Student Proof Exercises using MathsTiles and Isabelle/HOL in an Intelligent Book

William Billingsley (wbillingsley@cantab.net) and Peter Robinson (peter.robinson@cl.cam.ac.uk)
University of Cambridge

Abstract.

The Intelligent Book project aims to improve online education by designing materials that can model the subject matter they teach, in the manner of a Reactive Learning Environment. In this paper, we investigate using an automated proof assistant, particularly Isabelle/HOL, as the model supporting first year undergraduate exercises in which students write proofs in number theory. Automated proof assistants are generally considered to be difficult for novices to learn. We examine whether, by providing a very specialised interface, it is possible to build something that is usable enough to be of educational value. To ensure students cannot "game the system" the exercise avoids tactic-choosing interaction styles, but asks the student to write out the proof. Proofs are written using MathsTiles: composable tiles that resemble written mathematics. Unlike traditional syntax-directed editors, MathsTiles allow students to keep many answer fragments on the canvas at the same time, and do not constrain the order in which an answer is written. Also, the tile syntax does not need to match the underlying Isar syntax exactly, and different tiles can be used for different questions. The exercises take place within the context of an Intelligent Book. We performed a user study and qualitative analysis of the system. Some users were able to complete proofs with much less training than is usual for the automated proof assistant itself, but there remain significant usability issues to overcome.

Keywords: Intelligent Book, MathsTiles, Isabelle

1. Introduction

The Intelligent Book project aims to improve online education by designing materials that can model the subject matter they teach. For example, a question we designed for electronics asks students to choose current, voltage, and component values for a transistor amplifier, in order to meet a set of specifications [12, 35]. The question is backed by an AI model of the circuit based on a *constraint propagator* [39]. Whenever a student sets a value in the circuit, this constraint propagator uses the rules of electronics to deduce what other values in the circuit need to be. If a student's choices are inconsistent or do not meet the specifications, the question can play back the sequence of deductions in the model, to explain to the student exactly what the problem with his or her choices is.

It is important to note that unlike *intelligent tutoring systems* such as Andes [18] and *intelligent learning environments* such as ActiveMath [30], our exercises generally do not keep a student model measuring the students' understanding of particular domain rules or the solution steps they should take to answer a question. This is because as a research objective we wish to support questions where students are not taught the exact process used to answer the question (for example higher mental process questions), and questions where there may not be a general process that solves all problems in the domain. In these cases, there are no known solution steps to model students against, and it is not the students' knowledge of basic domain rules that is being tested. Instead, our questions are designed as *reactive learning environments* [15]. They give students the freedom and opportunity to try out their ideas, and use the AI model of the material to explain any consequences or problems with their answers.

The question arises, what kind of model could be used to support proof exercises for a first year undergraduate discrete mathematics course, covering introductory number theory? One possible choice, that we have chosen to investigate in this paper, is an automated proof assistant. These have been developed over many years to model and support the proofs of researchers and professionals. However, they are generally regarded to be difficult for novices to learn to use. From their experiences teaching postgraduates how to use the HOL system, Slind *et al* [37] found interactive proof assistants to be "powerful but bewildering". They identified general reasons for this, including: "simply managing to formulate correct statements can be difficult"; "finding the correct tool to use at any point can be hard"; and "even remembering how to look for existing theorems to use can be hard". Isabelle/HOL [34], the proof assistant we use in this paper, is similarly complex. The two shortest introduction courses to Isabelle/HOL [33, 6], presented to research audiences, each take four sessions of 90 minutes and each include more than 300 slides. We asked informally on the Isabelle/HOL users' mailing list how long it might take a first year undergraduate to learn to use the system well enough to answer induction or case proofs on the Fibonacci sequence (using an example from our evaluation study in this paper). The rough estimate we received from an experienced user was that if we offered a taught course in how to use the system then students "could do simple things within ten weeks" and "it might take as long as twenty weeks for an average student to become proficient at the level you are suggesting". We were also warned that if students could not already write a proof for a theorem on paper, they would not be able to prove it in Isabelle/HOL. In our design-stage discussions with experienced users of HOL and Isabelle/HOL, we were also warned that the reasoning output of most proof assistants is very low level and would be difficult for students to follow.

Our hypothesis in this paper, however, is that by using a very specialised interface to the proof exercises we can provide something of educational value that students can learn to use much more quickly and with much less training. We have three reasons for believing that this might be the case:

- The interfaces of proof assistants appear, by and large, to have been optimised for experienced users who work with the system regularly, rather than for novice users. There are many techniques in human computer interaction research that can reduce the learning barrier for first time users – for example structured editing [40, 3] can help novices to work with a new syntax, but can be cumbersome for more experienced users [24].

- Answering a homework proof exercise is a different situation from attempting a proof in professional practice, because in a homework exercise the proof has been set by a teacher. The teacher has the opportunity to look at the question in advance and make alterations to ensure that an answer is achievable by students.

- Our experience with the electronics question, described earlier, suggested it is possible to relate AI-generated reasoning to a student's level of detail. In the electronics questions, initially the constraint propagator output explanations that (anecdotally) were too detailed and low level for students to understand. We found that a successful approach to solving this was to define the circuit diagram students would interact with separately from the AI model of the circuit. The explanations were then automatically

> pruned so that only steps involving variables that were marked on the diagram were included. The principle here is that if the user interface is designed to represent the student's model of the question, then by mapping the reasoning onto that interface we are mapping it to a student's level of detail.

We constructed a set of six proof exercises that take place within the context of an *Intelligent Book*. This is a web-based textbook that as well as containing AI-supported exercises also contains extensible and adaptive content (students can add content to the book, and different explanations can be shown to different students for the same topic). The students' proofs are checked using the Isabelle/HOL system, but are written with our own specialised interface. We performed a qualitative evaluation on the exercises in two ways: by asking students and others with no automated proof experience to attempt the exercises, and by using one of the foremost analytical techniqes in human computer interaction.

There are two contributions in this paper. First, although we find there are many usability issues still to overcome, the exercises do represent an advance in enabling untrained students to write verifiable proofs in a system where the student must write the lines of proof (rather than asking the automated proof assistant to apply tactics to manipulate goal statements). There are many systems that ask students to write simple proofs in simpler domains such as predicate logic [28, 29], but this is the first web-based learning environment to ask students to write proofs in this manner for number theory. The second contribution is the results from the qualitative usability study we conducted. This study revealed a number of issues that are relevant for future design of educational proof interfaces.

The paper is organised as follows. Section 2 describes a number of design constraints on the exercises. From these we explain how the major design decisions in our exercises were made. Section 3 provides a summary of the architecture of Intelligent Book questions. Section 4 describes MathsTiles, which is a configurable structured interaction language we designed, and which we use as the proof language. Section 5 describes precisely how MathsTiles is used as a proof language. Section 6 walks through an example exercise. Section 7 describes how we have evaluated the system and presents the results of that evaluation. Section 8 suggests a number of avenues for future research. Section 9 describes related work by other researchers. Finally, Section 10 concludes the paper.

2. Design Goals

In this section, we describe three design goals of the exercises, and how those drove our design decisions. During development we made compromises on the second and third goals, as described in Sections 5 and 7, but nonetheless they were important to the design.

2.1. The exercises take place within an Intelligent Book

The first goal is part of our wider research aim, which is to develop an Intelligent Book. The concept of an Intelligent Book is a web-based textbook that contains both AI-supported exercises and content. Students can add content to the book and alter existing content within it, and the advice from the exercise AI systems can refer to that content. The proof exercises in this paper are designed as Intelligent Book exercises. While the architecture of the Intelligent Book is not a focus of this paper, relevant aspects of it are summarised in Section 3.

2.2. The student, not the system, should write the proof

Many proof assistants do not ask the user to write each line of the proof. Instead the user works by asking the assistant to apply tactics to statements on a goal stack. These tactics eliminate goals or produce new goal statements, until all the goals have been proved. It would be tempting, from a human computer interaction perspective, to use a similar mechanism in the exercises. This way the student would not have to learn the prover's expression syntax (not even for the overall proof goal, which would be set by the question) but could focus on applying the appropriate tactics. However, this would also enable students to "game the system" by rapidly trying each tactic in turn, rather than actively thinking about the problem. This behaviour has been observed in a number of educational settings and correlates strongly with reduced learning outcomes [5].

Instead we decided students should the write the statements and expressions for each line of their proofs, as they do when answering proof exercises on paper, rather than have them generated by the system. This means that to use a tactic, the student has to think about what it will produce. So, the students' investment at each step is much greater and there is less scope for gaming the system.

For this reason, we chose Isabelle/HOL to act as the model. It's Isar proof language [42, 32] supports "declarative" proofs that are somewhat similar to written proofs, rather than only supporting tactic scripts.

2.3. Proofs should resemble what students write on paper

While structured and menu-based editors have been known to reduce the learning burden of a new syntax (keywords and syntax rules can be recognised rather than recalled), this alone is unlikely to make Isabelle/HOL approachable for students with no experience of programming or proof. Isabelle/HOL contains both an inner HOL syntax and an outer Isar syntax. The outer Isar syntax contains keywords that appear similar in meaning but have very different effects. Fox example, the difference between the keywords `hence`, `thus`, `then`, `also`, and `moreover` is not readily apparant from the words themselves. There are also occasions where the same concept can be applied either at the Isabelle level or at the HOL level, for example whether the mathematical declaration *for all* is made using `!!` or `ALL`, and this decision will affect later proof commands.

Also, as described in the introduction, we would like the user interface to represent a "students' model" of the question rather than the AI model. In this case, we decided that the statements students make in questions should more closely resemble statements they might make on paper, rather than mimicking the Isar language exactly. (That is not to say, however, that they will look identical to written proofs.)

A related point is that when students write proofs on paper, they do not always take the strictly top-down approach that traditional structured editors encourage. We do not want the interface to force them into that approach. As an example, it would be very unusual for a student writing an algebraic expression on paper to write the symbols in the hierarchical order of the expression's syntax tree. Students may wish to start in the middle of the expression, or may wish to sketch out parts of the expression and then link them up. The interface should make some attempt to support this.

3. Intelligent Book Architecture

In this section, we describe the parts of the Intelligent Book's question architecture and content model that are relevant to the proof exercises.

3.1. Question Architecture

When students are working on questions in an Intelligent Book, they should be able to use the appropriate diagrams and notational forms for the subject matter. For example, students working on an electronics question may need to work with a circuit diagram. However, other questions in the same book might involve timing diagrams or state charts. The Intelligent Book question architecture is designed to support questions with different diagram and notational forms, different models, and different teaching pedagogies, in the same overall structure. Figure 1 shows the architecture of an Intelligent Book exercise.

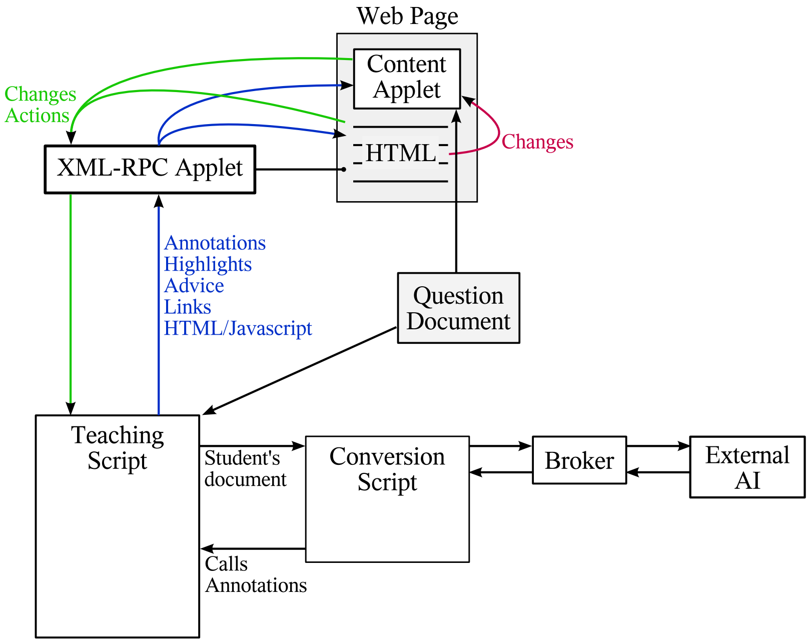


Figure 1. The architecture of an Intelligent Book question.

Question document. Because we want the server to be able to comment on the student's work progressively, we consider the student to be working on a remote document on the server, rather than preparing an answer on the client that will be sent to the server when a submit button is pressed. The question document describes the question – the initial state of the students' documents. When students begin work, they get their own copy of this document, and the actions they take to answer the question make modifications to it. Question documents are written in XML; the schema varies according to the diagram format. By convention, question documents are modular in that they can include other documents. For example a circuit diagram document may include a library document that describes what resistors should look like and what properties they have; the student's document would then describe particular instances of resistors on the page. (It is possible to have more than one diagram in a question, but the proof exercises do not do so.)

Web page. The exercises take place within a web page. Although it is impractical to implement the diagrams that students work with in HTML, when the system makes text comments about the diagram or links to related content, we prefer to use the full HTML capabilities of the browser rather than a limited HTML component included in an applet. To support this, the HTML on the page is dynamically updated from the server. Some of the links on the page call functions on the server. For example, in the proof exercises, the *check proof* link calls a function on the server. The system can also place Javascript-backed controls on the page that when used will make alterations to the student's document at both the client and the server. (For example, the server can make a suggestion of how to fix a mistake and offer a link that will make the change automatically.)

