UNIE Version ??

Overview

- 63 modules
- ~ 13,500 lines (probably will reduce to around 12,000 by submission date)
- A lot of documentation/help files to be written

Basic Interactions

Four ‘modes’ of operation:
- Initial, Transformation, Script, Transformation + Script = ‘Transcript’
- Dictate how UNIE responds to commands, what is displayed on screen to the user, the level of command/proof state logging
    - E.g., ‘left’ in Initial/Script mode gives a program state error because it is a navigation command and should only be used when a program is being transformed
    - Show what appears on screen for trans $\x.x$
        - show-hist
    - Show what appears on screen for run-script ./scripts/rolling_rule_a
        - show-active-script
    - <return> in initial/trans mode vs script mode

Commands are accepted into the terminal, implemented using Haskeline as per GHCI:
- Command completion ‘:<TAB>’
- Command history via up/down arrow keys
- Clear screen nicely Ctrl-L
- No buffering issues with pressing left/right arrows etc.
- Allows us to catch escape sequences which is important for when the ‘evaluation’ mode comes online (to be added very soon)
    - Cf. GHCI, will not store consecutive duplicates
- Command parameters are entered in an applicative style, and can be input in any order ‘within reason’
    - show-lib ctx-eqs std
    - show-lib std ctx-eqs
    - Depends on type of parameter, same type will be bias towards the order they appear on the command line
- Output is displayed rigorously and systematically
    - Terminal: formatted to 80 char width using HughesPJ printing library
        - show-lib .. example of borders, headings, spacing, bullets/ numbering

- File: the same underlying implementation but defaults to 100 char width
- Both settings can be changed and the output will update accordingly with no effort on the user's part
- I have basically written a mini library for formatting output. A new version will incorporate error message output (soon)
- Some colouring, mainly for my own amusement, will make it so user can turn it off at some point (in case not ANSI compatible)

Command/parameter errors

- Uniformly deals with commands that accept different parameters:
    - trans '<name>
    - trans <src>
    - trans <proposition>
    - trans <missing_param>
    - trans <prop> <additional param>
    - trans <additional param> <proposition>
    - StateCmd.hs line 29
        - Tell the command parser module the name of the command and the 'type' of parameters it accepts, and it does the rest for you, handling additional/invalid/missing parameters and reporting those errors to the user

Three step process for executing commands

- Command 'matching' which is a kind of of type/form checking for command parameters
- Command 'refining' where a command's parameters are made sense of
- Command 'interpreting' where the command is executed in the current context, whether it fails or succeeds depends on the context
- This gives us a well-defined error hierarchy
    - Lexical errors
    - Param type/form errors
    - **Relation errors — here the command is checked to ensure it is safe to apply by checking its operator <~>. Acts as thin wrapper around the KURE implementations for AST rewrites**
    - Refining errors
    - Interpretation errors

Base library features

- Import definitions form files and store them in the library
    - Term definitions checked for validity

    - Context definitions, checked to make sure they are the correct context kind and will be rejected if not
        - E.g., a value context marked as applicative A_ will be rejected
    - Import module checks contexts/terms are not vice-versa
    - Currently a context is marked with its kind e.g., V_ at start of name, UNIE can detect context kind automatically so this is not actually needed, but I've been happy to leave it on whilst I've been developing as it's been good documentation in the proofs
    - import-lib ./libs/rolling_rule
    - show-lib terms
    - show-lib 'm (side note , 'm is a nod to HERMIT)
- Import and store cost-equivalent contexts: widely used in proofs for simplicity. Needed for Jenny's proofs. This was where I put a backslash in front of commands previously, got rid of that and opted for safer approach.
    - Cost-equiv. contexts are uniformly built into UNIEs context generation/matching algorithms, meaning they are treated as if they were of the correct form
    - Open to abuse at the minute, but very useful so worth it
    - Will later update to make them safe once proved cost-equid
- Show/delete all library
- Export libraries to file
    - export-lib terms ./test
    - export-lib ./test (contexts + terms)
    - export-lib ctx-eqs std ./test (standard cost-equiv. contexts)
    - UNIE makes pretty files with a pretty header (needs updating)
    - Outputs pretty printed syntax and can parse it too

