$U{\cdot}(TP)^2$: Higher-Order Equational Reasoning by Pointing

Andrew Butterfield*
School of Computer Science and Statitics
Trinity College Dublin
Ireland
Andrew.Butterfield@scss.tcd.ie

We describe a prototype theorem prover, $U{\cdot}(TP)^2$, developed to match the style of hand-written proof work in the Unifying Theories of Programming semantical framework. This is based on alphabetised predicates in a 2nd-order logic, with a strong emphasis on equational reasoning. We present here an overview of the user-interface of this prover, which was developed from the outset using a point-and-click approach. We contrast this with the command-line paradigm that continues to dominate the mainstream theorem provers, and raises the question: can we have the best of both worlds?

1 Introduction

Unifying Theories of Programming (UTP) [12], is a framework that uses alphabetised predicates to define language semantics in a relational calculus style, in a way that facilitates the unification of otherwise disjoint semantic theories, either by merging them, or using special linking predicates that form a Galois connection. The framework is designed to cover the spectrum from abstract specifications all the way down to near-machine level descriptions, and as a consequence the notion of refinement plays a key role.

We are doing foundational work in the UTP [12], which requires formal reasoning with not only predicates, but also predicate transformers: $\mathbf{R3}(P) \mathrel{\widehat{=}} II \lhd wait \rhd P$ and predicates over predicates: $P = \mathbf{R3}(P)$. We also need to use recursion at the predicate level: $P \mathrel{\widehat{=}} \mu\, Q \bullet F(Q)$, as well as partially-defined expressions: $s \leqslant s \frown (tr' - tr) \equiv tr \leqslant tr'$. The logic being used is therefore semi-classical (two-valued logic, but expressions may be undefined) and of least 2nd-order. In addition, tool support for foundational work in UTP requires the ability to easily describe new language constructs, which can themselves be treated just like predicates, in keeping with the "programs are predicates" philosophy [11] of UTP. In [6] we gave an overview of the Unifying Theories of Programming Theorem Prover ($U{\cdot}(TP)^2$) that we are developing to support such theory development work[1]. The prover is an interactive tool, with a graphical user-interface, designed to make it easy to define a UTP theory and to experiment and perform the key foundational proofs. The motivation for developing this tool, rather than using an existing one, has been discussed in some detail in [6], but key elements will be reprised here. The logical and technical underpinning was further elaborated upon in [7], which described as being an adapted and generalised version of the equational reasoning system developed by Tourlakis [18], itself inspired by the equational logic of David Gries and his colleagues [10].

In this paper, we describe how the user interacts with this theorem prover, that was developed, *from the outset*, with the proof and reasoning styles typically used in UTP research and published work.

The key emphasis in development was to use window-based GUI techniques early on as the primary mode of interaction, in stark contrast to most modern interactive theorem provers that have essentially a

*This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855

[1] In that paper it was called SAOITHÍN, but the name has since changed to $U{\cdot}(TP)^2$

C. Benzmüller and B. Woltzenlogel Paleo (Eds.):
Eleventh Workshop on User Interfaces for Theorem Provers (UITP 2014)
EPTCS 167, 2014, pp. 14–22, doi:10.4204/EPTCS.167.4

command-line interface (most HOL flavours, CoQ, PVS, ...) sometimes wrapped with a elaborate interface built on top of a highly configurable text editor (e.g., Proof General on Emacs, new Isabelle/HOL interface on top of jEdit).

2 Motivation

There are a lot of theorem provers in existence, of which the most prominent feature in [19]. Of these, the most obvious candidates for consideration for UTP prover support are Isabelle/HOL[14], PVS[15], and CoQ [4]. They are powerful, well-supported, with decades of development experience and large active user communities. They all support higher-order logic of some form, with a command-line interface, typically based around tactics of some form. All three require functions to be total, but support some kind of mechanism for handling partial functions (e.g. dependent types in PVS). Their reasoning frameworks are based on some form of sequent calculus, and do not support equational reasoning in a native fashion.