Content applets support diagrams. They allow graphical interaction with the student's document. Their internal datamodel is a document object model of the student's XML document. Content applets are usually built around a common internal architecture that makes them easy to construct [11]. Whenever the student makes a change to the document, they report this to the server. They also support a common set of API functions that allows the server to alter the document at the client. Usually, however, the server does not alter the document itself but highlights or annotates elements in the document, and provides Javascript links that when clicked will call the API to alter the document. (Users might find it frustrating if the server made an alteration unexpectedly.) Content applets can also expose any other methods they wish, and these are immediately callable from the teaching script on the server without changes to any of the components in between.

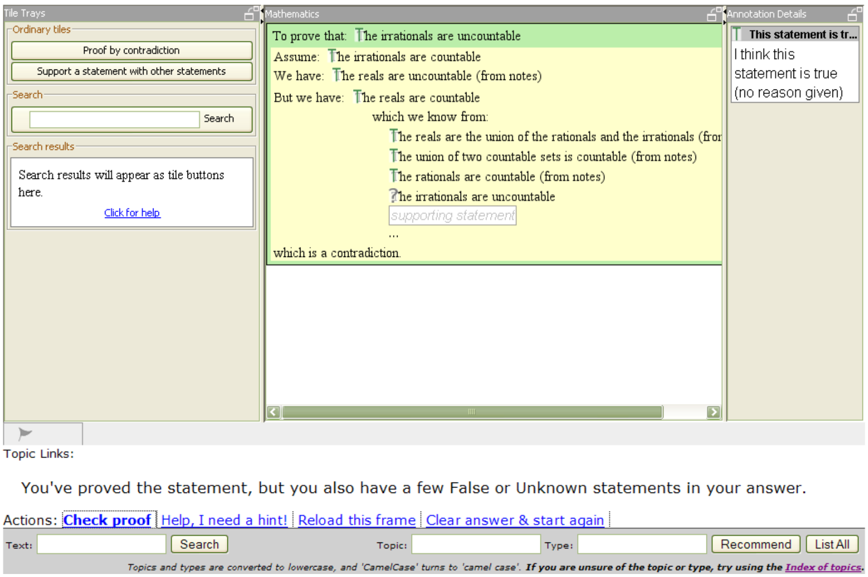
XML-RPC applet. Communication between the client and the server happens over XML-RPC [45] via the XML-RPC Applet. All calls are initiated from the client. However, calls to the server expect the returned data to be the set of method calls the server would like to make to the client in response (a list of calls, marked up in XML-RPC's XML schema). The XML-RPC applet also contains a set of methods that, when called, can make alterations to the HTML of the web page on behalf of the server.

Teaching script. Each question in an Intelligent Book is supported by a teaching script that describes how to respond to the student. The XML-RPC calls the client makes are all to the teaching script. Teaching scripts are Java classes, which means they support inheritance. A superclass provides the implementation for updating the student's question document, managing conversion scripts, storing annotations from the external AI, and support for adaptive advice functions as described in Section 3.2. It also provides hooks to analyse the student's document and the change the student is making both before and after the change is applied. Teaching scripts can also include any other methods they wish, and these are immediately callable from the web page or the content applet without changes to any of the components in between. Usually a subclass is defined for a particular kind of question, for example proof exercises, and that is subclassed again to provide the teaching scripts for particular questions.

Conversion scripts are responsible for processing the student's document and inputting it to the external AI, via the broker. Like XSLT [16] (the most common conversion technology applied to XML documents), the conversion scripts work by associating patterns with templates. The pattern matches a part of the source document, and the template describes what to produce for that pattern. However, while XSLT is primarily designed to transform an XML document into another XML document, in an Intelligent Book we generally need to transform an XML document into a set of procedural actions. So, our conversion scripts, rather than being written in an XML dialect, are written in Groovy (a scripting language that interoperates well with Java) and the template is a Groovy closure of actions to perform. Conversion scripts are modular, in that they can include and extend other conversion scripts.

Broker. A broker is needed where the external AI is a process rather than a module. The broker keeps a pool of processes ready to handle requests. In the proof exercises, where Isabelle/HOL is the external AI, the conversion scripts make frequent calls to write to PGIP-formatted [4] proof commands to the broker's buffer. When asked, the broker writes the contents of the buffer to Isabelle/HOL and collects the responses. This happens regularly throughout the document, rather than only at the end. The responses are post-processed in the broker, and passed back to the Conversion Script as annotations. Once the conversion script has finished, that Isabelle/HOL process is reset for the next request rather than kept in its current state. This means that if Isabelle/HOL identifies an error, the conversion script should take action to collect any context information it needs before it exits. (Otherwise when the student asks for advice, the teaching script will need to re-run the conversion process in order to analyse Isabelle/HOL's state any further.) The collected annotations are stored along with the student's document, so later calls to the teaching script can refer to them.

If content applets and brokers are written well, then content applets, brokers, and external AIs can often be reused across different kinds of question. For example, Figure 2 shows a question that uses informal modelling rather than Isabelle/HOL. This uses a different conversion script (that includes its own modelling), but the same content applet as the exercises in this paper. Figure 3 shows a proof exercise that uses Isabelle/HOL's native syntax. This uses a different content applet and conversion script, but the same broker and external AI as the proof exercises in this paper.



Search results will appear as tile buttons here.

To prove that: The irrationals are uncountable
Assume: The irrationals are countable
We have: The reals are uncountable (from notes)
But we have: The reals are countable
which we know from:
The reals are the union of the rationals and the irrationals (fror
The union of two countable sets is countable (from notes)
The rationals are countable (from notes)
?he irrationals are uncountable
supporting statement
...
which is a contradiction.

I think this statement is true (no reason given)

You've proved the statement, but you also have a few False or Unknown statements in your answer.

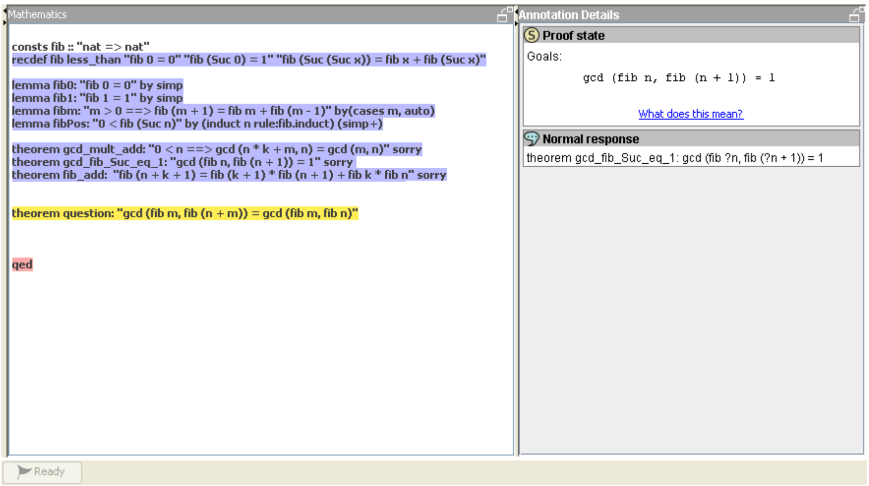
Actions: **Check proof** | Help, I need a hint! | Reload this frame | Clear answer & start again

Topics and types are converted to lowercase, and 'CamelCase' turns to 'camel case'. ***If you are unsure of the topic or type, try using the Index of topics.***

Figure 2. An informally modelled question that uses MathsTiles but does not use Isabelle/HOL

In practice, the different conversion scripts tend to have a broadly similar structure (and brokers, teaching scripts, and content applets similarly have their own common structures), so writing a new kind of question can be less effort than it might appear from Figure 1.

The electronics question mentioned in the introduction used the same question architecture at the client, but predates the server architecture.



```
consts fib :: "nat => nat"
recdef fib less_than "fib 0 = 0" "fib (Suc 0) = 1" "fib (Suc (Suc x)) = fib x + fib (Suc x)"

lemma fib0: "fib 0 = 0" by simp
lemma fib1: "fib 1 = 1" by simp
lemma fibm: "m > 0 ==> fib (m + 1) = fib m + fib (m - 1)" by(cases m, auto)
lemma fibPos: "0 < fib (Suc n)" by (induct n rule:fib.induct) (simp+)

theorem gcd_mult_add: "0 < n ==> gcd (n * k + m, n) = gcd (m, n)" sorry
theorem gcd_fib_Suc_eq_1: "gcd (fib n, fib (n + 1)) = 1" sorry
theorem fib_add:  "fib (n + k + 1) = fib (k + 1) * fib (n + 1) + fib k * fib n" sorry


theorem question: "gcd (fib m, fib (n + m)) = gcd (fib m, fib n)"


qed
```

```
Goals:
        gcd (fib n, fib (n + 1)) = 1
```

```
theorem gcd_fib_Suc_eq_1: gcd (fib ?n, fib (?n + 1)) = 1
```

Figure 3. A proof exercise that uses a native Isar Content Applet, but that uses the same Broker to talk to Isabelle/HOL.

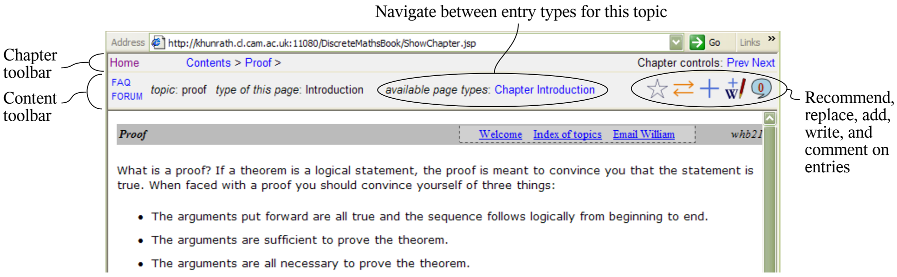
3.2. Content Model

A number of online textbook systems [30, 8] take a strict ontological or semantic approach to content, such as OMDoc [26]. However, a strict ontology could pose a barrier to students wishing to add or edit content within the book. They would need to know the ontology in detail in order to fit their entry within it, and as students they presumably do not have this expertise. The approach we have taken is to use an informal topic model that lets readers add alternative entries for topics, or even alternative chapters, more easily, but that still allows the automated advice from the teaching scripts to link to the content.

Content in an Intelligent Book is classified by topic and by the type of entry. For example, a student could ask for an *introduction* to *mathematical induction* and then ask for an *example*. Figure 5 shows part of the alphabetical topic index of a book. There may be more than one induction example in the book, in which case server scripts choose an appropriate one to show.

Figure 4 shows a page of content within the book. Using the controls above the content, students can navigate between different types of content for the same topic. They can also recommend the current entry for the topic, ask for a different entry for this topic instead (a list of all the available pages for that topic are shown), add an entry they have found on the web, or write their own wiki entry for the topic. The wiki stores more than one page per topic, so more than one student can write a wiki entry, but users can still edit each others'.

The topmost toolbar in Figure 4 is an ordered list of subtopics. A chapter is defined by including its structure in a hidden field on its contents page. Theoretically, this makes it possible for students to write alternative chapters in the wiki as well as alternative pages, but we have not so far allowed students to write chapter structures in the wiki markup interface.


Navigate between entry types for this topic
Chapter toolbar
Content toolbar
Recommend, replace, add, write, and comment on entries
Address
http://khunrath.cl.cam.ac.uk:11080/DiscreteMathsBook/ShowChapter.jsp
Go
Links »
Home
Contents > Proof >
Chapter controls: Prev Next
FAQ
FORUM
topic: proof
type of this page: Introduction
available page types: Chapter Introduction
Welcome
Index of topics
Email William
whb21

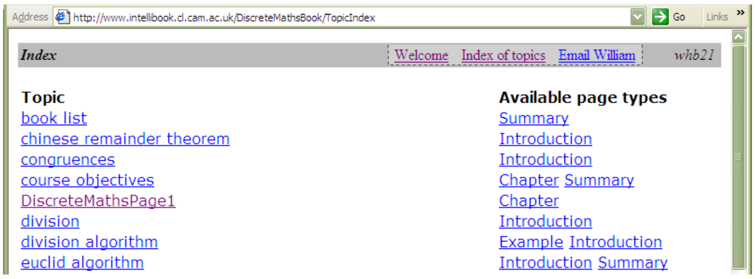
Proof

What is a proof? If a theorem is a logical statement, the proof is meant to convince you that the statement is true. When faced with a proof you should convince yourself of three things:

- The arguments put forward are all true and the sequence follows logically from beginning to end.
- The arguments are sufficient to prove the theorem.
- The arguments are all necessary to prove the theorem.

Figure 4. A content toolbar allows students to navigate between different types of entry for a topic, to recommend an entry, choose an alternate entry, add or write their own entry, or comment on the existing entries. A chapter toolbar allows students to navigate the topics of a chapter in an ordered manner.

The entry types allowed in the book are limited so that students cannot add so many different types of entry for a topic as to be unnavigable. The currently allowed types are: *Introduction*, *Summary*, *Example*, *Exercise*, *Exercise Advice*, *Chapter*, and *Search*.


Address
http://www.intellibook.cl.cam.ac.uk/DiscreteMathsBook/TopicIndex
Go
Links »
Index
Welcome
Index of topics
Email William
whb21

Topic	Available page types
book list	Summary
chinese remainder theorem	Introduction
congruences	Introduction
course objectives	Chapter Summary
DiscreteMathsPage1	Chapter
division	Introduction
division algorithm	Example Introduction
euclid algorithm	Introduction Summary

Figure 5. Pages in an Intelligent Book are classified by topic and type. This is the alphabetical index page, and lists the topics and the available page types for each of those topics; there may be more one entry per topic-type combination, in which case selection scripts choose which one to show.

When students ask for help with a topic, they are not always asking for content. They may be stuck with a particular error in their exercise and be asking for analytical help examining how to get past it. It can sometimes be useful to include very *ad hoc* analysis that relies on knowing how students are expected to answer the question. However, it is not always possible for the teacher to know whether this will be useful at the time the advice is written.

