Proof search with proof variables in Maude

Joseph Lynch

MComp (hons) Computer Science and Mathematics
The University of Bath
May 2020

Proof search with proof variables in Maude

Submitted by: Joe Lynch

Abstract

We introduce a new variant of proof search, and discuss the development of an automated deduction system for this. This variant of proof search includes proof variables, which give some structure of the solution. In particular, it is made possible by deep inference. The context of the theorem proving is open deduction. This is a formalism of deep inference, a methodology for designing formalisms, such that derivations can be composed by the same connectives as formulae. The system is written in Maude, and relies on Maude's Strategy Language to control the proof search techniques. This has been further combined with a graphical proof editor in Java. Specifically, the proof search is focused on subatomic logic within deep inference, and we discuss the support for this both in Maude and the graphical proof editor.

PROOF SEARCH WITH PROOF VARIABLES IN MAUDE

JOSEPH LYNCH

Master of Computing in Computer Science and Mathematics
The University of Bath
May 2020

Abstract. We introduce a new variant of proof search, and discuss the development of an automated deduction system for this. This variant of proof search includes proof variables, which give some structure of the solution. In particular, it is made possible by deep inference. The context of the theorem proving is open deduction. This is a formalism of deep inference, a methodology for designing formalisms, such that derivations can be composed by the same connectives as formulae. The system is written in Maude, and relies on Maude's Strategy Language to control the proof search techniques. This has been further combined with a graphical proof editor in Java. Specifically, the proof search is focused on subatomic logic within deep inference, and we discuss the support for this both in Maude and the graphical proof editor.

1. Introduction

Proof search is used as a general term to describe the automated process of finding a proof to some theorem. In relation to formalisms, this consists of automatically finding a derivation from a premiss to a conclusion, through the application of inference rules. These inference rules are dependent on the proof system being used.

The key result of this paper is the implementation of a new variant of proof search. In particular, this variant uses proof variables, which provide more information about the resultant structure of the proof. This has been implemented in open deduction [1], a formalism of deep inference. Deep inference is a methodology for designing formalisms, with the key notion that terms can be rewritten at depth inside formulae. As a consequence, we can compose not only formulae by connectives, but also derivations. This form of proof search using proof variables is a consequence of the properties of deep inference.

Maude [7] is used for the implementation of proof search. It is a high-performance reflective language and system that supports equational and rewriting logic. The grammar of a proof system is defined in a functional module by equations, while inference rules are represented as rewrite rules in a system module. The strategy language written within strategy modules is a new addition, and 'allows the definition of strategy expressions that control the way a term is rewritten' [12]. As a result of this, we can reduce the time complexity of proof search, by imposing restrictions on rewriting.

The open deduction proof editor is a graphical user interface that lets us deconstruct open deduction derivations step-by-step. We can perform operations on a derivation, and see the results graphically. The proof editor works with user-defined proof systems written in Maude, and XML. Subatomic logic, and the ability to move between it and regular logic, has also been added to the GUI. Furthermore, proof search with proof variables has been implemented. To support this, we can now enter whole derivations and view the graphical results of proof search directly.

2. Preliminaries

2.1. **Deep Inference.** Deep inference is a general concept of structural proof theory for designing formalisms. These formalisms are a generalisation of one-sided sequent calculus, and can be seen as an 'extreme form of linear logic' [3]. The main theory behind deep inference is that we can compose proofs by the same connectives as formulae. As a consequence, inference rules can be applied deep inside formulae. This differs from Gentzen formalisms, where inference rules can only be applied at the root of formulae.

One motivation for deep inference is the effect it has on proof size. This is linked with proof search because the size of proofs affects the size of the proof search space directly. Thus, the time to find a proof is reduced. It has been proved that deep inference has an exponential speed-up over Gentzen on analytic proof systems [10]. In particular, the Statman tautologies [14] have exponential-size proofs in the cut-free sequent calculus, but polynomial proofs in cut-free deep inference.

Open deduction is a formalism of deep inference, where Figure 1 is a valid proof, and the derivations

$$\frac{A}{C} \quad \text{and} \quad \frac{B}{D}$$

are being composed by a connective. In both deep inference derivations in Figure 1 and 2, the subformulae C and D are rewritten inside the formula $C \vee D$. Figure 2 is the calculus of structures, a special case of open deduction, and illustrates the bureaucracy that can occur when not composing derivations by connectives.

$$\frac{A}{C} \vee \frac{B}{D}$$

Figure 1. Open deduction

$$\cfrac{\cfrac{A \vee B}{A \vee D}}{C \vee D} \quad \text{and} \quad \cfrac{\cfrac{A \vee B}{C \vee B}}{C \vee D}$$