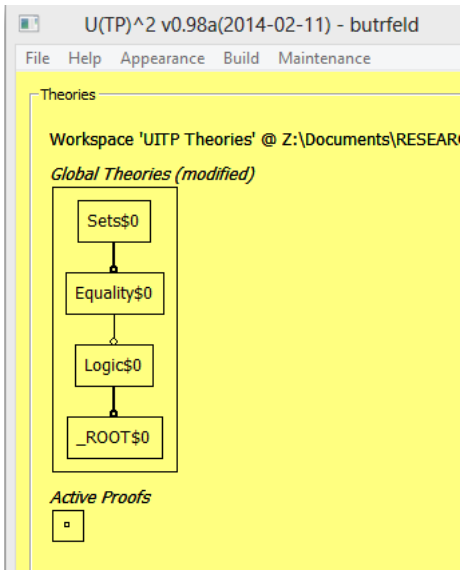
There has been work done on improving the user interfaces of theorem provers of this kind. An interesting example was "proof by pointing" [5] for CoQ which allowed the user to select a subterm, whereupon it would generate and apply a tactic based on the subterm's top-level operator. Whilst proof-by-pointing is not supported in more recent versions of CoQ, it has been incorporated into "Proof General" [1], a general purpose user interface for theorem provers, built on top of Emacs. It supports Isabelle and Coq, among others, and is basically a proof-script management system. In essence it supports the command-line tactics of the provers, allowing the user to edit proof scripts at will, whilst maintaining prover consistency behind the scenes. Other explorations in this area include INKA [13], Lovely OMEGA [16], Window inference [17], Generalized Rewriting in Type Theory [3], The CoRe Calculus, [2] and the Jape Theorem Proving framework (`http://japeforall.org.uk/`.) Of the above, [2],[3] seems designed to support equational reasoning, but lack any notion of a GUI. In [13] and [16] we have GUIs, but the logic/proof style is tree based. The window inference work [17] has a notion of "focus" similar to ours, but has no GUI, and while capable of handling equational rewrites seems to be more general. Jape has a GUI and facilities to encode logics, but again is deduction-biased, and has no easy way to extend the language.

3 Interaction

We shall illustrate $U{\cdot}(TP)^2$'s use by walking through a simple proof, from a theory of sets, regarding the commutativity of set intersection.

We start by launching the theorem prover, and we assume that some theories have been preloaded: `Sets`[2], `Equality`, `Logic` and `_ROOT` (a base theory always present). All theories have access to definitions and laws from lower theories.

[2] The `$0` suffix is a version number



If we double-click on the `Sets` box, a window opens up showing the "Laws" of the Set theory. Laws have names, a "provenance" indicator, side-conditioning, and their defining schema (a predicate).

LAWS | OBS. | LANGUAGE | PRECEDENCE | TYPEdef | CONSTdef | EXPRdef | PREDdef | TYPES | CONJ. | THEOREMS

Laws (Name -+-> SideCond x Pred)

~-in-{}	S	true	~(e1 in {})
in-singleton	S	true	(e1 in {e2}) == (e1 = e2)
in-union	S	true	(e1 in (s1 union s2)) == (e1 in s1) \/ (e1 in s2)
in-intersect	S	true	(e1 in (s1 intsct s2)) == (e1 in s1) /\ (e1 in s2)
in-setdiff	S	true	(e1 in (s1 \ s2)) == (e1 in s1) /\ ~(e1 in s2)
set-extensionality	S	true	(s1 = s2) == (forall x @ (x in s1) == (x in s2))
DEF-subseteq	S	true	(s1 subseteq s2) == (forall x @ (x in s1) => (x in s2))
DEF-subset	S	true	(s1 subset s2) == (s1 subseteq s2) /\ ~(s1 = s2)
DEF-card-empty	S	true	card {} = 0
DEF-card-single	S	true	card {e1} = 1
DEF-card-union	S	true	card (e1 union e2) = (card e1 + card e2) - card (e1 intsct e2)

Clicking on the "CONJ." tab shows some conveniently preloaded conjectures, which have yet to be proven.