To support this, we allow *advice functions* to be associated with topic keys in questions' teaching scripts. They are also associated with a *relevance function*. When the student asks, the teaching script will attempt to choose an advice function for the topic that is relevant to the current situation. If there is more than one relevant advice function, then the teaching script chooses a function based on which have been found to be useful in the past. (When the advice is presented, the student is asked to say whether or not it was helpful.) There is no restriction on what these relevance functions can do.

4. MathsTiles

4.1. Overview

MathsTiles is an interface for students to edit structured content, such as mathematical equations and proofs, within web pages. The syntax is not fixed but is configurable from question to question. This means that MathsTiles is not itself a formal or semantic language for mathematics, but is a structured interaction language designed so that the constructed mathematics can easily be transformed into other representations (including formal and semantic representations). For example in the proof exercises, tiles gain a semantic meaning on the server because they are transformed into Isabelle/HOL's modelling language.

MathsTiles is designed around the following goals:

1. **Resemblance to maths**.
The notation used to enter and manipulate maths should look like the maths students are expected to write on the exam paper. If the notation were very different, for example a text-based formula language, then this would add a learning burden which is not directly related to the material being taught.

2. **Ease of alteration**.
We expect students to enter incorrect expressions and proofs most of the time – if they already knew the material they wouldn't be students. So, it is important that students can make changes to their expressions easily.

3. **No forced order**.
The interface should not force students to write syntax in a particular chronological order. While there are occasions where we do want to teach students to use a particular methodology, this should be enforced in the explicit teaching feedback, rather than as an implicit

by-product of the interface design. So for example, students should be able to build the middle parts of an expression before the outer parts if they want to.

4. **Low commitment**.
   It needs to be possible for students to write and play around with fragments of answers without being committed to them. The interface should allow students to construct as many answer fragments as they like in parallel.

5. **Progressive evaluation**.
   Sometimes, students might know what part of the proof or expression needs to look like, but get stuck on how to complete the structure. They should be able to ask for feedback from the tutor on an incomplete answer fragment.

6. **Ease of authoring**.
   Because it is not possible to identify in advance all the mathematical structures and notations that questions will need to include, it needs to be simple for question authors to implement new pieces of mathematics.

7. **Reasonable size for the web**.
   While fast broadband connections are becoming more common, performance over slower or more congested networks still should be reasonable. This means that both the code size of the client applet and the size of the MathsTiles documents need to be reasonably small.

Tiles containing arbitrary pieces of maths can be added to the canvas, dragged around and dropped into sockets in other tiles to build up the structure of an expression or proof by containment. In this way, the notation is kept closely mapped to handwritten mathematics, but the students are exposed to the hierarchical nature of the expressions they are building. A simple example of some tiles is shown in Figure 6.

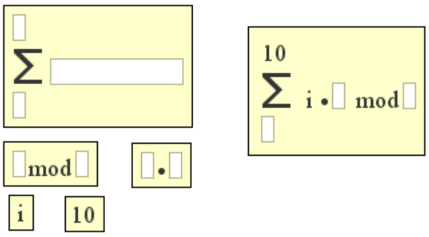


Figure 6. Some maths tiles, loose and combined

Tiles can be pulled out of and dropped into sockets by holding the `Ctrl` key when pressing or releasing the mouse over the tile or socket, so the effort required to change a structure is low. When a student drops a tile into a socket in another tile, the border of the contained tile is removed so that the appearance of the constructed maths is not interrupted. However, the tile border reappears when the mouse is moved over the tile, giving the student a clear sense of the structure of the tile.

The fact that tiles and groups of tiles can sit on the canvas without being combined into the student's answer means that students are able to write parts of their answers without being committed to them. New parts of an expression or proof can be plugged in without discarding the old parts. Also, because the tiles can be combined as easily in an outside-in or inside-out order, the student is not constrained to working in a top-down or bottom-up manner.

4.2. Document Structure

As described in Section 3.1, the student's document is an XML file and its document object model is updated in real-time on both the client and the server as the student works on it.

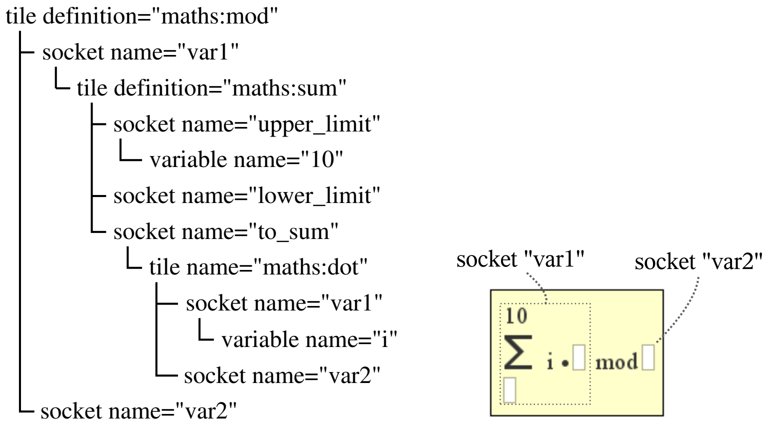


Figure 7. The combined tiles from Figure 6, together with the XML of the structure, shown as a tree. The sockets of the `mod` tile have been labelled on the diagram.

Figure 7 shows the combined tiles from Figure 6 together with their XML structure. The outermost `tile` element has its `definition` attribute set to `maths:mod`. Most tiles in a document, like this one, are *defined tiles*. Their appearance and structure is not fixed in the MathsTiles program, but are described by *tile definitions*. Here, the tile is defined by the `mod` tile definition in a separate tile document called `maths`.

Within the `tile` element are two `socket` elements which are the two sockets of the `mod` tile. The socket called `var2` (the right socket) is empty, while the socket called `var1` (the left socket) contains a `sum` tile. This `sum` tile in turn contains sockets, some of which contain other tiles. Note that the socket names are local to the tile – if there was a second `mod` tile on the page, its left and right sockets would also be named `var1` and `var2`.

```
tileDefinition name="sum" layout="InstructionLayout"
├ socketDefinition name="to_sum" width="100" height="20"
├ text name="sum_sign" font-name="Math" font-size="20"
│  └ Σ
├ socketDefinition name="upper_limit" width="10" height="20"
├ socketDefinition name="lower_limit" width="10" height="20"
└ layout
   ├ move c1="sum_sign" e1="North" by="5" c2="upper_limit" e2="South"
   ├ move c1="to_sum" e1="West" by=">5" c2="sum_sign" e2="East"
   ├ move c1="to_sum" e1="v_middle" by="0" c2="sum_sign" e2="v_middle"
   ├ move c1="lower_limit" e1="North" by="5" c2="sum_sign" e2="South"
   ├ move c1="lower_limit" e1="h_middle" by="0" c2="sum_sign" e2="h_middle"
   └ move c1="upper_limit" e1="h_middle" by="0" c2="sum_sign" e2="h_middle"
```

Figure 8. The definition and layout of a Sum tile. "Component" and "Edge" have been abbreviated to "c" and "e" in this figure. The horizontal middle, vertical middle, and text baseline are also edges that can be used in alignment operations.

Figure 8 shows the tile definition for the `sum` tile from Figures 6 and 7. Within the `tileDefinition` element, there are three `socketDefinition` elements that define the three sockets in `sum` tiles. The names of the tile's sockets in Figure 7 match the names of the socket definitions in Figure 8. Here, the socket definitions have specified the sockets' widths and heights. There is also a `text` element that defines the sum symbol that appears on the tile.

The `layout` element corresponds to the fact that the tile definition's `layout` attribute is set to `InstructionLayout`. This `layout` element contains a sequence of `move` and `pull` elements that describe operations that will arrange the sockets and text on the tile appropriately. Alternatively, if the layout attribute was set to `BaselineFlowLayout`, then all the components of the tile would be arranged left to right, vertically aligned by the baselines of any text that appears on them. (The baseline of a tile laid out using `InstructionLayout` is the baseline of the first element in its tile definition.)

A tile is loosely coupled to its definition, so the visual appearance of a MathsTiles document can be changed by loading it with a different set of tile definitions. This is not as flexible as a stylesheet, however, because changing a tile definition always changes the appearance of every tile in the document referring to it.

4.3. Definable Tile Components

Tile definitions can include the following components:

Text. The text that appears on a tile is specified by `text` elements in the tile definition. By setting the `visible` attribute to an XPath [17] expression, a piece of text can be made to appear only if the expression evaluates to *true*. This can be used, for example, to make brackets appear on a `plus` tile when it is placed in a socket in a `times` or `power` tile.

Symbols. Symbols can be defined using the using the Scalable Vector Graphics (SVG) path syntax, and given a name. Once defined, a symbol can be placed on a tile by including a `symbol` element in the tile's definition. As with text, each symbol on a tile can be given a visibility that depends on an expression.

Sockets. Each socket is defined by a `socketDefinition` element in the tile definition. Background text can be set to appear on the socket when it is empty. The colour, height, and width of the socket can also be specified. The `tagName` attribute provides a rudimentary way of setting what kinds of tile can be inserted into the socket. If it is set then only tiles whose element tag (for non-defined tiles) or definition (for defined tiles) appears in the list of names in the `tagName` attribute will be accepted into the socket. When a tile is being moved and the `Ctrl` key is pressed, the socket underneath the tile that the student might want to drop it into will outline itself in green or red depending on whether it would accept the tile or not.

Socket lists. Horizontally or vertically arranged lists of sockets can also be placed on a tile. Socket lists can have a specified number of sockets, or they can be set to expand automatically so that there is always an empty socket in the list. Expanding socket lists place an ellipsis ('`...`') at the end of the list to show that it will expand. A `socketDefiniton` within the `socketListDefinition` defines what the sockets in the list should look like.

Three attributes of tiles are also worth noting. `Selectable` (default *yes*) sets whether or not the user can select this tile. Unselectable tiles are effectively stuck on the canvas or in their sockets. If they are stuck within sockets then the socket border will not highlight when the mouse moves over the tile, and the unselectable tile will appear to be an integral part of its parent tile. `Delible` (default *yes*) sets whether or not the tile can be deleted. `Background` sets the background colour of the tile.

4.4. Non-Defined Tiles

In addition to defined tiles, MathsTiles also provides four hardcoded kinds of tile for convenience with mathematics:

Variable. A variable is a simple tile containing text that matches its `name` tag. It is also useful for representing numbers.

Function. A function contains text that matches its `name` tag, and sockets for its parameters. The sockets are surrounded by parentheses. Functions can take a configurable number of parameters, or can be set to automatically expand. A separator character can also be configured.

Labelled statement. A labelled statement is a tile that contains a socket for the statement, and text for the label. The label is set using the `id` attribute.

Statement reference. A statement reference is a simple tile containing text that matches the label of the statement it references (defined by the `id` attribute).

4.5. Tile Trays

The set of buttons and controls that the student can use to add tiles to the proof (called a *tile tray*) is also defined in XML. It can form part of the student's proof document, or it can be part of a separate document in the same way that the tile library documents are.

Tile button. Inserts a tile, as specified by the definition referred to by the `definition` attribute.

Xml button. Rather than inserting a single tile, an XML button inserts tiles to match a defined XML structure. This is useful for commonly-used expressions (such as the expression contained in the theorem to be proved), and also where we wish to insert a nest of tiles but treat it as a single tile. By marking the contained tiles in the nest as unselectable in the XML, we can prevent them from being pulled out of their parent tile.

Variable button. Inserts a variable. The name is specified by typing it into an edit box set into the button.

Statement button. Inserts a statement label or a statement reference. If the text typed into the edit box (within the button) is already the label of a statement on the canvas, then a statement reference is added. If not, then a statement label is added. If the edit box is left blank, then the button automatically generates a new label.

Tabbed pane. Holds a set of tabs.

Tab A labelled tab group that can hold a set of buttons. (May or may not be within a tabbed pane.)

Expression button. Parses an expression typed or pasted in by the user, and produces a tile structure to match that expression. Its primary purpose is that if a hint message or a response from the prover contains an expression, the user should be able to paste that expression into the proof. It is also included, however, because simple one-dimensional expressions such as $3 + 4$ are much faster to type than to construct with the mouse. (See Section 4.6.)

Tile search button. This takes advantage of the dynamic nature of the tile tray. The tile tray, like the proof document, can be altered at run-time by scripting calls from the server. This means that not all of the buttons the student will use for the question need to be in the tile tray at the start. The `TileSearchButton` sends the student's search query to a function in the question's teaching script, which usually responds by adding found tile buttons to a "search results" tab in the tile tray.

4.6. Closing Note

In this section we discuss two limitations in MathsTiles that were known at design time.

Despite the fact that being able to type is known to be useful in structured editors, it is not possible to edit in MathsTiles by typing. The only expression control provided uses a parser that only accepts a few formats (XML, Isabelle/HOL expressions, and basic arithmetic), although it is reasonably forgiving of errors. The reasons for this limitation become clear when you consider that MathsTiles does not have a fixed syntax, but a changeable syntax from question to question. It is also technically possible for new tile definitions to be introduced during questions. Furthermore, many of the tiles use a two-dimensional syntax ordered by layout rules. It is not obvious what is the most usable technique to convert from a one-dimensional syntax (text) to an *ad hoc* two-dimensional syntax. So, this is left for future work.