Figure 2. Calculus of structures

A proof system for classical logic that is used with deep inference is SKS [3], the rules of which can be seen in Figure 3. A weaker, but equal, variation of this has been implemented with the proof search system. This variation was originally proposed by Schäfer [13].

$$
\mathsf{ai}{\downarrow}\,\frac{\mathrm{t}}{a \vee \bar{a}} \qquad \mathsf{ai}{\uparrow}\,\frac{a \wedge \bar{a}}{\mathrm{f}}
$$

$$
s\,\frac{(A \vee B) \wedge C}{(A \wedge C) \vee B} \qquad m\,\frac{(A \wedge B) \vee (C \wedge D)}{(A \vee C) \wedge (B \vee D)}
$$

$$
\mathsf{ac}{\downarrow}\,\frac{a \vee a}{a} \qquad \mathsf{ac}{\uparrow}\,\frac{a}{a \wedge a}
$$

$$
\mathsf{aw}{\downarrow}\,\frac{a}{\mathrm{f}} \qquad \mathsf{aw}{\uparrow}\,\frac{a}{\mathrm{t}}
$$

FIGURE 3. Proof system SKS.

2.2. **Subatomic Logic.** Subatomic logic is a new methodology within the framework of deep inference. It is called subatomic because it goes 'inside' the atoms. An example of a subatomic formula in classical logic is

$$((\mathrm{f}\ a\ \mathrm{t}) \vee (\mathrm{t}\ a\ \mathrm{t})) \wedge (\mathrm{t}\ b\ \mathrm{f}).$$

With respect to the atom a, the formulae f a t and t a f can be understood as superpositions of the values t and f. In this case, we take f a t to be the positive atom a, and t a f to be the negative atom $\bar{a}$. One purpose of doing this is that we can present subatomic proof systems in which every rule is an instance of the rule scheme

$$\frac{(A\ \alpha\ B)\ \nu\ (C\ \beta\ D)}{(A\ \nu\ C)\ \alpha\ (B\ \gamma\ D)}$$

where $\alpha, \beta, \gamma, \nu$ are relations, and A, B, C, D are formulae. This provides us with a useful way to reason generally about proof systems. This subatomic approach is used to study normalisation procedures which ultimately obtain a normalisation theory.

The subatomic proof system SAKS [15] can be seen in Figure 4. It is possible to then translate any derivation in SKS into an equivalent derivation in SAKS, through a representation mapping R. The interpretation map I does the converse. If we take a as an atom, f, t as constants, and A, B as formulae, then the mappings for classical logic are

$$
\begin{array}{lll}
R(\mathrm{t}) = \mathrm{t}; & R(a) = \mathrm{f}\ a\ \mathrm{t}; & R(A \vee B) = R(A) \vee R(B); \\
R(\mathrm{f}) = \mathrm{f}; & R(\bar{a}) = \mathrm{t}\ a\ \mathrm{f}; & R(A \wedge B) = R(A) \wedge R(B);
\end{array}
$$

and

$$
\begin{array}{llll}
I(\mathrm{t}) = \mathrm{t}; & I(\mathrm{t}\ a\ \mathrm{t}) = \mathrm{t}; & I(\mathrm{f}\ a\ \mathrm{t}) = a; & I(A \vee B) = I(A) \vee I(B); \\
I(\mathrm{f}) = \mathrm{f}; & I(\mathrm{f}\ a\ \mathrm{f}) = \mathrm{f}; & I(\mathrm{f}\ a\ \mathrm{t}) = \bar{a}; & I(A \wedge B) = I(A) \wedge I(B);
\end{array}
$$

such that for every formula A, we have $I(R(A)) = A$.

$$
a\!\downarrow \frac{(A \vee B)\ a\ (C \vee D)}{(A\ a\ C) \vee (B\ a\ D)} \qquad a\!\uparrow \frac{(A\ a\ B) \wedge (C\ a\ D)}{(A \wedge C)\ a\ (B \wedge D)}
$$

$$
\wedge\!\downarrow \frac{(A \vee B) \wedge (C \vee D)}{(A \wedge C) \vee (B \vee D)} \qquad \vee\!\uparrow \frac{(A \vee B) \wedge (C \wedge D)}{(A \wedge C) \vee (B \wedge D)}
$$

$$
m \frac{(A \wedge B) \vee (C \wedge D)}{(A \vee C) \wedge (B \vee D)}
$$

$$
ac \frac{(A\ a\ B) \vee (C\ a\ D)}{(A \vee C)\ a\ (B \vee D)} \qquad a\bar{c} \frac{(A \wedge B)\ a\ (C \wedge D)}{(A\ a\ C) \wedge (B\ a\ D)}
$$

Figure 4. Proof system SAKS.

