function that is required at that time. This does not prevent advanced design planning but does keep the development focussed on immediate needs. So initially the focus was on being able to use the calculator to expand the UTP definition of an atomic action. Once that was working, then the focus shifted to additional code to support the calculation of the sequential composition of two atomic actions, and so on. For example, the feature to take a final calculation and output it to a file was only developed when this paper was being written, because there was no need for it until this point.

2.2 Software Architecture

All the code described here is available online as a git repo at `https://andrewbutterfield@bitbucket.org/andrewbutterfield/utp-calculator.git` as Literate Haskell Script[4] (`.lhs`) files in the `src` sub-directory.

Taking into account the repetitive nature of the calculations, as mentioned at the end of §1.3, and the need for shorthand notations we very rapidly converged on four initial design decisions:

1. No parsers! All calculation objects are written directly in Haskell.
2. We would keep the expression and predicate datatype declarations very simple, with only equality being singled out.
3. We would need to have a good way to pretty-print long predicates that made it easy to see their overall structure
4. We would reply 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:

[3] More formally, "Simplicity—the art of maximizing the amount of work not done—is essential."
[4] Note, this paper is not itself a literate Haskell script (you'll be relieved to know).

```
data Expr s
  = St s | B Bool | Z Int | Var String
  | App String [Expr s] | Sub (Expr s) (Substn s) | Undef

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

type MPred s = (Marks, Pred s)
```

Fig. 2. Expression and Predicate Datatypes (`CalcTypes.lhs`)

- 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 use 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.

We will now expand on some of the points above, and in so doing expose some of the concrete architecture of the calculator code.

2.3 Expressions and Predicates

In Fig. 2 we show the Haskell declarations of the datatypes to represent both expressions and predicates. Of course this means we are developing a deep embedding[9], which is required in order to be able to do the kind of calculations we require. Both datatypes are parameterised on a generic state type `s`, which allows us to be able to handle different concrete types of shared state with one piece of code. Both types have basic values (`St`,`B`,`Z`,`T`,`F`), variables (`Var`,`PVar`), generic composites (`App`,`Comp`), substitution (`Sub`,`PSub`) and undefined values (`Undef`,`PUndef`). The predicate datatype has a way to embed a (boolean-valued) expression to form an atomic predicate (`Atm`), as well as an explicit form for atomic predicates that take the form of an equality (`Equal`). We give a specialised form for equality simply because its use in laws is widespread and worth optimising.

Also important to point out is the fact that `Pred` is defined in terms of `MPred`, which in turn is defined in terms of `Pred`. This is done to facilitate the association of markings (lists of integers) with predicate constructs. These markings are used to indicate which parts of a predicate were changed at each calculation step. We will not discuss marking further in this document as it runs completely "under the hood", and its only impact on the users of this calculator is their need to be aware of the interplay involving `Pred` and `MPred`.

2.4 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. An example of such pretty-printing in action is

```
    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)))
```

The top-level structure of this is

$$D(out) \lor ((\neg ls(out) \land \ldots); \mathbf{W}(\ldots))$$

The pretty printing support can be found in `PrettyPrint.lhs`.

2.5 Dictionaries

The abstract syntax is very simple, and uses names as Haskell Strings to distinguish between different composites. So for example logical operators $\land$ and $\lor$ might be represented as `Comp "And"` and `Comp "Or"` respectively. We have a default way to pretty-print composites, so that the predicate **true**$\land$**false**, represented in Haskell by `Comp "And" [bT,bF]`, would render as `And(true,false)`. However it would be nice to be able to render it as the usual infix operator, giving `true /\ false`. In addition, we need some way to associate laws and definitions, as appropriate, with composites.

The approach taken is to implement 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 standard approach is that the calculator will seek information regarding some name that is currently of interest. It will perform a dictionary lookup which can result in either failure, for which a suitable default action will be used, or success, in which case the dictionary entry will supply information to guide the required name-specific behaviour.

There are four kinds of dictionary entries, one each for expressions and predicates, one for laws of various kinds, and one for alphabet handling. We will discuss each kind in turn, but leave examples of their use until Sect.3.

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 discussed below.