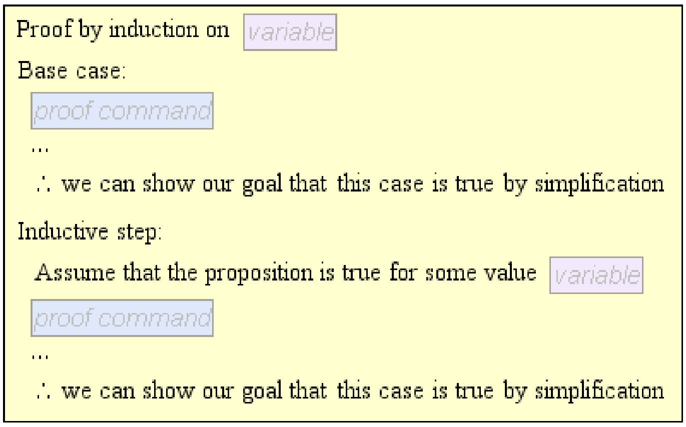
It would be useful if parts of a tile or socket definition could depend upon an attribute of the tile or socket. For example, if a piece of text that appears on a defined tile could be set to match the `name` attribute of the tile, then it would not have been necessary to hardcode variable and function tiles. As a second example, if a socket could be defined to only accept tiles where an expression such as "$socketDefinition.type = tile.type$" was true then this would allow question authors, if they wished, to prevent students from inserting tiles into unsuitable sockets. Currently the `tagName` attribute provides only rudimentary support for this. However, MathsTiles was designed to work with version 5.0 of the Java Runtime Environment, which includes an expression parser for XPath but not for any more general purpose languages. XPath expressions cannot bridge documents and we usually keep the tile definitions in library documents that are separate from the question document. So, we would need to include our own general purpose expression language for tiles, which we decided would make the applet size too large. Java version 6.0 does include general purpose languages that we can use for this purpose in future versions.

5. MathsTiles as a Proof Language

5.1. Proof tiles

One use for definable tiles is to expose to students what they need to do to fully answer an exam question. For example, what is needed to complete an induction proof, or how to show that a set relation is an equivalence relation. Tiles can be defined that include sockets for each part that students need to complete. A tile for natural induction that we use in a worked example later in this paper is shown in Figure 9, along with its Isar translation. It is implemented as a nest of tiles, but some of them are marked as unselectable (and so cannot be taken out of the parent tile), so to the user it appears to be a single tile. The tile contains a socket for the student to fill in the induction variable. Beneath that is a section for the base case. This contains an expanding socket list for the proof steps the student will take to show the base case. The final goal step has already been filled in, and in this case the tile uses the shortcut "*this case*" tile for the goal statement. The reason for this is described in Section 5.5. A second section in the tile is provided for the inductive step case.

This is not, in fact, the only induction tile we use. For example, Figure 12 uses a different induction tile.



Proof by induction on variable

Base case:

proof command

...

∴ we can show our goal that this case is true by simplification

Inductive step:

Assume that the proposition is true for some value variable

proof command

...

∴ we can show our goal that this case is true by simplification

```
proof (induct variable rule: altInduct)
case base
proof commands
with prems show ?case by simp
next
case (step variable)
proof commands
with prems show ?case by simp
qed
```

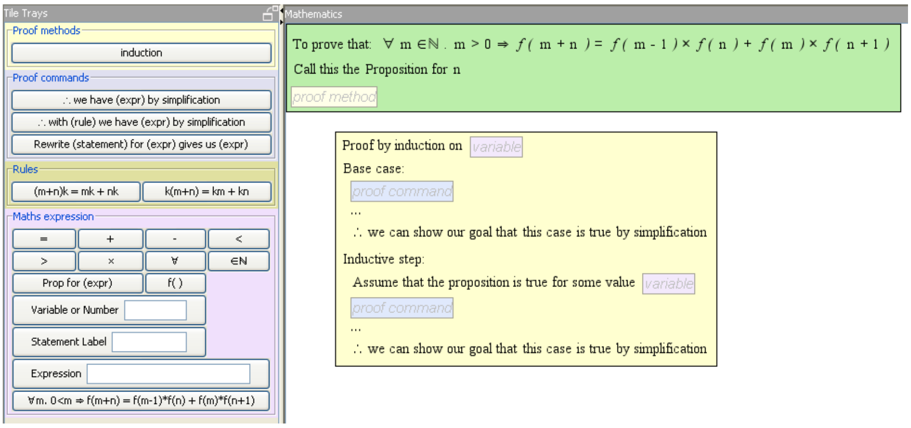
Figure 9. A tile for natural induction that is used in Section 6, and its Isar translation.

It is important to note, however, that socketed tiles are not proof sketches. Proof sketches [27, 44] are proofs with some of the low level reasoning omitted to make the essence of the proof more readable. The main reasoning steps are shown in full in a proof sketch. Proof tiles, meanwhile, are syntax templates that do not contain any of the statements in the proof until the student fills them in.

In the Isar code of Figure 9, notice the text "`rule: altInduct`". This is not represented anywhere on the tile. This is a small example of how we can hide code that is specific to the question in the Isar conversion of tiles. In this case it is simply that Isabelle/HOL's default induction rules use the successor function and consider cases 0 and $Suc(n)$, whereas for this question we wanted to reason with cases 0 and $n + 1$. We therefore hid an alternative induction rule in the Conversion Script for the question, and set the induction tile to use it.

5.2. Colour Coding

Although MathsTiles does not support a formal type system, we can provide the user with a few hints. In the proof exercises, we colour code the sockets of tiles, and colour code the background of sections of the tile tray to match. This is illustrated in Figure 10.



To prove that: $\forall$ m $\in\mathbb{N}$. m > 0 ⇒ f(m + n) = f(m - 1) × f(n) + f(m) × f(n + 1)

Call this the Proposition for n

Proof by induction on

Base case:

...

∴ we can show our goal that this case is true by simplification

Inductive step:

Assume that the proposition is true for some value

...

∴ we can show our goal that this case is true by simplification

Figure 10. A tile containing a coloured socket with background text, indicating what kind of tile should be dropped into it. The buttons with the same background colour produce the right kind of tile for the socket

There are four different socket colours used. The pink sockets are for expressions. These correspond to the inner HOL syntax in Isabelle, whereas the other three colours all correspond to aspects of the outer Isar syntax. As Isabelle/HOL works through the proof, its Isar virtual machine [43] moves between two modes that describe what kind of operation is expected next. In the *proof(state)* mode, the proof is expected to state new assumptions, goals, and intermediate results. The blue sockets and buttons are “proof commands” that correspond to this mode. In the *proof(prove)* mode, the proof is expected to justify a goal or result that it has just stated. The yellow sockets and buttons are “proof methods” that correspond to this mode. (The Isar VM has a third mode, *proof(chain)*, that the proof exercises do not use.) The khaki sockets and buttons are for statement labels and rule names.

Dark green, meanwhile, has been used as a colour code for the question tile – the unselectable and indelible tile that describes the statement to be proved and contains an empty socket waiting for the proof.

5.3. Reasoning Step Size

Answering a proof exercise is a very different situation from professional or research use of a theorem prover. In professional use, users should be able to use advanced automated proof-finding techiques to make their work easier. In a proof exercise, however, we need to limit the automated proof-finding techniques that students can use because the students are supposed to answer the question, not the prover. We want to limit the prover to only being able to take “obvious steps”.

The approach we have taken is to limit the student to only using `simp`, which is Isabelle’s simplifier. The simplifier can handle many simple steps, such as algebraic rearrangements, but cannot automatically solve the proofs we wish to set. It allows us to have a “configurable notion of triviality”. Rules can be added or removed from the simplifier. This can be used to force students to state steps we consider important.

5.4. Annotations

```
processor.matcher(MATHSTILES_NAMESPACE, "tile",
{it.getAttribute("definition")=="proofs:inductionNatManual"},
{
  output.append("<proofstep>proof (induct ");
  processor.process(it, "mt:socket[@name=’variable’]);
  output.append(" rule: altInduct)</proofstep>");

  processor.talk(it);

  processor.process(it, "mt:socketList[@name=’step list’]");
  processor.process(it, "mt:socket[@name=’show’]");

  output.append("<proofstep>qed</proofstep>");

  processor.talk(it);
});
```

Figure 11. A “matcher” (pattern + template) for one kind of induction tile. The second (large) code closure describes the procedural actions to take for these tiles. `it` refers to the document element that has been matched: the tile. The base case and step assumption are implemented as unselectable tiles contained within the the `step list`. Consequently, their Isar code is not produced by this matcher but by their own separate matchers.

As described in Section 3, when proofs are executed in Isabelle/HOL, the responses are collected as annotations to make on the student’s proof document. Figure 11 shows a matcher from one of the conversion scripts. The `output.append(...)` calls append PGIP-formatted Isar commands to the broker's buffer. The `processor.talk(...)` calls then tell the broker to write its buffer out to Isabelle/HOL and collect the responses as annotations. The annotations are associated with the tile that is passed into `processor.talk(...)`. Usually, this is "`it`", which is the tile the matcher is processing. So, by choosing which matchers should call `processor.talk(it)`, we select where the annotations will appear.

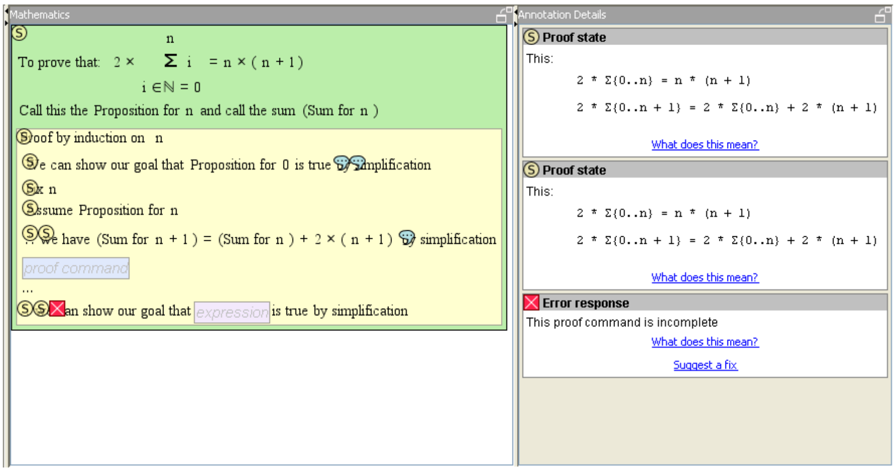
The annotations are first shown to the student as small icons on the tiles. These annotations are the reason why the induction tile in Figure 9 is implemented as an inseparable nest of tiles: although the nest behaves to the user like a single tile, the annotations need to be marked against the commands that caused them. For example, the proof state in the base case is different from the proof state in the inductive step. The annotation types are:

- Ⓢ Proof state – these annotations let the user see what goals need to be proved at this stage of the proof, and what premises are being used.
- 🗨 Comment – non-error comments, such as saying that a goal has been successfully shown.
- ☒ Error – faults Isabelle/HOL has found with the proof, or errors in syntax.

Clicking on an icon causes more details about the annotations on that tile to be shown in a separate pane, as shown in Figure 12.

The responses from the prover are post-processed in the broker in order to make the messages more understandable to the student. They are also assigned topic keys, which refer to the content model described in Section 3.2. The "*What does this mean?*" link in the annotation pane looks up a the associated topic in the book. Error annotations have a "*Suggest a fix*" link underneath them. Clicking this link calls an advice function in the teaching script for the error's topic.

The teaching script superclass for proof questions contains some advice functions for common errors topics. For example, it includes a helper function for the "*Proof command failed*" error message that will try a number of different values for variables to try to find a counter-example that would show the proof line was untrue rather than just unproven. This finds the relevant state annotation that contains the premises and goals of the failed command and parses each goal and premise. It attempts to find numbers which match the premises but do not match the goal statement. An advantage it has over just using Isabelle/HOL's in-built mechanism for finding counter examples is that the teaching script can use a different definition of a function. For



To prove that: $2 \times \sum_{i \in \mathbb{N} = 0}^{n} i = n \times (n + 1)$

Call this the Proposition for n and call the sum (Sum for n)

roof by induction on n

e can show our goal that Proposition for 0 is true mplification

x n

ssume Proposition for n

∴ we have (Sum for n + 1) = (Sum for n) + 2 × (n + 1) simplification

...

an show our goal that is true by simplification

Proof state

This:

```
        2 * Σ{0..n} = n * (n + 1)

        2 * Σ{0..n + 1} = 2 * Σ{0..n} + 2 * (n + 1)
```

Proof state

This:

```
        2 * Σ{0..n} = n * (n + 1)

        2 * Σ{0..n + 1} = 2 * Σ{0..n} + 2 * (n + 1)
```

Error response

This proof command is incomplete

Figure 12. The responses from Isabelle/HOL are marked on the proof tiles as annotation icons; these annotations can then be shown in full in the annotation pane by clicking on their icons. (Since the user has already placed the tiles, and so knows what they are, the fact that the icons can obscure some of the text on the tile is less of a problem than it might appear from the picture.)

example, using the equation for the n^{th} Fibonacci number instead of the recursive definition of the sequence.

5.5. Two Design Compromises

In Section 2, we described design goals that the student should have to write the statements in a proof and that the proof should resemble what students write on paper. In this section, we describe two compromises we made in this area.

5.5.1. *The student does not always have to write the goal statements.* Referring back to the induction tile in Figure 9, the goal statements for the base case and inductive step are simply the shortcut "*this case*". The student has not been forced to write them.

The reason we sometimes use this shortcut is that when we tell Isabelle/HOL that we are using induction or proof by cases, Isabelle/HOL automatically works out what the goals need to be for each of the cases. Students, if they were allowed to write in the goal, might write it in a way that a human would consider equivalent but that is very slightly different to the goal Isabelle/HOL calculated. This would then cause the goal statement to fail. Isabelle/HOL expects the goal statement to be shown exactly as calculated, and will not allow something to be shown that is a few steps of logic away instead.