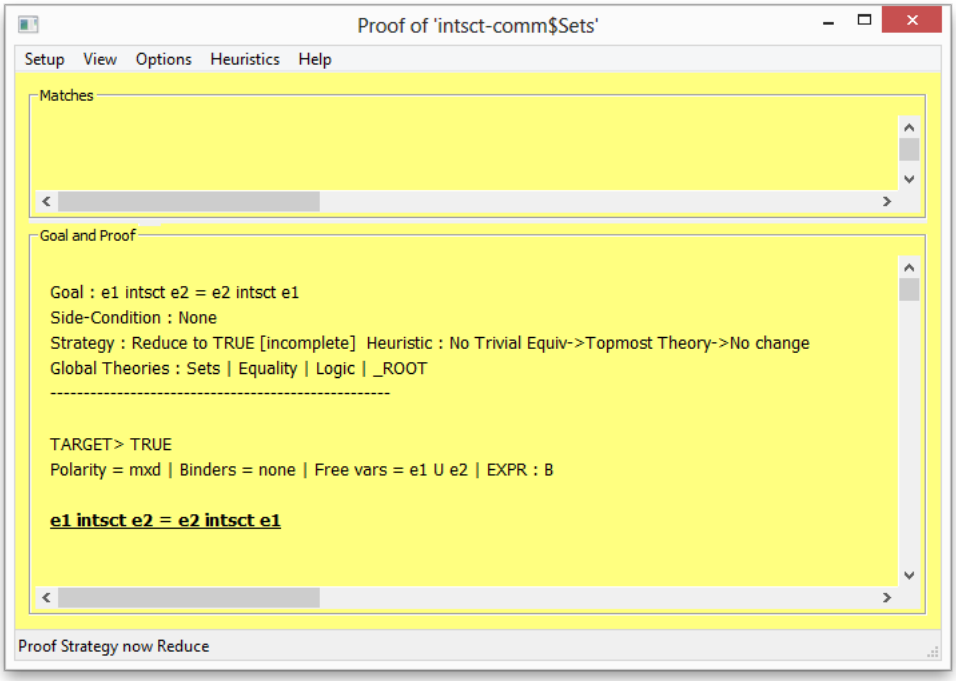
LAWS | OBS. | LANGUAGE | PRECEDENCE | TYPEdef | CONSTdef | EXPRdef | PREDdef | TYPES | CONJ.

Conjectures (Name -+-> SideCond x Pred)

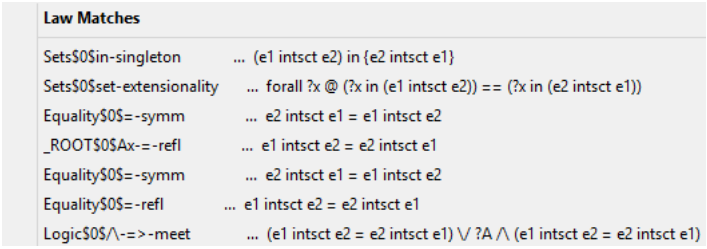
in-self	true	e1 in {e1}
int-sdiff-distr	true	e1 intsct e2 \ e3 = (e1 \ e3) intsct (e2 \ e3)
int-union-distr	true	e1 intsct e2 union e3 = (e1 union e3) intsct (e2 union e3)
intsct-assoc	true	e1 intsct (e2 intsct e3) = (e1 intsct e2) intsct e3
intsct-comm	true	e1 intsct e2 = e2 intsct e1
intsct-idem	true	e1 intsct e1 = e1
sdiff-int-distr	true	e1 \ e2 intsct e3 = (e1 \ e2) union (e1 \ e3)
sdiff-self	true	e1 \ e1 = {}
sdiff-twice	true	(e1 \ e2) \ e3 = e1 \ (e2 union e3)
sdiff-union-distr	true	e1 \ (e2 union e3) = (e1 \ e2) intsct (e1 \ e3)
union-assoc	true	e1 union (e2 union e3) = (e1 union e2) union e3
union-comm	true	e1 union e2 = e2 union e1
union-idem	true	e1 union e1 = e1
union-int-distr	true	(e1 union e2) intsct e3 = e1 intsct e3 union e2 intsct e3
union-sdiff-distr	true	(e1 union e2) \ e3 = (e1 \ e3) union (e2 \ e3)