Expression Entries

```
Entry s
  = 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}
  | ...
```

Imagine that `App name args` is being processed and that a dictionary lookup with key `name` has returned an expression entry. The entry contains four pieces of information:

`ecansub` is a list of variable names, over which it is safe to perform substitutions.

`eprint` is a function that takes a dictionary as first argument, the argument-list `args` as its second argument and returns a string rendering of the expression. Currently we view expressions as atomic one-line texts for output purposes.

`eval` takes similar arguments to `eprint`, but returns a string/expression pair, that denotes a possible evaluation or simplification of the original expression. The string is empty if no change occurred, otherwise it is a short description of/name for the eval/simplify step.

`isEqual` has a dictionary argument, followed by two sub-expression lists. it tests for (in)equality, returning `Just True` or `Just False` if it can establish (in)equality, and `Nothing` if unable to give a definitive answer.

The dictionary argument supplied to the three functions is always the same as the dictionary in which the entry was found.

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. Now, consider the following instance, with a substitution, and two attempts to calculate a full expansion:

$$\begin{array}{rcl} & D(\{out\})[\{\ell_1\},\ell_2/ls,out] & \\ (\{out \subseteq ls\})[\{\ell_1\},\ell_2/ls,out] = & & = D(\{\ell_2\}) \\ \{\ell_2\} \subseteq \{\ell_1\} = & & = \{\ell_2\} \subseteq ls \end{array}$$

The lefthand calculation is correct, the righthand is not. The substitution refers to variables (e.g. ls) that are hidden when the D shorthand is used. 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"]`. Given that entry, the calculator would simplify (correctly) as follows:

$$D(\{\ell_2\})[\{\ell_1\}/ls]$$

The righthand side of the definition is what should be returned by the `eval` component. If we want to state that any substitution is safe, then we use the "wildcard" form: `["*"]`.

Predicate Entries

```
Entry s
  = ...
  | PredEntry
    { pcansub :: [String]
    , pprint :: Dict s -> MarkStyle -> Int -> [MPred s] -> PP
    , alfa :: [String]
    , pdefn :: Rewrite s
    , prsimp :: Rewrite s }
type Rewrite s = Dict s -> [MPred s] -> (String, Pred s)
```

The predicate entry associated with Comp name pargs has five fields:

`pcansub` does for predicates what `ecansub` does for expressions.

`pprint` is similar to `eprint` except it has two extra arguments, used to help with special renderings for predicates marked as changed, and precedence levels for managing bracketing. We do pretty-printing tricks such as indenting and adding line-breaks at the predicate level.

`alfa` is used to identify the alphabet of the predicate.

`pdefn` is a function invoked when `name` has a definition, and we want to expand it.

`prsimp` is called by the simplifier when processing `name`.

Both `pdefn` and `prsimp` take a dictionary argument and list of sub-component (marked)predicates, returning a string/predicate pair, interpreted in a manner similar to `eval` above.

Law Entries

```
Entry s
  = ...
  | LawEntry  { reduce :: [DictRWFun s]
              , creduce :: [CDictRWFun s]
              , unroll :: [String -> DictRWFun s] }
```

Law entries are currently only associated with one key, namely the string `"laws"`, which is what the calculator uses to find such entries. There are three parts, each consisting of lists of functions. These lists are intended to be be applied in order to the “current predicate”, until either one succeeds, or none do. All the functions take a dictionary and (marked) predicate as arguments, and return either an single-outcome indicator (`reduce`,`unroll`):

```
type DictRWFun s = Dict s -> RWFun sdata
type RWFun s = MPred s -> RWResult s
type RWResult s = (String, MPred s)
```

or a conditional multiple-outcome result (`creduce`).

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

`reduce` is a list of reduction laws, to be tried out when a reduction step is invoked by the user.

`creduce` is a list of conditional reduction laws, which have multiple outcomes dependent on a side-condition. Rather than try to resolve conditions automatically, we prefer to let the user choose the appropriate outcome.

`loop` is a list of loop-unrolling functions. The extra string argument is to give the user finer control of how much unrolling is done, and how it is presented.