A possible workaround for this would be for the tile not to use the `show` command for the user's goal, but to treat it as just another `have` command and then hide a command to show the real goal by simplification in the conversion script. This would allow the user to put in a goal that was "trivially close" to the goal and the proof would succeed. Unfortunately, for goals that Isabelle/HOL's simplifier can prove from the definition, such as $\sum 0..0 = 0$, this would also allow the user to write in a true but irrelevant statement, such as $1 = 1$, as the goal and the hidden proof command would still prove the real goal. The human notion of a "trivial step" is different from the notion of whether a statement is equivalent to the goal.

Instead, we provide, the "*this case*" shortcut for questions where the student might find it awkward to write the goal statement exactly. In other cases, a "*Proposition for ...*" tile is provided. Inserting 0, 1 or $n + 1$ into this tile usually forms the correct goal for a base case or an inductive step.

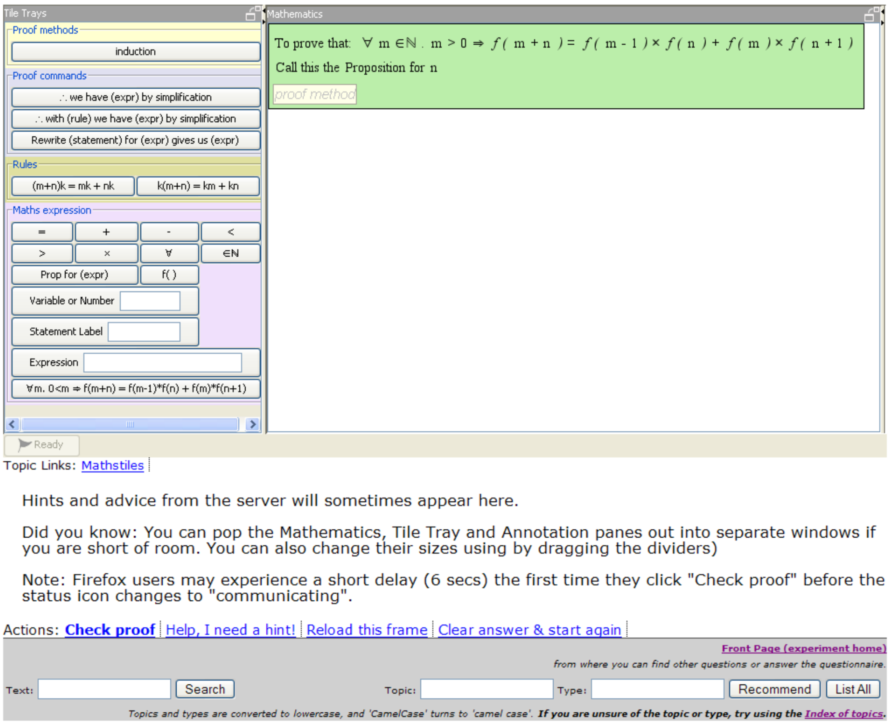
5.5.2. *The proof is checked linearly.*

The student is free to write the proof in any order using MathsTiles. However, because the proof is translated into Isar, an error in the proof is likely to cause every following line of proof to fail. These follow-on errors could be an unhelpful distraction from the original (causative) error, so when the proof is checked, the broker stops collecting annotations after the first error. This means that the student gets no feedback on correctness for the lines after the first error. While the interface does not prevent the student from constructing the proof in any order, the system provides much stronger support for starting at the beginning of the proof and working towards the end.

6. A Worked Example

This example is part of a question from the 2004 written exam sat by first year undergraduates in the computer science tripos. The student is given a definition of the Fibonacci sequence and is asked to prove by induction that $f(m + n) = f(m - 1) \times f(n) + f(m) \times f(n + 1)$ for all $m > 0$, where $f(n)$ corresponds to the n^{th} element of the Fibonacci sequence. Initially, the question appears as shown in Figure 13. The tiles on the page at the start of a question are fixed in place and coloured green; these need to be filled out to complete the proof.

The text "*Call this the Proposition for n*" on the green question tile corresponds to Isar's `(is "PROP ?P n")` syntax. This allows the



To prove that: ∀ m ∈ℕ . m > 0 ⇒ *f* (m + n) = *f* (m - 1) × *f* (n) + *f* (m) × *f* (n + 1)

Call this the Proposition for n

proof method

Topic Links: Mathstiles

Hints and advice from the server will sometimes appear here.

Did you know: You can pop the Mathematics, Tile Tray and Annotation panes out into separate windows if you are short of room. You can also change their sizes using by dragging the dividers)

Note: Firefox users may experience a short delay (6 secs) the first time they click "Check proof" before the status icon changes to "communicating".

Actions: **Check proof** | Help, I need a hint! | Reload this frame | Clear answer & start again

Front Page (experiment home)

from where you can find other questions or answer the questionnaire.

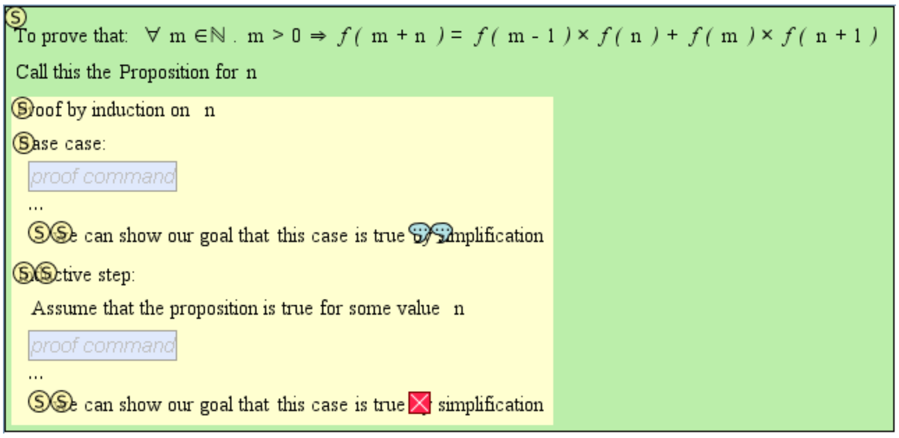
Text: Topic: Type:

Topics and types are converted to lowercase, and 'CamelCase' turns to 'camel case'. ***If you are unsure of the topic or type, try using the Index of topics.***

Figure 13. An induction proof question waiting to be filled in. Because this question is specifically set as an induction proof, no other proof methods are available.

"*Proposition for ...*" tile to be used as a shorthand for the entire statement to be proved.

The only socket available in the answer asks for a proof method. In the tile tray, there is only one button in the section marked "proof methods": induction. The induction tile is the same as from Figure 9 and has a number of sockets to fill. Let us assume the student inducts on n, and also uses n as the name of the particular value he has assumed the proposition is true for in the inductive step. The goals for the base case and the inductive step have been set to the short-cut statement "*this case*" for this form of the induction tile (a different induction tile used in other questions asks us to set them ourselves). Clicking "*Check Proof*", it seems that the base case can be solved automatically by the simplifier but the inductive step cannot. This is shown in Figure 14. The base case must have succeeded because its goal has comment annotations but no error annotation. (One of the comment annotations states that the goal has been proved.) The reason there is still an empty socket in the base case is because it contains an expanding socket list – the list will always expand to keep an empty socket available.



Ⓢ
To prove that: ∀ m ∈ℕ . m > 0 ⇒ f (m + n) = f (m - 1) × f (n) + f (m) × f (n + 1)

Call this the Proposition for n

Ⓢoof by induction on n

Ⓢase case:

proof command

...

ⓈⓈe can show our goal that this case is true ⓈⓈmplification

ⓈⓈctive step:

Assume that the proposition is true for some value n

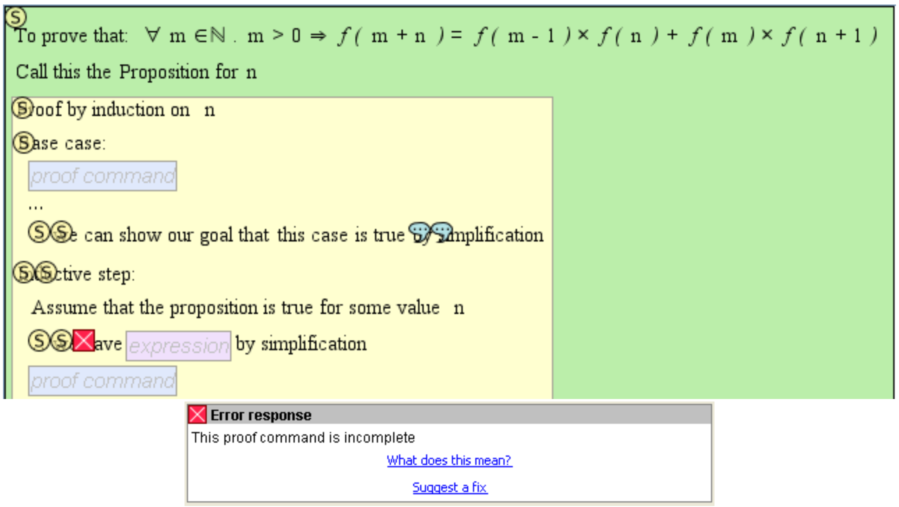
proof command

...

ⓈⓈe can show our goal that this case is true ☒ simplification

Figure 14. The base case can be solved automatically with the simplifier, but the inductive step is marked with an error icon because it still needs some work. (The reason there is still an empty socket in the base case is because the proof command socket list is set to automatically expand.)

To see what happens with errors, let's try dropping a "∵ *we have ... by simplification*" tile into the proof, and check it without inserting an expression (Figure 15). We could, of course, enter an incorrect expression or put the wrong kind of tile into the expression box to generate our error instead.



Ⓢ
To prove that: ∀ m ∈ℕ . m > 0 ⇒ f (m + n) = f (m - 1) × f (n) + f (m) × f (n + 1)

Call this the Proposition for n

Ⓢoof by induction on n

Ⓢase case:

proof command

...

ⓈⓈe can show our goal that this case is true ⓈⓈmplification

ⓈⓈctive step:

Assume that the proposition is true for some value n

ⓈⓈ☒ave expression by simplification

proof command

☒ Error response

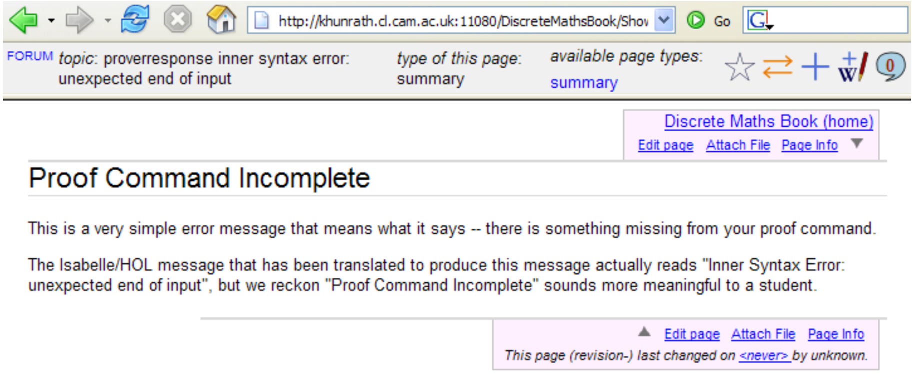
This proof command is incomplete

What does this mean?

Suggest a fix

Figure 15. An incomplete command in the proof, and its error annotation.

Clicking the "*What does this mean?*" link underneath the error annotation brings up a topic entry from the book explaining the error in more detail, as shown in Figure 16.



topic: proverresponse inner syntax error: unexpected end of input

type of this page: summary

available page types: summary

Discrete Maths Book (home)

Edit page Attach File Page Info

Proof Command Incomplete

This is a very simple error message that means what it says -- there is something missing from your proof command.

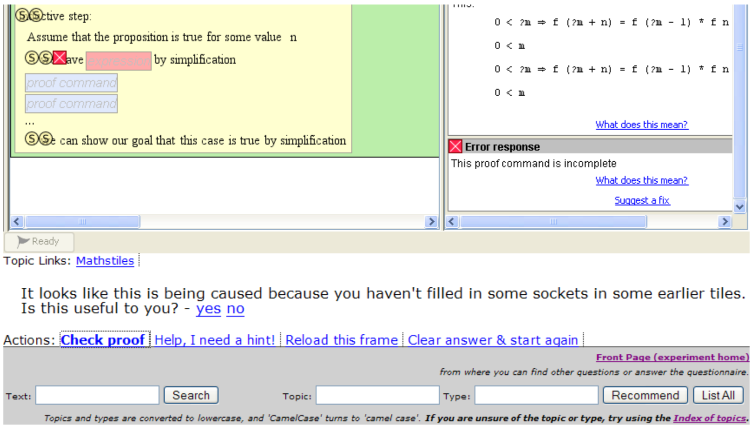
The Isabelle/HOL message that has been translated to produce this message actually reads "Inner Syntax Error: unexpected end of input", but we reckon "Proof Command Incomplete" sounds more meaningful to a student.

Edit page Attach File Page Info

This page (revision-) last changed on <never> by unknown.

Figure 16. A content entry explaining an error from the proof assistant. In this case it happens to tell us what the original Isabelle/HOL message was.

Clicking the "*Suggest a fix*" link asks the teaching script for advice on how to get past this error. The advice appears on the HTML page just below the MathsTiles applet, as shown in Figure 17. In this case, the suggestion is fairly simple – the script tells us about our empty expression socket and highlights it in red.



ctive step:
Assume that the proposition is true for some value n
ave
expression
by simplification
proof command
proof command
...
can show our goal that this case is true by simplification
Ready

```
0 < ?m ⇒ f (?m + n) = f (?m - 1) * f n
0 < m
0 < ?m ⇒ f (?m + n) = f (?m - 1) * f n
0 < m
```

What does this mean?