Transformation features

- Transform raw source code
    - trans <src>
- Transform terms from the library
    - trans '<name>
- Transform with a proposition
    - trans X improved_by Y
- Cannot transform contexts with holes: not valid
- Full history of a transformation stored in environment
    - show-hist … laid out like a proof with commands as hints
    - show-hist state .. useful when you get some large commands printed
    - show-hist cmds
    - Can be exported to file
        - export-hist ./test

        - Prints a warning if the global relation wasn't set.. i.e., if UNIE wasn't checking for operator consistency
    - Run the script and show complete history
        - run-script ./scripts/rolling_rule_a
        - !
        - show-hist
    - Commands can be exported as a script
        - export-script ./test
- Can set a transformation goal using propositions, this will also set the global relation and ensure all commands executed adhere to this relation
    - trans x I x
    - UNIE notifies you when the transformation goal has been reached
- Can traverse back through history if you make a mistake
    - back-hist
- Can clear previous history
    - del-hist <idx>
- Transformations can access the library
    - Term library used as if it was a let x .. in <working term>
        - This is how Jenny uses them in her papers
    - Context library used for CVars, so their definitions can be applied/unapplied etc.
- Can define terms/contexts on the fly
    - define '<name> for terms
    - define $X_NAME[SUB]$ for contexts
    - A calculational style of programming which is needed for our proofs
- WRT contexts/patterns with commands, if the user is prompted for additional information, e.g., when asking UNIE to generate contexts on their behalf, the complete command ultimately executed will be stored in the history for an accurate record
    - Demo this
        - trans $ (let x = 1 in x) x $
        - let-eval
        - unlet-eval
        - show-hist
    - If the command input only has one option, the initial command will be stored even though UNIE may need additional input from the user under different circumstances
    - This has the benefit that we can output scripts without the user needing to choose again in the future when the run it
    - A feature Jenny suggested
- Transformation implementations are much nicer now — see KureCmdSettings file ?

Scripts

- Script mode shows active script and next command
    - run-script ./scripts/rolling_rule_a
    - show-active-script
    - next-cmd
    - prev-cmd
    - <return> to execute next command
    - Can still enter other commands whilst script running
- Actually an active script stack, allowing scripts to import each other and use each other
    - Naive at the minute so can loop infinitely
        - Just need an import check like GHCI
    - When a new active script starts, previous ones halt at where they were, then when it's done, they pick up from where they left off
- Use ! to run all commands in the script stack
- Can even execute ! from within the script <meta-command>
    - I plan to make some basic libraries that will fire when UNIE starts up and designed it for that reason
- Scripts imported into script library
    - show-scripts
- Can be viewed
    - show-script <name>
- show-active-script is my favourite, highlights current command waiting to be executed
- Can export command history as a script
    - export-script ./test
- Again it is all pretty printed

Implementation

- Improvement theory is now a use case for UNIE
    - Nothing about it is a fundamental part of UNIE's implementation
    - Context generation/pattern matching modular
- Everything is built to be extensible in the future
    - All the exportation features are geared towards exporting to Coq/Agda
- Parsers/pretty printers/lexers all stand alone
    - KURE commands are just input into a file, which means users can add their own easily
    - Set of basic commands inbuilt into the system, but could be changed/removed quite easily
- One day I'd like to rename it the Nottingham University Inequational Engine
- Normalisation

        - Allows UNIE to use a richer syntax than Jenny's paper, but can be desugared via normalisation
            - trans $[1..5]$
            - desugar
        - We also have sugared contexts, and UNIE handles them uniformly too - Jenny uses them in her paper (can't remember where off the top of my head)
        - Use context variables to hide detail and allow tactics to be attempted
            - Something I'm really interested in looking at in the future

- Some bits still to add before submission date
    - One or two tick algebra rewrites
    - Assumptions
    - Help files

End note

It's turned out to be such a huge project and it's been an intense learning experience