Alphabet Entries

```
Entry s
  = ...
  | AlfEntry  { avars :: [String]}
```

Most simply put, an alphabet is simply a set of variables. In any UTP theory we typically have well-defined alphabets, often with particular subsets of interest, such as all the "before" observations (undashed variables), or "after" observations (dashed variables). We give these subsets standardised names, and use the dictionary to list the relevant variables.

2.6 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. It then offers the user the opportunity to invoke various commands to perform calculation steps. The user can then indicate when they are finished, at which point the calculator function returns a data-structure that logs the complete calculation outcome. This is a triple consisting of the final predicate, a list of the steps, each with a justification string, and the dictionary that was used.

```
type CalcLog s = (MPred s, [RWResult s], Dict s)
```

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, depth-first 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 turns out to be strikingly effective in practice. This is because these kinds of semantic "smoke-test" calculations tend to go in phases: expand all definitions; simplify; reduce; simplify; perhaps unroll a loop a bit; etc...

Defn. Expand ('`d`') Find the first predicate changed by applying its `pdefn` dictionary entry, which should unfold its definition.

Law Reduce ('`r`') Find the first predicate transformed by a function in the `reduce` list of the `LawEntry` indexed by the string `"laws"`. This function captures an equational law, that is only applied in a left to right direction.

Loop Unroll ('`l`') Find the first predicate transformed by a function in the `unroll` list of the `LawEntry` indexed by the string `"laws"`. The remainder of the command string after the initial '`l`' is passed to the function to control the nature and degree of unrolling. How this string is interpreted is entirely up to the user.

Conditional Reduce ('`c`') Find the first predicate transformed by a function in the `creduce` list of the `LawEntry` indexed by the string `"laws"`. This function captures an equational law with a side condition, which returns a list of alternatives, each alternative being a side-condition predicate paired with a result. These are presented to the user, who then selects which outcome is appropriate. In effect the user has to look at each condition and select the one (if any) that evaluates to true. This is done to prevent the calculator from having to embody predicate simplifiers "modulo various expression theories".

3 Building Theories

We now present extracts from a theory built using this approach to illustrate how the calculator is used. As already stated, the user of this calculator does require some expertise in functional programming with languages that support pattern-matching (e.g. Haskell, ML). We do this by giving an overview of the Haskell encoding of the UTCP theory that was the motivation for this work. The emphasis here is on how to encode a UTP theory for use with this calculator, rather than trying to explain the theory itself, or its design motivation

3.1 Before we dive in . . .

The UTP Calculator is implemented as a series of Haskell modules, which are broken into two groups:

Infrastructure are modules that implement the calculator mechanics, pretty-printing, etc. These include `PrettyPrint`, and all modules with names starting with `Calc`.

Builtin Theories are pre-defined theory modules that cover standard logic, whose names start with `Std`, and modules that cover "standard" UTP, whose names start with `StdUTP`. These theory modules typically come in three, covering `Predicates`, `Precedences` and `Laws`.

All the Haskell modules are found in the `src` directory of the repository, with a `.lhs` file extension (e.g., `CalcTypes.lhs`).

3.2 UTCP Syntax

We start by defining the syntax of our language

$$p, q \in UTCP ::= Idle \mid \mathbf{A}(A) \mid p \,;;\, q \mid p \blacktriangleleft c \blacktriangleright q \mid p \parallel q \mid c \circledast p$$

and assign them pretty printing precedences, so they work well with the definitions in modules `StdPrecedences` and `StdUTPPrecedences`.

```
precPCond = 5 + precSpacer  1
precPPar  = 5 + precSpacer  2
precPSeq  = 5 + precSpacer  3
precPIter = 5 + precSpacer  6
```

The `precSpacer` function is used to leave space between precedence values so that new constructs can be given an in-between spacing, if required. Currently it returns its argument times 1.

3.3 UTCP Alphabet

As already stated, the theory alphabet is $s, s', ls, ls', g, in, out$. We declare each as a variable in our expression notation, noting that Haskell allows identifiers to contain dashes, which proves very convenient:

```
s' = Var "s'"
```

Note, here `s'` is a Haskell variable of type `Expr`, while `"s'"` is a Haskell literal value of type `String`.