Error response

This proof command is incomplete

What does this mean?

Suggest a fix

Topic Links: Mathstiles

It looks like this is being caused because you haven't filled in some sockets in some earlier tiles.
Is this useful to you? - yes no

Actions: Check proof | Help, I need a hint! | Reload this frame | Clear answer & start again

Front Page (experiment home)

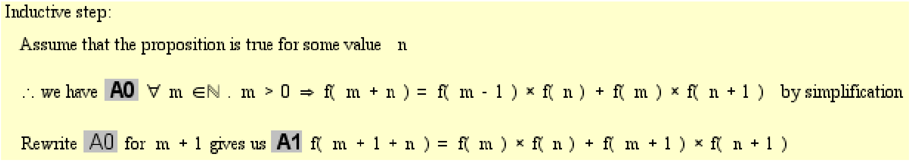
from where you can find other questions or answer the questionnaire.

*Topics and types are converted to lowercase, and 'CamelCase' turns to 'camel case'. **If you are unsure of the topic or type, try using the Index of topics.***

Figure 17. A piece of advice from the teaching script, shown in response to the student clicking the "*Suggest a fix*" link underneath an error annotation.

We'll now return to trying to complete the proof. If the student is stuck at this point, there is a hint in the teaching script which says "*If you have assumed the proposition is true for some* n *and any* m*, it must also be true for the same* n *and any* $m+1$*. What you want to do is to show it is true for* $n+1$ *and any* m". The student needs to rewrite the step assumption with $m+1$ instead of m.

However, in this version of the induction tile, the expression for the step assumption is not stated explicitly by the student but is automatically inferred and indicated by the text "*Assume the proposition is true for some value ...*" on the tile. We usually use a different induction tile in which the step has to be stated by the student, as seen in Figure 12, but for this question our usual induction tile kept striking a bug in the particular version of Isabelle/HOL we were using. So, in order to label the step assumption, the student must write out a proof line containing it, as in Figure 18. Because the expression in statement `A0` is identical to the step assumption, it is obviously true by simplification.



Inductive step:

Assume that the proposition is true for some value n

∴ we have **A0** ∀ m ∈ℕ . m > 0 ⇒ f(m + n) = f(m - 1) × f(n) + f(m) × f(n + 1) by simplification

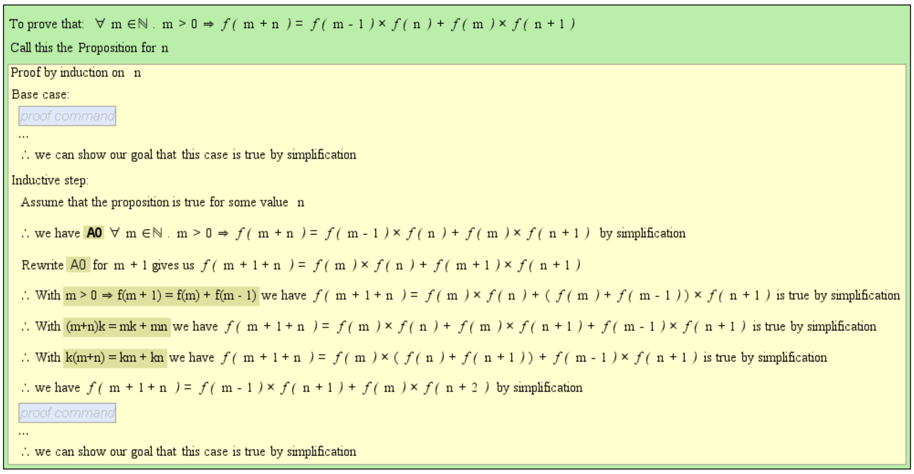
Rewrite A0 for m + 1 gives us **A1** f(m + 1 + n) = f(m) × f(n) + f(m + 1) × f(n + 1)

```
with prems have A0:
"!!m::nat. m>0⟹f(m+n) =f(m-1)×f(n) + f(m)×f(n+1)"
by simp
with prems A0[of "m+1"] have A1:
"f(m+1+n) = f(m)×f(n) + f(m+1)×f(n+1)" by simp
```

Figure 18. Labelling a statement and rewriting it for a new expression, in tiles and in Isar.

At this point, the simplifier still cannot get the rest of the way to proving the step case because it does not include particular rules it needs for particular steps. The student therefore has to write these steps in more detail, bringing the rules in using "*With ... we have ...*" tiles until we reach the finished proof in Figure 19. The rules to use are available in the "rules" section of the tile tray.

A final point to note about the question is that while m is universally quantified in the question statement, n is not. The reason for this is that when we univesally quantified n as well, the proof failed on the induction tile. The reason is because Isabelle/HOL, in the version we used, could no longer work out the induction scheme for the goal statement. There are other potential workarounds, but only universally quantifying m was the simplest to implement. The point we are illustrating with this, and with the change of induction tile, is that where a proof is technically difficult in Isar, the question author can work through the proof ahead of time and adjust the tiles and the conversion script so that the students' proofs progress more smoothly. We understand that more recent versions of Isabelle/HOL have an improved induction tactic and would not require this particular workaround.



To prove that: $\forall$ m $\in\mathbb{N}$. $m > 0 \Rightarrow f(m+n) = f(m-1) \times f(n) + f(m) \times f(n+1)$

Call this the Proposition for n

Proof by induction on n

Base case:

...

$\therefore$ we can show our goal that this case is true by simplification

Inductive step:

Assume that the proposition is true for some value n

$\therefore$ we have **A0** $\forall$ m $\in\mathbb{N}$. $m > 0 \Rightarrow f(m+n) = f(m-1) \times f(n) + f(m) \times f(n+1)$ by simplification

Rewrite A0 for $m+1$ gives us $f(m+1+n) = f(m) \times f(n) + f(m+1) \times f(n+1)$

$\therefore$ With $m > 0 \Rightarrow f(m+1) = f(m) + f(m-1)$ we have $f(m+1+n) = f(m) \times f(n) + (f(m) + f(m-1)) \times f(n+1)$ is true by simplification

$\therefore$ With $(m+n)k = mk + mn$ we have $f(m+1+n) = f(m) \times f(n) + f(m) \times f(n+1) + f(m-1) \times f(n+1)$ is true by simplification

$\therefore$ With $k(m+n) = km + kn$ we have $f(m+1+n) = f(m) \times (f(n) + f(n+1)) + f(m-1) \times f(n+1)$ is true by simplification

$\therefore$ we have $f(m+1+n) = f(m-1) \times f(n+1) + f(m) \times f(n+2)$ by simplification

...

$\therefore$ we can show our goal that this case is true by simplification

Figure 19. The last few steps in the question involve bringing rules in one by one using "*With . . . we have . . .*" tiles, to rearrange the expression until we reach something that simplifies to the goal.

7. Evaluation

7.1. Overview

Our goal in the evaluating the system is twofold. By asking students and others who have no experience of automated proof to attempt the exercises, we wish to see whether novices can make progress with the exercises with a bare minimum of training. More importantly, however, since this is a complex and unusual interface, we wish to understand the usability issues that arise from the system so we can see whether they are insurmountable and a different approach is required, or whether they suggest fruitful avenues of further inquiry.

To this end, we performed a user trial and qualitative usability study using the cognitive dimensions of notations (CDs) framework [21, 13]. CDs provide a formalised vocabulary for discussing usability issues, with sixteen "dimensions" that can affect usability. An example of a cognitive dimension is "viscocity", which is the question of how difficult it is to make changes to previous work using the interface. The CDs framework also provides means for considering *secondary notations*, *helper devices*, and *redefinition devices*, but in this study we only examined the primary notation: the MathsTiles proofs.

Two methods were used to collect usability issues:

1. A server with an introduction to the system and six proof exercises was made publically available, and its use over three weeks in July 2006 was examined. A range of users were asked to try the system,

including Cambridge undergraduate students, undergraduate students from other universities, postgraduate tutors of discrete mathematics, and other interested parties. The proof exercises we made available were about the greatest common denominator $GCD(a, b)$ and the Fibonacci series $f(n)$. The exercises were:

a) Prove that $2 \times \sum 0..n = n \times (n + 1)$, by induction on the natural numbers. This was the introductory example for which a walkthrough was given.

b) Prove that $GCD(f(n), f(n + 1)) = 1$, by induction on the Natural numbers.

c) Prove that $n > 0 \Longrightarrow GCD(n \times k + m, n) = GCD(m, n)$, by assuming the left hand side of the *implies* is true and showing the right hand side must follow.

d) Prove that $f(n + k + 1) = f(k + 1) \times f(n + 1) + f(k) \times f(n)$, by induction using a different induction rule.

e) Prove that $\forall m.m > 0 \Longrightarrow f(m+n) = f(m-1) \times f(n) + f(m) \times f(n + 1)$, by induction on the Natural numbers.

f) Prove that $GCD(f(n + m), f(m)) = GCD(f(n), f(m))$, by considering cases on m.

Three kinds of training item were provided. Two Flash videos, totalling just over three minutes in length, showed how to use the MathsTiles interface. An “introductory chapter” to the exercises, three pages long, explained similar material to the videos (for participants who might not have had the Flash plugin installed). A walkthrough described how to solve the first and simplest question, with screenshots.

The comments, feedback, and requests for help from users were coded against the CDs Framework by two researchers.

2. Additionally, to identify issues that novice participants might be prone to miss or unable to articulate, two researchers analysed the system using a cognitive dimensions of notations questionnaire [14]. This analysis was carried out both by the primary developer of the system and also by an undergraduate intern who worked with the system for two months.

Table I. The number of participants reaching each stage of the exercises.

Participants	Stage
83	Accessed the server and read about the system
19	Accessed the introductory question
8	Completed the introductory question
13	Accessed question two
8	Made a serious attempt at question two
5	Completed question two
3	Completed five of the six proofs

7.2. Numerical Results

The numerical results from the trials are shown in Table I.

While very few participants indicated whether or not they were students, from examining their email addresses and how they became involved with the trials we have confidently identified 44 of the participants as students. Of the five participants who completed question two, three were students. One of the participants who completed five proofs was a student; the other two were tutors of students but had no prior experience with Isabelle/HOL.

The five participants who accessed question two but were judged “not to have made a serious attempt” put fewer than six tiles on the canvas, placed all their tiles on the canvas very quickly, and did not attempt to construct any expressions or place a proof method into the answer tile. From this we concluded that they played briefly with the interface, but did not attempt the proof.

On the one hand, the results are encouraging. In the Introduction we noted that the shortest training course in Isabelle/HOL was four sessions of ninety minutes, with 300 slides, and we were given an (unscientific) estimate that students might take 10 weeks to be able to do simple things using the prover. In our trial, some novice users and students have been able to complete proof exercises despite their training being barely three minutes of videos, three explanatory web pages, and a walkthrough of a single proof. On the other hand, however, there is a very large attrition from 83 initial participants down to three who completed five proofs, and only one of those was a student. This suggests there are still some major issues to overcome.

We cannot, of course, determine the reason for the attrition from 83 participants down to 8 who made a serious attempt at a proof without a walkthrough. Many of these participants may simply have been interested in looking at a new interface, but not interested in attempting a mathematical proof. On the other hand they may have been scared away by the complexity of the system. We have, however, determined that the three participants who failed question two had became stuck.

7.3. Qualitative Results

In the user study, we asked participants to fill in a feedback questionnaire. However, we found that many of the participants were reluctant to fill in a questionnaire form, but were more than happy to contact us informally to give us their feedback. Consequently, feeback was received by email, instant messenger, and discussions with users who came to our office or phoned us to tell us their thoughts and demonstrate the issues they were having. While this meant that feedback was received in a less controlled manner, we found it had the advantage of immediacy – we were able to examine the participants' question documents when the issues were reported to see the issues in practice and ensure we had not misunderstood them.

After the user feedback had been received, we then conducted an analysis using the cognitive dimensions questionnaire.

For discussion purposes, we have classified the 30 qualitative statements produced by the studies into five categories, three of which we discuss in detail:

Non-problems: *Statements 10, 18, 21, 24, and 26.*
These are the positive and non-negative remarks. Since they obviously do not represent a usability issue to be overcome, they are not discussed in detail.

MathsTiles UI (simple): *Statements 2, 5, 6, 7, 8, 9, 11, 12, 15, 16, 17, 22, 27, and 30.*
These are usability issues we regard as straightforward enhancements to make to the MathsTiles or Intelligent Book user interface – bug fixes and simple feature requests that do not impact on the approach. Because of this we do not believe they are academically very interesting to discuss.

MathsTiles UI (complex): *Statements 13, 28, and 29.*
These are usability issues with the MathsTiles and Intelligent Book interface we regard as more complex or interesting. They are discussed in Section 7.4.

Proof language: *Statements 4 and 14.*
These are usability issues that specifically relate to using MathsTiles as a proof language that translates to Isar. They are discussed in Section 7.5.

Domain-specific (here number theory): *Statements 1, 3, 19, 20, 23, and 25.*
These are usability issues we regard as inherent to the problem of freely-written student proofs in "difficult" domains such as number theory. They are discussed in Section 7.6.

7.4. MathsTiles UI (Complex)

Statements 13 and 28: Limitations with the expression control
Statement 13 is an issue that to an extent has already been discussed in Section 4.6. A user has seen that it is possible to type expressions, and has assumed that any text that appears on a tile can be typed into the expression control and recognised as a valid expression. Unfortunately, the MathsTiles applet in its current version uses a traditional generated LL(k) parser with a fixed grammar. So, it is incapable of adding the defined tiles for a question to its expression grammar.