Double-clicking on the `intsct-comm` row (5th) opens up a proof window, and we use its setup menu to select the "Reduce" strategy, which attempts to transform the goal predicate into TRUE. Other strategies, depending on the goal structure include: "left-to-right", for equality/equivalence conjectures, that converts the lefthand side until equal to the righthand side, or "reduce-both" which tries to transform both sides into some common form.

In the proof window we have the goal and side-conditions displayed, and there is some material about heuristics we ignore in this paper. The lower half of the proof window displays the "TARGET", determined by the goal and the chosen strategy. Some context information is also shown, the most important being the free variables, and the type, in this case, of each side of the equality. We see the starting goal at the bottom, in bold and underlined[3]:



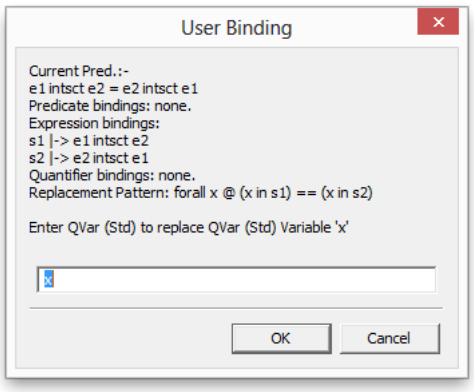
If we right-click anywhere in the "Goal and Proof" subwindow, then a menu of laws applicable to the goal pops-up. In effect the goal was matched against all the laws present in the `Sets`, `Equality`, `Logic` and `_ROOT` theories, the successful matches were then ranked (by various user-selectable heuristics), the top twenty chosen, then applied to the goal to show the result, and presented in the menu.



If we pick the second, "set-extensionality", as it has new variables not present in the goal, (e.g. `?x`), we are asked to supply instances for these, with a reasonable default being offered. This feature is not obviously useful in this example (except if `x` was present elsewhere) but comes in handy when matching

[3]The "Matches" subwindow will not be discussed here

the rhs of a law like $A \vee (A \wedge B) \equiv A$, to get the rhs, in which we are free to instantiate B as we see fit.



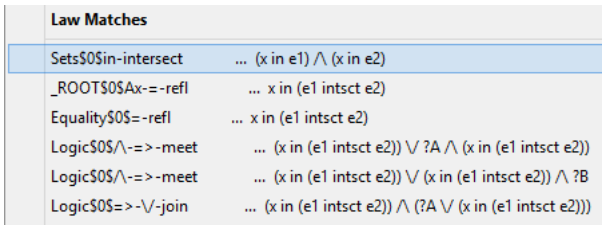
If we go with the default suggestion, then we obtain the following proof state:

```
TARGET> TRUE
Polarity = +ve | Binders = none | Free vars = e1 U e2 | PREDICATE

forall x @ (x in (e1 intsct e2)) == (x in (e2 intsct e1))

===  "set-extensionality (L-to-R) @"
   e1 intsct e2 = e2 intsct e1
```

We can use arrow-keys to move around the goal, changing the proof "focus". If we go "down" twice, we focus in on the first set membership assertion. It is worth noting that the line above records that the focus is on an expression (EXPR) of type boolean (B). $U{\cdot}(TP)^2$ has a on-the-fly type inference algorithm that runs every time the focus changes[4], and is used by the law matching algorithm to avoid spurious matches. We avoid lots of explicit type annotations, preferring to deal with such issues behind the scenes. This is of course in keeping with the general traditional UTP approach to theorem development.

```
TARGET> TRUE
Polarity = mxd | Binders = x | Free vars = e1 U e2 U x | EXPR : B

forall x @ (x in (e1 intsct e2)) == (x in (e2 intsct e1))

===  "set-extensionality (L-to-R) @"
   e1 intsct e2 = e2 intsct e1
```

Right-clicking now leads to laws relevant to the focus:

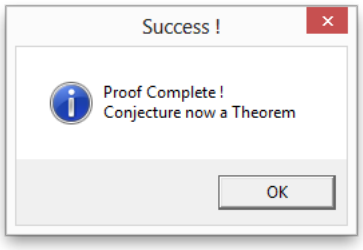


If we pick the first option, then we get a conjunction of simpler membership statements.