We have two ways to classify UTP observation variables. Along one axis we distinguish observations of program variable values ("script" variables, e.g. s, s') from those that record other observations such as termination/stability, or traces/refusals ("model" variables, e.g. ls, ls'). On the other axis we distinguish observations that are dynamic, whose values change as the program runs (e.g. s, ls with s' and ls') from those that are static, unchanged during program execution (e.g. g, in and out). We have pre-defined names for these categories, and an function `stdAlfDictGen` that builds all the appropriate entries given three lists of script, dynamic model and static variable strings. We also declare that the predicate variables A, B and C will refer to atomic state-changes, and so only have alphabet $\{s, s'\}$.

```
alfUTCPDict
 = dictMrg dictAlpha dictAtomic
 where
   dictAlpha = stdAlfDictGen ["s"] ["ls"] ["g","in","out"]
   dictAtomic = makeDict [ pvarEntry "A" ss'
                         , pvarEntry "B" ss'
```

```
                              , pvarEntry "C" ss' ]
   ss' = ["s", "s'"]
```

(See modules `CalcAlphabets` , `CalcPredicates` , `StdPredicates`.)

3.4 UTPC Expressions

We have sets of labels so we need a way to implement set-expressions. To avoid long set expressions a number of shorthands 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 $\{in\} \subseteq ls$. So we might encode a set construct as follows

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

We also define an equality tester for sets, that gets the two element-lists

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

The predicate `isGround` checks to see if an expression has no dynamic variables. For the purposes of this theory at least, we know we can treat these expressions as values. This is a common feature of encoding theories for this calculator—knowing when a particular simplification makes sense. The dictionary entry for the set construct then looks like

```
ExprEntry ["*"] showSet none eqSet
```

where we permit any substitutions directly on the elements, and we use the special builtin function `none` as an evaluator that does not make any changes, since we regard these sets as evaluated, in this theory.

Similar tricks are used to code a very compact rendering of a mechanism that involves unique label generators that can also be split, so that an expression like

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

can be displayed as $\ell_{g:1:}$, or, within the calculator, as `lg:1:` .

3.5 Coding Atomic Semantics

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

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

where A is a predicate whose alphabet is restricted to s and s'. We code this up as follows:

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

defnAtomic d [a] = ldefn shPAtm $ mkAnd [lsin,a,ls'eqlsinout]

inp = Var "in" -- 'in' is a Haskell keyword
out = Var "out"
lsin = atm $ App "subset" [inp,ls]
lsinout = App "sswap" [ls,inp,out]
ls'eqlsinout = equal ls' lsinout

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

Here `atm` lifts an expression to a marked predicate, while `"sswap"` names the ternary operation $\_ \ominus (\_, \_)$, and `equal` is the marked form of `Equal`.

We won't show the encoding of the composite constructs, or a predicate transformer called *run* that actually enables us to symbolically 'execute' our semantics. We will show how the `pprint` entry for sequential composition in UTCP is defined, just to show how easy support for infix notation is.

```
ppPSeq d ms p [mpr1,mpr2]
 = paren p precPSeq -- parenthesise if precedence requires it
     $ ppopen  (pad ";;") [ mshowp d ms precPSeq mpr1
                          , mshowp d ms precPSeq mpr2 ]
```

Here `pad` puts spaces around its argument, while `ppopen` uses its first argument as a separator between all the elements of its second list argument. The function `mshowp` is the top-level predicate printer.

3.6 Coding UTCP Laws

The definition of the semantics of the UTCP language constructs, and of *run*, make use of the (almost) standard notions of skip, sequential composition and iteration in UTP. The versions used here are slightly non-standard because we have non-homogeneous relations, where the static parameters have no dashed counterparts. In essence we ignore the parameters as far as flow-of-control is concerned:

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

Here, the definition of $\lhd \_ \rhd$ is entirely standard, of course.

What is key here though, is realising that we do not want to define the constructs as above and use them directly, as it involves quantifiers and explicit recursion, both of which would introduce considerable complexity to the calculator. Instead, we encode useful laws that they satisfy, that do not require their definitions to be expanded. Such laws might include the following:

$$\begin{array}{rcl} II \ ; \ P \ = P = & P \ ; II \\ c * P & = & (P \ ; c * P) \lhd c \rhd II \\ (c * P)[e/x] & = & P[e/x] \ ; c * P, \qquad if c[e/x] \\ (c * P)[e/x] & = & II[e/x], \qquad if \neg c[e/x] \end{array}$$

These laws need to be proven by hand (carefully), by the theory developer, and then encoded into Haskell (equally carefully), as we are about to describe.