Statement 28 is another case where the fixed LL(k) parsing is insufficient, but for a different reason. The design assumption had been that users would wish to type complete algebraic expressions into the box to save the effort of composing them from tiles, or would cut and paste expressions into the box from annotations. However it turns out that very often users only want to add the few tiles they need to alter an existing expression, but they still type them into the expression control. These few tiles are necessarily an incomplete expression fragment, and might or might not be parsable with the current parser. One possible solution to this would be to support placeholders in expressions (or effectively to have a syntax element for an empty socket). For example, the expression "3+_" could represent an addition where the right socket is left empty.

An interesting observation we made was that the users who became confused by the expression box and reported Statement 28 were trying to use the expression control in the first exercise. After it was reported, we removed the control from this exercise. The user who expressed praise for the expression control was a user who, because of its removal from question one, only met the Expression button in question two. This might just be an individual difference in the users of course, but it might suggest that as users gain even a little more experience of an interface, they become much readier to work around the limitations of newly introduced components.

Statement 29: Missing entry in the book
In Statement 29, a user was surprised by the Intelligent Book defaulting to a web search when it found it did not have an entry for a topic. While we ensured that participants saw an introduction to the maths problems, we did not ensure that they saw an explanation of how the book's content features work. (There was a low-key link on the instructions page, but we deliberately did not draw attention to it.) We left this particular entry out of the book to see whether or not participants would add an entry when they discovered the feature, even though they had not been explicitly taught how to. They did not.

7.5. Proof Language

Statement 4: Universal quantification and the rewrite tile
In Statement 4, a student is unaware that a statement must include a universally quantified variable before it can be rewritten with a different expression substituted for that variable. This appears to reflect that either students do not yet understand the difference between a variable that has and has not been universally quantified, or they assume that all the variables in the statement are implicitly universally quantified. Unfortunately, we also found from experience of writing questions that proofs run into fewer technical problems in Isabelle/HOL when the variables in the expression are not universally quantified.

Statement 14: Labelling of prior statements
Statement 14 perhaps represents a difference between the way people informally view proofs and the way formal proof languages do. The students were surprised that the prover appeared to "forget" statements that were only two lines back in the proof. When we write English language arguments, we expect the reader to remember the context of the text so far without labelling the earlier sentences we refer to. The "*∵ we have ... by simplification*" tiles that students were using in their questions, however, translate to the Isar structure "`with prems have ... by simp`". This uses only the previous line and the assumptions to justify the new line of proof. If any earlier lines of the proof need are needed, they must be labelled and referenced explicitly. On the other hand, forcing students to state which previous lines they are using seems like a good thing educationally – it forces students to think about the structure of their proofs.

Perhaps a suitable approach would be to make referencing earlier statements much easier, by automatically labelling every proof line, and to add some kind of a visual hint to show that by default proof statements only use the immediately prior statement and the assumtions.

7.6. Domain Specific Issues (here Number Theory)

Statement 1: Tiles were only provided for one solution
In Statement 1, the problem is that not enough tiles have been provided to allow the student to solve the problem by a different proof strategy than the author intended. In terms of cognitive dimensions of notations, there is a trade-off between visibility and premature commitment here – by providing more tiles it becomes slightly harder to identify which ones you need. In this particular case, the extra tile is a different induction tile, and providing it would be unlikely to make the tiles too hard to find. However, in cases where you need to provide extra rules to support alternate strategies, this loss of visibility could become a much greater problem.

The set of rules that the simplifier includes, and so do not need to be mentioned explicitly, is called the *simpset.* In the second question, the simpset included some 1,570 rules. While students do not need to know what rules are in the simpset, they need to be able to ascertain what rules are *not* in the simpset. How else could they know know they need to state them? The set of rules in the tile tray gives a strong visual cue as to which rules have to be stated. However, the more rules there are in the tile tray, the harder it is to spot each rule. Taking the rules off the screen (and using a query mechanism for them) does not appear to be a viable option. Students would only be able to articulate what rules are necessary for a step if the step size was very small.

Allowing students to use more complex automated methods, rather than just the simplifier, would be one possible way of resolving this issue. However, there is the danger that students might abuse the automated methods to solve the question by trial and error. There appears to be an interesting trade-off between allowing students to "game the system" and making it easier for them to explore the proof.

Statement 3: Proofs are fragile
Statement 3 describes how changing an early line of the proof can cause later lines to fail. Even a trivial re-ordering of additive terms in an equation can cause a rewrite rule to fail – the terms are equivalent to the student but not to the prover. There are two aspects to this. On the one hand, perhaps the system should remember which lines it has already proved, and be more reluctant to mark those lines as no longer proved. On the other hand, this could give an inaccurate proof

document, where lines of proof purport to have come from one chain of reasoning, but actually come from another. Another potential solution might be to use a less rigorous theorem prover that treats “equivalence” in a manner more similar to what the student expects.

Statements 19 and 20: Students could only take small proof steps
Statements 19 and 20 describe how the steps students can make at each line of a proof in an exercise are very small. In the proof exercises, this relates to the fact that we only allow students to use the simplifier, and we only allow them to invoke one non-trivial rule at a time. However, even if these restrictions were relaxed it is questionable whether students would be able to take steps as large as they would like. The reasoning steps that automated methods can make do not easily and naturally correspond to the steps that a human can make. So, just as humans can take reasoning steps that are hard to verify automatically, automated methods can also take steps that a human would find hard to follow. If we rely purely on automated reasoning to provide the model for a question, then we can only support smaller steps that both automated methods and humans can follow.

Statement 23: Students could not recognise a bug from a mistake
Statement 23 describes how a bug in the tile translation caused an error in some proofs, but students could not tell that this was due to a bug and assumed their proofs were wong. This is perhaps an inherent problem with a teaching system in a difficult domain – because students are inexperienced with the material and the system, they find it difficult to think critically about whether the system is operating as expected. This means that very careful testing and debugging of proof questions is necessary before they are made available to students.

Statement 25: Insufficient measure of progress
Statement 25 describes how the only visible measure of how far a student has progressed through a proof is the number of rule tiles that have been provided but not used yet. It would be possible to provide a more direct measurement of progress by comparing the student’s proof to a pre-written proof, but as with Statement 1, this would raise the issue that unexpected solutions could not be supported in this way. Practically it might be appropriate for exercises to provide guidance and support for a number of pre-planned proofs, but allow unexpected proofs also to be constructed even though only limited assistance could be provided for them.

8. Future Work

This paper has not looked at the issue of how students can define lemmas in their documents. The reason why we did not consider it here is that proof exercises are often set in a number of stages. Parts (a), (b), and (c) might ask the student to prove particular useful lemmas, and then part (d) might ask the student to use those lemmas to derive an important result. In the exercises, each of these parts could be set as a separate exercise. (And indeed the final question in the study did draw together the lemmas proved in the previous questions.) However, not all questions on paper are set in this broken-down style, and if we are building a reactive learning environment to let students try out their ideas, then it is important that they should be able to take their own approaches to solving the proofs.

Performing a direct translation from MathsTiles to Isar is a fairly naïve approach to the problem. It would be prefectly reasonable instead for the broker, when examining a line of proof, to set all of the previous statements as lemmas, define the proof line as a goal theorem, and see whether an automated tactic can prove it or not. There would need to be some careful consideration of what theorems should be given to the tactic, however, so this would move much of the problem into the configuration of the proof tool. However, it could allow the MathsTiles proofs to resemble Isar much less – there would not need to be a straightforward translation to Isar.

The question of how to support text-based editing of tile documents, despite the two-dimensional dynamic nature of the syntax, has been raised but not addressed in this paper. A possiblity is to include a boolean parser in the system, and allow the tile definitions to specify one-dimensional (text) grammar rules as well as defining the appearance and structure of the tiles.

Writing proofs using tiles is currently a one-directional activity, where the student writes the proof and the system comments on it. However, where there are dependencies between elements in the proof, it may be helpful to allow the system to write or adjust parts of the proof, or to allow parts of tiles to be calculated from their surroundings rather than strictly defined in the XML. For example, if an early proof line is changed that breaks later lines of proof, perhaps the system should attempt to automatically adjust the later lines so they are no longer broken. This might also be a suitable approach for dealing with meta-variables.

Some of the usability issues raised suggest that proving is more like programming than we had assumed. For example, the need for automatic labelling of proof statements is similar to line numbering.

The annotations appearing on the tiles were found to be problematic because they could be obscured by other tiles or the edge of the window, and so a more traditional gutter seems appropriate. However, there are also aspects of the proof exercises that may be applicable to programming environments. For example, it has been observed that programmers frequently find themselves substituting blocks of code between a set of alternatives [25]. Being able to extract syntactic sections of code and leave them on the page but not in the code might be helpful.

It may be helpful educationally to be able to have a configurable level of formality in the prover. For example, we observed that students did not appear to understand the issues around universal quantification. What if the model could be made to temporarily forget those issues until the student was due to learn them? A common technique through school education is first teach something that is not quite correct but is easier to understand, and teach the harder truth later.

9. Related Work

9.1. Teaching Systems for Proof

The EPGY theorem proving environment [38] is a stand-alone proof environment used in a number of courses at Stanford University. Students begin with a set of given statements and a proof goal. A menu based system allows the student to apply built-in strategies and inference rules to goals in order to build up a proof — this aspect of the system is intended to encourage "structured theorem proving". Additionally, students can enter their own intermediate goals using a formula editor, and the proving environment will attempt to verify these goals using the Otter automated theorem prover.

The Dialog project [10, 9] is an ongoing project developing a system that can discuss proofs with students in natural language. The principles behind their philosophy are similar to those behind AutoTutor [19]. Human-to-human tutorials have frequently been found to be an effective teaching technique, so they wish to carry the pedagogy from those human tutorials across to automated tutorials. The proof domain the project has most examined is naïve set theory.

A number of educational systems have been designed for propositional (or sentential) logic. The Carnegie Mellon Proof Tutor (CPT) [36], the P-Logic Tutor [29], and Logic-ITA [28] are all examples of intelligent tutors designed to teach propositional logic. CPT uses a combination of Fitch diagrams and a goal tree to describe the proof being developed. Logic-ITA represents proofs fairly simply — as a sequence of proof lines in a table — and focusses instead on detailed and effective modelling and assessment of the student. It assesses the validity of proof steps as the student works on them, and once the proof is complete returns to assess the usefulness of each of the steps. P-Logic Tutor doubles both as a tutor and as a research environment for tracking student learning and exploring the cognitive issues involved. ETPS [2] assists students in writing and checking formal proofs in propositional logic. The student asks ETPS to apply particular rules of inference, and ETPS handles writing the mathematics. Ehrensberger's and Zinn's DiaLog system [20] treats propositional logic as a game between a proponent and an opponent. Proving a thesis is correct involves demonstrating that the proponent has a winning strategy that can successfully defend against any possible attack from an opponent. The user plays the part of the proponent, while DiaLog ensures that all possible alternatives of the opponent are considered. Hyperproof [7] teaches students the principles of analytical reasoning and propositional logic in the blocks world of Tarski's World.

Tutch [1] is a tutorial proof checker that does away with proof environments completely and requires the proof to be written in a human-readable text-only syntax. In its goal to provide a human readable formal proof syntax, it is similar to the Isar language that the Maths-Tiles proofs in our system are translated to, but designed specifically for education.

There appear to be a wide variety of educational proof systems for domains where artificial intelligence can reasonably be expected to find an answer automatically without human intervention, for example propositional logic. There are comparatively fewer systems for "harder" domains, such as number theory. The EPGY theorem proving environment is the most similar to our system in that regard. The obvious difference is that EPGY permits students to complete the proof by applying tactics from a menu, which we were concerned might lead to students gaming the system. (We are not aware of any studies that have investigated "gaming" of EPGY to verify whether this occurs.) It has the advantage, however, of making the interface much more straightforward.

9.2. Web-based Textbooks

Deploying texbooks and exercises on the web has been an active area of research recently. Two recent projects in this area, both of which have looked at builting web-based textbooks for mathematics, are particularly of note.

The Living Book [8] project has developed an online adaptive book for teaching logic to computer scientists. Its content model is based on semi-automatically dividing documents into *slices* that can represent a piece of information about a topic. These slices are reassembled in different levels of detail depending on the student's level of knowledge and the scenario the book is being used for. The mathematical exercises in the living book use a text entry language that is converted to PDF for display. It interfaces to the Otter theorem prover, producing output as proof trees.

ActiveMath [30] is a web based learning environment based on a detailed semantic model of mathematics (OMDoc [26]). Additionally a detailed model of each student is kept, modelling against competencies as well as knowledge. Using these detailed models of the content and the student, it is able to generate appropriate personalised courses for individual students.

As described in Section 3.2, these systems have a much more formalised content ontology and do not permit students to add new topics and content into the book. In the case of ActiveMath, it is also an open question as to how proof questions like ours could fit within the system. ActiveMath expects its exercises to be able to update the student model very precisely with details of which rules and which process steps the student understands, and which competencies he or she has shown. We suspect that the cognitive process of finding a proof is not yet well enough understood to be able to model the skills of an individual student.

9.3. Structured Editing

Structure-based (or syntax-directed) editing has a long history as a technique. As early as the 1970s, systems such as EMILY [22] and the Cornell Program Synthesizer [40] allowed programs to be constructed by choosing syntactic templates in a top-down manner, rather than by typing text to be parsed. Recently, GNU TeXMacs [41] has applied the technique for WYSIWYG editing of mathematical and TEXdocuments.

Syntax-directed editing has been found to be a useful technique to help novices to work with an unfamiliar programming syntax – the novice is guided by menus of legal operations, and syntax errors become impossible to make. The Carnegie Mellon programming environments [31] pioneered this use for the technique in the 1980s, and the Alice2 programming environment [23] is a more recent example.