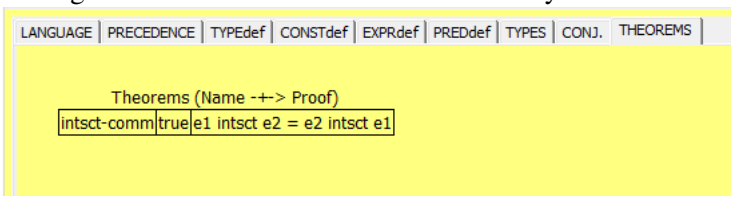
[4] speed has never been a problem with this

```
TARGET> TRUE
Polarity = mxd | Binders = x | Free vars = e1 U e2 U x | PREDICATE

forall x @ (x in e1) /\ (x in e2) == (x in (e2 intsct e1))

===  "in-intersect (L-to-R) @1.1"
   forall x @ (x in (e1 intsct e2)) == (x in (e2 intsct e1))
===  "set-extensionality (L-to-R) @"
```

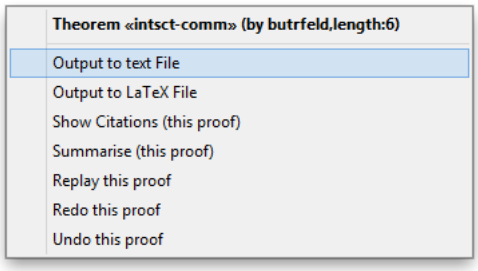
Moving to the righthand side of the equality, we can apply the same `in-intersect` law, then apply the commutativity of conjunction, pull back out and we get instances of the reflexivity of equals. Finally we get rid of a vacuous quantifier, so resulting in the goal `TRUE`, and $U \cdot (TP)^2$ proclaims!



Examining the "THEOREMS" tab in the Set theory window shows our new theorem.



Right-clicking on it gives another pop-up menu of interesting things to do with it.



We render a simple text version of the resulting proof:

```
Complete Proof for 'Sets$intsct-comm
Goal : e1 intsct e2 = e2 intsct e1
Strategy: Reduce to TRUE

     e1 intsct e2 = e2 intsct e1
 ===   " set-extensionality (L-to-R) @ "
     forall x @ (x in (e1 intsct e2)) == (x in (e2 intsct e1))
 ===   " in-intersect (L-to-R) @1.1 "
     forall x @ (x in e1) /\ (x in e2) == (x in (e2 intsct e1))
 ===   " in-intersect (L-to-R) @1.2 "
     forall x @ (x in e1) /\ (x in e2) == (x in e2) /\ (x in e1)
 ===   " /\-comm (R-to-L) @1.2 "
```

```
    forall x @ (x in e1) /\ (x in e2) == (x in e1) /\ (x in e2)
===   " Ax-==-id (R-to-L) @1 "
    forall x @ TRUE
===   " forall-vac (L-to-R) @ "
    TRUE
```

We have only skimmed over the interactive proof features of $U{\cdot}(TP)^2$ here. Others include

- keyboard shortcuts to apply built-in procedures to the focus, e.g., convert to disjunctive normal form
- a clickable help feature in the proof window
- strategies to support inductive proofs
- all tables in each tab of each theory can be edited, with entries added or deleted — even laws!
- Some of the tabs, ("OBS.","LANGUAGE","PRECEDENCE") have tables that support user definitions of languages. See [7] for further details.