We can easily give a definition of II, which is worth expanding.

$$II \mathrel{\widehat{=}} s' = s \land ls' = ls$$

```
defnUTCPII = mkAnd[ equal s' s, equal ls' ls ]
```

For more complex laws, the idea is that we pattern-match on predicate syntax 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 a direct match for how Haskell pattern-matching works. So for UTCP we have a function called reduceUTCP, structured as follows:

```
reduceUTCP :: (Show s, Ord s) => DictRWFun s
reduceUTCP (...1st law pattern...) = 1st outcome
reduceUTCP (...2nd law pattern...) = 2nd outcome
...
reduceUTCP d mpr = lred "" mpr  -- catch-all at end, no change
```

The last clause matches any predicate and simply returns it with a null string, indicating no change took place. The main idea is find a suitable collection of patterns, in the right order, to be most effective in performing calculations. The best way to determine this is 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 is the essence of the agile approach to theory calculator development.

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

```
reduceUTCP d
 (_,Comp "Seq" [(_,Comp "Skip" []), mpr]) = lred ";-lunit" mpr
```

The second argument has type marked-predicate (`MPred`) which is a marking/predicate pair. We are not interested in the markings so we use the wildcard pattern '_' for the first pair component. The sub-pattern in the second pair component, `Comp "Seq" [(_,Comp "Skip" []), mpr])`, matches a composite called "Seq", with a argument list containing two (marked) predicates. The first argument predicate pattern `(_,Comp "Skip" [])` matches a "Skip" composite with no further subarguments. The second argument pattern `mpr` matches an arbitrary predicate (P in the law above). The righthand-side returns the application `lred ";-lunit" mpr` which simply constructs a string/predicate pair, with the string being a justification note that says a reduction-step using a law called ";-lunit" was applied.

3.7 UTCP Recognisers

Some laws require matching that is a bit more sophisticated. For example, consider a useful reduction for tidying up at the end, assuming that $ls' \notin A$ and $ls \notin B$, and both k and h are ground:

$$A \land ls' = k; B \land ls' = h \equiv (A; B) \land ls' = h \quad \langle\!\langle ls'\text{-cleanup}\rangle\!\rangle$$

However, we want this law to work when both A and B are themselves conjunctions, with the $ls' = \ldots$ as part of the same conjunction, located at some arbitrary position. We can break the problem into two parts. First we do a top-level pattern match to see that we have a top-level sequential composition of two conjunctions, then we use a function that will check both conjunction predicate-lists for the existence of a $ls' = \ldots$ component, and that everythiong else also satisfies the requirements regarding the occurrence, or otherwise of ls or ls':

```
reduceUTCP d pr@(_,Comp "Seq" [ (_,Comp "And" pAs)
                              , (_,Comp "And" pBs)])
 = case isSafeLSDash d ls' ls' pAs of -- no ls' in rest of pAs
    Nothing -> lred "" pr
    Just (_,restA) ->
     case isSafeLSDash d ls' ls pBs of -- no ls in rest of pBs
      Nothing -> lred "" pr
      Just (eqB,restB)
       -> lred "ls'-cleanup" $    -- build RHS
             comp "And" [ comp "Seq" [ bAnd restA
                                     , bAnd restB ]
                        , eqB ]
```