However, MathsTiles is different in three ways. Firstly, it allows multiple code fragments to be scattered across the canvas, which means it does not have the restriction that "if it is on the page, it is in the code" that is common to other structured editors. Secondly, it is a structured editor for informally defined languages that translate to formal language, rather than for languages with formally defined syntaxes (and it allows students to make mistakes). Thirdly, it allows the interaction behaviour to be altered for individual pieces of syntax at run-time. For example, the green question tiles are individually set to be unselectable and indelible. A change message from the server, however, can remove that restriction, or make any other tile on the page unselectable. Another change message could introduce a new tile with a new tile definition, effectively altering the syntax of the language.

10. Conclusions

The exercises appear to have enabled a few users in the study to complete formally verifiable proofs with a surprisingly small amount of training. The usability issues raised with the interface during the study do not appear to be insurmountable, although there remain a number of significant challenges these proof exercises have not addressed. For example, each of the exercises only provided the right tiles for a solution that had already been carefully checked by the teacher. This means that although students are theoretically free to "try out their ideas" in a reactive learning environment, in practice they can only succeed with ideas the teacher has thought of for them.

Two participants commented informally after the study that through attempting the exercises they felt they had learnt a little more about automated proof assistants, and felt braver to try using Isabelle/HOL, where before they thought Isabelle/HOL would be too difficult to learn.

Some challenging user interface issues arise where the student's expectation of how something should work is different from the goals of formal proof. For example, students appeared to hope that all the statements they have made so far in the proof would be remembered, and the checker would automatically determine which ones should be used to demonstrate the next statement; formal proofs, meanwhile, attempt to be explicit about their structure and which statements are involved in which steps.

Another challenge is developing automated systems that are simple enough for a student to understand roughly how they work, but that can make the same kind of steps that humans do when reasoning about a proof. The system needs to be able to verify human reasoning steps so that automated proof exercises do not have to differ too much from paper proofs. Students must be able to understand roughly how the reasoning system works because there are often proof steps that a reasoning system cannot verify and cannot disprove. Students need a mental model of why the system cannot verify a step, so they can change the step accordingly. Making the reasoning system understandable is especially challenging. In the proof exercises described in this paper, we use a very simple model of "triviality": there is a set of trivial rules. But even with this simple model, the sheer number of rules means that it is difficult for a student to know whether or not a proof step requires a non-trivial rule. With a more complex notion of triviality, it might become very difficult indeed for a user to understand why a step is not trivial to the reasoning system.

Structure based editing is a fairly long-standing technique for programming interfaces. However, MathsTiles is somewhat unusual in that the syntax of the interface does not directly match the syntax of the underlying language, and the syntax can vary from question to question (or even during the life of the question). Allowing tiles to be scattered on the canvas makes it simpler to work in a bottom-up manner than it is in many structure based editors, and allows sketching out of parts of answers. Whereas in most programming languages, code needs to be commented out or cut and paste into a notepad to detach it from the program without deleting it, a MathsTiles can simply be unplugged from its parent and left on the page.

Acknowledgements

Our research has been supported by the Cambridge-MIT Institute and the Cambridge Commonwealth Trust. The authors acknowledge the advice and assistance of their project partners Hal Abelson, Gerald Sussman and Chris Hanson from the Massachusetts Institute of Technology, and of Mark Ashdown, Kasim Rehman and Sparsh Gupta from the University of Cambridge.

We also thank the reviewers for the extensive and detailed feedback that they provided. This feedback was invaluable and helped us to improve the paper greatly.

References

1. Abel, A., B. Chang, and F. Pfenning: 2001, 'Human-readable machine-verifiable proofs for teaching constructive logic'. In: U. Egly, A. Fiedler, H. Horacek, and S. Schmitt (eds.): *Proceedings of the Workshop on Proof Transformations, Proof Presentations and Complexity of Proofs (PTP01).*

2. Andrews, P. B., C. E. Brown, F. Pfenning, M. Bishop, S. Issar, and H. Xi: 2004, ‘ETPS: A System to Help Students Write Formal Proofs’. *Journal of Automated Reasoning* **32**, 75–92.
3. Arefi, F., C. E. Hughes, and D. A. Workman: 1990, ‘Automatically generating visual syntax-directed editors’. *Commun. ACM* **33**(3), 349–360.
4. Aspinall, D., C. Lüth, and D. Winterstein: 2005, ‘Parsing, Editing, Proving: The PGIP Display Protocol.’. In: *International Workshop on User Interfaces for Theorem Provers 2005 (UITP’05).*
5. Baker, R. S., A. T. Corbett, K. R. Koedinger, and A. Wagner: 2004, ‘Off-task behavior in the cognitive tutor classroom: when students ”game the system”’. In: E. Dykstra-Erickson and M. Tscheligi (eds.): *Proceedings of ACM CHI 2004 Conference on Human Factors in Computing Systems.* pp. 383–390.
6. Ballarin, C. and G. Klein: 2004, ‘Introduction to the Isabelle Proof Assistant’. In: *Second International Joint Conference on Automated Reasoning.* Available from http://isabelle.in.tum.de/coursematerial/IJCAR04/index.html. Accessed 24 February 2007.
7. Barwise, J. and J. Etchemendy: 1994, *Hyperproof.* Stanford, California: CSLI Publications.
8. Baumgartner, P., U. Furbach, M. Groß-Hardt, and A. Sinner: 2004, ‘Living Book - Deduction, Slicing, and Interaction.’. *Journal of Automated Reasoning* **32**(3), 259–286.
9. Benzmüller, C., H. Horacek, I. Kruijff-Korbayová, M. Pinkal, J. Siekmann, and M. Wolska: 2007, ‘Natural Language Dialog with a Tutor System for Mathematical Proofs’. *Journal of Computer Science and Technology.* To appear.
10. Benzmüller, C., H. Horacek, H. Lesourd, I. Kruijff-Korbayova, M. Schiller, and M. Wolska: 2006, ‘A corpus of tutorial dialogs on theorem proving; the influence of the presentation of the study-material’. In: *Proceedings of International Conference on Language Resources and Evaluation (LREC 2006).* Genova, Italy. To appear.
11. Billingsley, W. and J. Billingsley: 2004, ‘The Animation of Simulations and Tutorial Clients for Online Teaching’. In: *Proceedings of the 15th Annual Conference for the Australasian Association for Engineering Education and the 10th Australasian Women in Engineering Forum, Toowoomba, Australia.* pp. 532 – 540.
12. Billingsley, W., P. Robinson, M. Ashdown, and C. Hanson: 2004, ‘Intelligent tutoring and supervised problem solving in the browser’. In: *Proceedings of the IADIS International Conference WWW/Internet 2004, Madrid, Spain.* pp. 806 – 811.
13. Blackwell, A. and T. Green: 2003, ‘Notational systems - the Cognitive Dimensions of Notations framework’. In: J. M. Carroll (ed.): *HCI Models, Theories and Frameworks.* Amsterdam, pp. 103 – 133.
14. Blackwell, A. and T. Green: 2007, ‘A Cognitive Dimensions Questionnaire’. Available online from http://www.cl.cam.ac.uk/ afb21/CognitiveDimensions/CDquestionnaire.pdf. Accessed 25 February 2007.
15. Brown, J. S., R. R. Burton, and A. G. Bell: 1975, ‘SOPHIE: A Step towards a reactive learning environment’. *International Journal of Man-Machine Studies* **7**, 675–696.

16. Clark, J. (ed.): 1999, *XSL Transformations (XSLT) Version 1.0.* World Wide Web Consortium. Accessed 13 August 2006 from http://www.w3.org/TR/1999/REC-xslt-19991116.
17. Clark, J. and S. DeRose (eds.): 1999, *XML Path Language (XPath) Version 1.0.* World Wide Web Consortium. Accessed 30 January 2005 from http://www.w3.org/TR/1999/REC-xpath-19991116.
18. Conati, C., A. S. Gertner, and K. VanLehn: 2002, 'Using Bayesian networks to manage uncertainty in student modeling'. *User Modelling and User-Adapted Interaction* **12**(4), 371–417.
19. Craig, S. D., X. Hu, B. Gholson, W. Marks, A. C. Graesser, and T. T. R. Group: 2000, 'AutoTutor: A human tutoring simulation with an animated pedagogical interface'. In: P. Hamberger (ed.): *Proceedings of the International Society for Optical Engineering: Integrated Command Environments.*
20. Ehrensburger, J. and C. Zinn: 1997, 'DiaLog: A System for Dialogue Logic'. In: *Conference on Automated Deduction.* pp. 446–460.
21. Green, T. R. G. and M. Petre: 1996, 'Usability Analysis of Visual Programming Environments'. *Journal of Visual Languages and Computing* **7**.
22. Hansen, W. J.: 1971, 'Creation of Hierarchic Text with a Computer Display'. Technical report, Argonne National Laboratories.
23. Kelleher, C., D. Cosgrove, D. Culyba, C. Forlines, J. Pratt, and R. Pausch: 2002, 'Alice2: Programming without Syntax Errors'. In: *User Interface Software and Technology.*
24. Khwaja, A. A. and J. E. Urban: 1993, 'Syntax-directed editing environments: issues and features'. In: *SAC '93: Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing.* New York, NY, USA, pp. 230–237.
25. Ko, A. J., H. Aung, and B. A. Myers: 2005, 'Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks'. In: *International Conference on Software Engineering.*
26. Kohlhase, M.: 2000, 'OMDoc: Towards an Internet Standard for the Administration, Distribution and Teaching of Mathematical Knowledge'. In: *AISC 2000 Artificial Intelligence and Symbolic Computation Theory.* pp. 32–52.
27. Lamport, L.: 1995, 'How to Write a Proof'. *The American Mathematical Monthly* **102**(7), 600–608.
28. Lesa, L. and K. Yacef: 2002, 'An Intelligent Teaching System for Logic'. In: *Intelligent Tutoring Systems : 6th International Conference, ITS 2002, Biarritz, France and San Sebastian, Spain, June 2-7, 2002.*
29. Lukins, S., A. Levicki, and J. Burg: 2002, 'A Tutorial Program for Propositional Logic with Human/Computer Interactive Learning'. In: *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education.* pp. 381–385.
30. Melis, E., E. Andrès, J. Büdenbender, A. Frischauf, G. Goguadze, P. Libbrecht, M. Pollet, and C. Ullrich: 2001, 'ActiveMath: A Generic and Adaptive Web-Based Learning Environment'. *International Journal of Artificial Intelligence in Education* **12**(4), 385–407.
31. Miller, P., J. Pane, G. Meter, and S. Vorthmann: 1994, 'Evolution of Novice Programming Environments: the Structure Editors of Carnegie Mellon University'. *Interactive Learning Environments* **4**(2), 140–158.
32. Nipkow, T.: 2003, 'Structured Proofs in Isar/HOL'. In: H. Geuvers and F. Wiedijk (eds.): *Types for Proofs and Programs (TYPES 2002).* pp. 259 – 278. Also available online from http://www4.informatik.tu-muenchen.de/ nipkow/pubs/types02.pdf accessed 10 June 2005.

33. Nipkow, T.: 2006, ‘A Compact Introduction to Isabelle/HOL’. Available from http://isabelle.in.tum.de/coursematerial/Shanghai06/index.html. Accessed 24 February 2007.
34. Nipkow, T., L. C. Paulson, and M. Wenzel: 2002, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Vol. 2283 of *LNCS*. Springer.
35. Rehman, K., W. Billingsley, and P. Robinson: 2006, ‘Writing questions for an Intelligent Book using external AI’. In: *Proceedings of the Sixth International Conference on Advanced Learning Technologies (ICALT2006)*. pp. 1089 – 1091.
36. Scheines, R. and W. Sieg: 1994, ‘Computer Environments for Proof Construction’. *Interactive Learning Environments* **4**(2), 159–169.
37. Slind, K., S. Barrus, S. Choe, C. Condrat, J. Duan, S. Gopalakrishnan, A. Knoll, H. Kuwahara, G. Li, S. Little, L. Liu, S. Moore, R. Palmer, C. Tuttle, S. Walton, Y. Yang, and J. Zhang: 2005, ‘Teaching a HOL Course: Experience Report’. In: J. Hurd, E. Smith, and A. Darbari (eds.): *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*. pp. 170–179.
38. Sommer, R. and G. Nuckols: 2004, ‘A proof environment for teaching mathematics’. *Journal of Automated Reasoning* **32**, 227 – 258.
39. Stallman, R. and G. J. Sussman: 1977, ‘Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis’. *Artificial Intelligence* **9**, 135–196.
40. Teitelbaum, T. and T. Reps: 1981, ‘The Cornell Program Synthesizer: a syntax-directed programming environment’. *Communications of the ACM* **24**(9), 563–573.
41. Van Der Hoeven, J.: 2001, ‘GNU TeXmacs: a free, structured, wysiwyg and technical text editor’. *Le document au XXI-ième siècle* **39–40**, 39–50.
42. Wenzel, M.: 1999, ‘Isar - a Generic Interpretative Approach to Readable Formal Proof Documents’. In: *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs’99*. Also available online from http://www4.in.tum.de/ wenzelm/papers/Isar-TPHOLs99.pdf accessed 10 June 2005.
43. Wenzel, M.: 2005, *The Isabelle/Isar Reference Manual*. TU München.
44. Wiedijk, F.: 2004, ‘Formal proof sketches’. In: *Types for Proofs and Programs*, Vol. 3085/2004 of *LNCS*. Springer.
45. Winer, D.: 1999, *XML-RPC Specification*. UserLand Software. Accessed 30 January 2005 from http://www.xmlrpc.com/spec.