4 Discussion

Proofs done with $U{\cdot}(TP)^2$ are, in our opinion, more "open", in that we can easily see the steps and laws used in a proof, in an equational style. A consequence of this is readily seen when we consider the students taking the Formal Methods course offered at Trinity College Dublin, that focusses on the UTP, and uses $U{\cdot}(TP)^2$ for part of the classwork. The feedback obtained from these students shows clearly that (i) the learning curve to get good at $U{\cdot}(TP)^2$ proofs is fairly shallow—they almost never get "stuck", once a few tricks are shown—experimentation is easy; (ii) their concerns are regarding improvement to the GUI itself, either in terms of how it looks, or having the flexibility to define their own keyboard shortcuts. A key feature that reduces the learning curve is the ability of the prover to suggest possible next steps, by doing advance pattern-matching and instantiation.

Proofs in CoQ or Isabelle/HOL are, again in this authors words, more "procedural", and "opaque", but definitely more powerful. The disadvantage is that the learning curve is much steeper, particularly when early success it obtained by tactics like `auto`, `simp` or `sledgehammer`. When these fail, the best approach is not so clear to the beginner. However, there is undeniable power once that learning curve has been climbed.

$U{\cdot}(TP)^2$ was really developed to assist in the development of new semantic theories within the UTP framework. Others have also put effort into doing this for UTP using both ProofPowerZ[20] and Isabelle/HOL[8, 9]. The price they pay is having to recast material in the ProofPower/HL style. The benefit they tap into is the power of their proof engines.

The key questions raised here are:

- Should point-n-click GUIs be added to existing provers?
- To what extent are front-ends like Proof-General or jEdit are step in this direction?
- Should more attention be paid to developing equational reasoning approaches?
- Can the $U{\cdot}(TP)^2$ front-end be fruitfully turned into a wrapper around Isabelle/HOL say?
- Should it use Isabelle/HOL as a way to check its proofs (would save trying to develop a small safe LCF-style kernel for $U{\cdot}(TP)^2$) ?

- Can we envisage proofs been done using gestures on a tablet?

Very recent work, presented as Tutorial 2 at FM2014 in Singapore, by Jim Woodcock, Simon Foster and Frank Zeyda of the University of York, showed an encoding of UTP and some key theories into Isabelle/HOL. One the negative side, they had to employ further nested quotation schemes, but on the positive side, they used Isar in such a way that it may be relatively easy to use Isabelle/HOL to check proof steps made by $U{\cdot}(TP)^2$. We hope to explore this connection in the near future.

4.1 Obtaining Code

$U{\cdot}(TP)^2$ is written in Haskell using the wxHaskell GUI library, and is available open-source, currently under a GPL v2 license, from `https://bitbucket.org/andrewbutterfield/saoithin`. The screenshots in this paper were produced using version 0.98a.

References

[1] David Aspinall (2000): *Proof General: A Generic Tool for Proof Development*. In Susanne Graf & Michael I. Schwartzbach, editors: *TACAS*, *LNCS* 1785, Springer, pp. 38–42. Available at `http://dx.doi.org/10.1007/3-540-46419-0_3`.

[2] Serge Autexier (2005): *The CoRe Calculus*. In Robert Nieuwenhuis, editor: *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings*, *Lecture Notes in Computer Science* 3632, Springer, pp. 84–98. Available at `http://dx.doi.org/10.1007/11532231_7`.

[3] David A. Basin (1994): *Generalized Rewriting in Type Theory*. *Elektronische Informationsverarbeitung und Kybernetik* 30(5/6), pp. 249–259. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.3314`.

[4] Yves Bertot & P. (Pierre) Castéran (2004): *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Texts in theoretical computer science, Springer Verlag, doi:10.1007/978-3-662-07964-5.

[5] Yves Bertot, Gilles Kahn & Laurent Théry (1994): *Proof by Pointing*. In Masami Hagiya & John C. Mitchell, editors: *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, *LNCS* 789, Springer, pp. 141–160. Available at `http://dx.doi.org/10.1007/3-540-57887-0_94`.