The function `isSafeLSDash` is designed to (i) locate the $ls' = e$ conjunct and check that its rhs is a ground expression; (ii) check that none of the remaining conjuncts make use of the 'unwanted' version of the label-set variable (ls or ls'); and (iii), if all ok, return a pair whose first component is the ($ls' = \ldots$) equality, and whose second is the list of other conjuncts. To achieve (i) above, we make use of two functions provided by the `CalcRecogniser` module:

```
mtchNmdObsEqToConst :: Ord s => String -> Dict s -> Recogniser s
matchRecog :: (Ord s, Show s)
           => Recogniser s -> [MPred s]
           -> Maybe ([MPred s],(MPred s,[MPred s]),[MPred s])
```

where

```
type Recogniser s = MPred s -> (Bool, [MPred s])
```

A recogniser is a function that takes a predicate and if it "recognises" it, returns `(True, parts)`, where parts are the subcomponents of the predicate in some order. The recogniser `mtchNmdObsEqtoConst v d` matches a predicate of the form `Equal (Var v) k`, returning a list with both parts. The function `matchRecog` takes a recogniser and list of predicates and looks in the list for the first predicate to satisfy the recogniser, returning a triple of the form (before,satisyingPred,after). If the recogniser suceeds, we then check the validity of the expression, and the absence of the unwanted variable from the rest of the conjuncts — using boolean function `dftlyNotInP` ("definitely not in P), so handling task (ii) above.

```
isSafeLSDash d theLS unwanted prs
 = case matchRecog (mtchNmdObsEqToConst theLS d) prs of
    Nothing -> Nothing
    Just (pre,(eq@(_,Equal _ k),_),post) ->
     if notGround d k
      then Nothing
      else if all (dftlyNotInP d unwanted) rest
       then Just (eq,rest)
       else Nothing
     where rest = pre++post
```

3.8 Conditional Reductions

To avoid having to support a wide range of expression-related theories, we provide a conditional reducer, that computes a number of alternative outcomes, each guarded by some predicate that is hard to evaluate. The user elects which one to use by checking the conditions manually. We define a function, similar to reduceUTCP, that contains a series of patterns fro each conditional reduction law.

```
creduceUTCP :: (Show s, Ord s) => CDictRWFun s
```

Provided that $\boldsymbol{x} \subseteq in\alpha P$ (which in this case is $\{s, ls\}$):

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

```
creduceUTCP d (_,PSub w@(_,Comp "Iter" [c,p]) sub)
 | isCondition c         -- true if expr c is a UTP 'condition'
   && beforeSub d sub    -- true if subst-vars are all undashed
 = lcred "loop-substn" [ctrue,cfalse]
 where
   csub = psub c sub             --  psub builds a substitution
```

```
ctrue  = ( psimp d csub          -- psimp runs a simplifier
         , bSeq (psub p sub) w )
cfalse = ( psimp d $ bNot csub
         , psub bSkip sub )
```

If this succeeds, the user is presented with two options, each of the form (side-condition, outcome) The user can then identify which of those side-conditions is true, resulting in the appropriate outcome.

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]
```

3.9 What's Missing?

No Quantifiers They bring a world of pain with them, and as it turns out, are not required for the kinds of calculations we wish to do. Handwritten proofs of laws involving concepts defined using quantifier and binders are required to validate the laws, but these are easy to do—the pain is automating these proofs, not doing them by hand.

No Targetting First-come, first-served, just works. We don't provide a facility to allow the user to specify precisely which law to apply, or which sub-predicate should be the target of a rewrite attempt. Instead we allow a blunt command that simply chooses between definition-expansion, reduction, simplification or conditional-reduction. The calculator engine then simply looks for the first location were the command succeeds. To date this has had no major effect on the ability of the calculator to work, but instead makes it very easy and fast to explore the calculation space.

4 Using the Calculator

The first property of interest for this calculator was calculating the effect of $run(A;;B)$, where A and B where atomic action predicates with alphabet $\{s, s'\}$. For convenience we predefined the predicate $A \mathbin{;;} B$, as

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

We then invoke the calculator as follows,

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

and proceed to interact (here the prompt " `?,d,r,l,s,c,u,x :-`" shows the available commands)