[6] Andrew Butterfield (2010): *Saoithin: A Theorem Prover for UTP*. In Shenchao Qin, editor: *Unifying Theories of Programming, Third International Symposium, UTP 2010, Shanghai, China, November, 2010.*, *LNCS* 6445, Springer, Shanghai, China, pp. 137–156. Available at `http://dx.doi.org/10.1007/978-3-642-16690-7_6`.

[7] Andrew Butterfield (2012): *The Logic of $U{\cdot}(TP)^2$*. pp. 124–143. Available at `http://dx.doi.org/10.1007/978-3-642-35705-3_6`.

[8] Abderrahmane Feliachi, Marie-Claude Gaudel & Burkhart Wolff (2012): *Isabelle/Circus: A Process Specification and Verification Environment*. In Rajeev Joshi, Peter Müller & Andreas Podelski, editors: *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, *Lecture Notes in Computer Science* 7152, Springer, pp. 243–260. Available at `http://dx.doi.org/10.1007/978-3-642-27705-4_20`.

[9] Simon Foster, Frank Zeyda & Jim Woodcock (to appear): *Isabelle/UTP: A mechanised theory engineering framework*. In David Naumann, editor: *Unifying Theories of Programming, Fifth International Symposium, UTP 2014, National University of Singapore, May 13, 2014,*.

[10] David Gries & Fred B. Schneider (1993): *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science, Berlin: Springer Verlag, doi:10.1007/978-1-4757-3837-7.

[11] Eric C. R. Hehner (1984): *Predicative programming — Part I & II*. *Commun. ACM* 27(2), pp. 134–151, doi:10.1145/69610.357990.

[12] C. A. R. Hoare & Jifeng He (1998): *Unifying Theories of Programming*. Prentice-Hall.

[13] Dieter Hutter & Claus Sengler (1996): *The Graphical User Interface of INKA*. In Nicholas A. Merriam, editor: *Proceedings International Workshop on User Interfaces for Theorem Provers*, N. Merriam, pp. 43–50. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.9511`.

[14] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. *Lecture Notes in Computer Science* 2283, Springer. Available at `http://www.springer.com/computer/theoretical+computer+science/book/978-3-540-43376-7`.

[15] Natarajan Shankar (1996): *PVS: Combining Specification, Proof Checking, and Model Checking*. In Mandayam K. Srivas & Albert John Camilleri, editors: *Formal Methods in Computer-Aided Design, First International Conference, FMCAD '96*, *LNCS* 1166, Springer, pp. 257–264, doi:10.1007/BFb0031813.

[16] Jörg Siekmann, Stephan Hess, Christoph Benzmüller, Lassaad Cheikhrouhou, Armin Fiedler, Helmut Horacek, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Martin Pollet & Volker Sorge (1999): *LOmegaUI: Lovely OMEGA User Interface*. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.1864`.

[17] Mark Staples (1995): *Window Inference in Isabelle*. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.104.6012`.

[18] George Tourlakis (2001): *On the Soundness and Completeness of Equational Predicate Logics*. *J. Log. Comput.* 11(4), pp. 623–653, doi:10.1093/logcom/11.4.623. Available at `http://www3.oup.co.uk/logcom/hdb/Volume_11/Issue_04/110623.sgm.abs.html`.

[19] Freek Wiedijk, editor (2006): *The Seventeen Provers of the World, Foreword by Dana S. Scott*. *Lecture Notes in Computer Science* 3600, Springer. Available at `http://www.springer.com/computer/ai/book/978-3-540-30704-4`.

[20] Frank Zeyda & Ana Cavalcanti (2008): *Encoding Circus Programs in ProofPower-Z*. In Andrew Butterfield, editor: *Unifying Theories of Programming, Second International Symposium, UTP 2008, Trinity College, Dublin, Ireland, September 8-10, 2008, Revised Selected Papers*, *Lecture Notes in Computer Science* 5713, Springer, pp. 218–237, doi:10.1007/978-3-642-14521-6_13.