```
UTCP-0.7, UTP-Calc v0.0.1
run(A(A) ;; A(B))
 ?,d,r,l,s,c,u,x :- d
 = "defn. of run.3"
   (A(A) ;; A(B))[g::,lg,lg,lg:/g,in,ls,out]
 ; ~ls(lg:) * (A(A) ;; A(B))[g::,lg,lg:/g,in,out]
 ?,d,r,l,s,c,u,x :- d
 = "defn. of ;;"
   (A(A)[g:1,lg/g,out] \/
   A(B)[g:2,lg/g,in])[g::,lg,lg,lg:/g,in,ls,out]
 ; ~ls(lg:) * (A(A) ;; A(B))[g::,lg,lg:/g,in,out]
 ?,d,r,l,s,c,u,x :- s
 = "simplify"
   A(A)[g:::1,lg,lg,lg::/g,in,ls,out] \/
   A(B)[g:::2,lg::,lg,lg:/g,in,ls,out]
 ; ~ls(lg:) * (A(A) ;; A(B))[g::,lg,lg:/g,in,out]
.... 10 more steps
 ?,d,r,l,s,c,u,x :- r
 = "ls'-cleanup"
(A ; B) /\ ls' = {lg:}
```

The user simply indicates the broad class of command required, and the calculator works on the current goal predicate. A text transcript is produced, which is essentially the above without the prompts. The marking facility, requiring the `Pred`/`MPred` complication, is currently used when displaying the transcript in a terminal window to highlight old and new parts of predicates as chages are made. Currently this only works well on the Mac OS X terminal, because it seems to be the only one that properly supports ANSI secpae sequences.

5 Conclusions

The calculator was used to develop the semantic definitions for our UTP theory of concurrent programming (UTCP), and has proved to be invaluable in checking that all the definitions led to the correct outcomes. It continues to be in active use with work in this area that is extending and improving the theory.

However, there are no guarantees of soundness. Great care has been taken to ensure that the pre-packaged dictionaries are correct, and similar care is needed to ensure that theory-specific definitions are correct. But working on a 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 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.

A key lesson learnt during the development of both the calculator, and the UTCP theory, is the value of the agile approach. By focussing development of both on what was the immediate need at any given moment, we found that the calculator, and its dictionaries, were prevented from excessive bloat e.g., coding up a common, useful, obvious law, that actually wasn't needed. What was important was finding the "sweet spot" between the use of definition expansion, and hard-coded laws based on by-hand proofs.

A very interesting observation was that most effective use of the calculator seems to involve defining shorthand combinator like notations, and hand-proving key laws about how they interact with key operators used in semantics definitions, such as standard UTP sequential composition. This drives the development of what is a bespoke semantics algebra.

There is very little work similar to this in the literature. In [2] we find a calculator for point-free equational proofs as a final case-study. Also of interest is the discussion of deep/shallow embedding in [9], which suggests, that although our calculator is based on deep embedding, the way it uses hand-coded laws from dictionaries it more like shallow-embedding in character.

5.1 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[3] or the Isabelle/UPT encoding[7] in order to be able to validate the dictionary entries.

References

1. Atkinson, D.C., Weeks, D.C., Noll, J.: Tool support for iterative software process modeling. Information & Software Technology 49(5), 493–514 (2007), `http://dx.doi.org/10.1016/j.infsof.2006.07.006`
2. Bird, R.: Thinking Functionally with Haskell. Cambridge Univ. Press (Dec 2014)
3. 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`

4. Butterfield, A.: The logic of $U \cdot (TP)^2$. In: Wolff, B., Gaudel, M., Feliachi, A. (eds.) Unifying Theories of Programming, 4th International Symposium, UTP 2012, Paris, France, August 27-28, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7681, pp. 124–143. Springer (2012), http://dx.doi.org/10.1007/978-3-642-35705-3_6
5. Butterfield, A., Mjeda, A., Noll, J.: UTP Semantics for Shared-State, Concurrent, Context-Sensitive Process Models. In: Bonsangue, M., Deng, Y. (eds.) TASE 2016 (ubnder review)
6. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: compositional reasoning for concurrent programs. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 287–300. ACM (2013)
7. 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
8. Fowler, M., Highsmith, J.: The Agile Manifesto. Software Development Magazine 9(8) (Aug 2001), http://agilemanifesto.org
9. Gibbons, J., Wu, N.: Folding domain-specific languages: deep and shallow embeddings (functional pearl). ACM SIGPLAN Notices 49(9), 339–347 (Sep 2014)
10. Marlow, S. (ed.): Haskell 2010 Language Report. Haskell Community (2010), https://www.haskell.org/definition/haskell2010.pdf
11. 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