To demonstrate this, the interpretation of the subatomic formula,

$$
A = (((\mathrm{f} \wedge \mathrm{t})\ a\ \mathrm{t}) \vee \mathrm{t}) \wedge (\mathrm{t}\ b\ \mathrm{f}) \quad \text{is} \quad I(A) = (a \vee \mathrm{t}) \vee \bar{b}
$$

where $\mathrm{f} \wedge \mathrm{t}$ reduces to f, causing $\mathrm{f}\ a\ \mathrm{t}$ and $\mathrm{t}\ b\ \mathrm{f}$ to be interpreted to a and $\bar{b}$, respectively. Furthermore, this idea can be applied to inference rules. For example, where the occurrence of the SAKS rule $a\!\downarrow$ is interpreted to be the rule $ai\!\downarrow$ in SKS. Another example is the switch rule in SKS, which can be derived from rule $\wedge\!\downarrow$.

$$
\frac{(\mathrm{t} \vee \mathrm{f})\ a\ (\mathrm{t} \vee \mathrm{f})}{(\mathrm{t}\ a\ \mathrm{f}) \vee (\mathrm{t}\ a\ \mathrm{f})} \quad \xrightarrow{\text{interpret}} \quad \frac{\mathrm{t}}{a \vee a}
$$

It important to note that not every SAKS derivation can be interpreted as a valid SKS derivation. This is the case for formulae and derivations with atoms inside the scopes of other atoms. This occurs with the formula $((\mathrm{t}\ b\ \mathrm{f}) \wedge \mathrm{t})\ a\ \mathrm{f}$ where the atom b is occuring inside the scope of the atom a, and thus we cannot interpret it.

3. Proof Search

Maude is an effective language for a variety of applications, due to its simplicity, expressiveness and performance. It is particularly effective for reducing the time complexity of proof search with its uniquely fast rewrite speeds. Maude uses algorithms which perform one step of associative rewriting in constant time, and one step of associative-commutative rewriting in logarithmic time. This is opposed to polynomial time which was the prior standard [7]. In the context of proof search, if rules of a proof system are rewrite rules in Maude, then solely relying on fast rewriting leads to a rapid increase in time complexity. As such, it is not particularly feasible to search for non-trivial proofs. This is especially true for proof systems with rules that match any formula, such as the co-contraction rule in SKS. This is because the rule will be iteratively applied to every possible subformula in a proof search scenario.

Therefore, some method of controlling the rewrite would be extremely useful. Maude has always provided limited functionality for this, simply due to its implementation of the meta-level, which addresses the reflective nature of rewriting logic. In fact, strategies were initially developed this way [9]. However, in December 2019 with the release of Maude 3.0, a strategy language was integrated at the object level, and in core Maude. This introduced strategy modules which are dedicated to controlling rewrites, and allow for the design of complex strategies in a more efficient way.

3.1. **Traditional Proof Search.**

Traditional proof search is a well known practice, and has been implemented in various different pieces of software in many contexts. For example, Lean [6] for automated theorem proving, and iProver [5] for theorem proving with first-order logic. Furthermore, the graphical proof editor GraPE [13], provides some functionality for proof search in the calculus of structures. It attempts to search from a premiss to the truth value `t` in SKS. This was done using Maude's meta-level equation `metaSearch`, which specifies the initial term, desired term, and bound. Additionally, it relied on the sequential nature of the calculus of structures.

In the traditional proof search case, my system takes a premiss and conclusion, both of which are formulae. Rules that have been defined in the given proof system, are then extracted to the meta-level and adapted to be able to match all possible occurrences. These rules can then be applied to the premiss, following the strategy that has been written. Currently, for the subatomic proof system SAKS, this follows the structure of applying all rules to the premiss with the exception of contraction and co-contraction, which are applied until a specified maximum depth. Additionally, there are limitations on how many times a particular formula and the results of the rewrite can be rewritten by that same rule.

3.2. **Proof Search with proof variables.**

Traditional proof search is clearly a useful tool, especially with the addition of strategies to control and confine rewrites of formulae. However even so, the computation time of proof search increases exponentially and thus the utility reduces.

In the field of deep inference, it is sometimes the case that some structure of the proof is known [2]. This is demonstrated in Figure 6, where unknown formulae and derivations are replaced with proof variables, denoted ϕ_i, $\forall i \in \mathbb{N}$. The given premiss and conclusion for this specific proof can be seen in Figure 5. The system uses this information to find the value of all proof variables, such that the derivation is sound. The resultant derivation is searched for using algorithms with Maude's meta-level and strategy language.

The addition of proof variables and information about the structure of the derivation, allows this type of proof search to be much more efficient than traditional proof search. This is because we are cutting off proof search trees earlier,

$$\dfrac{\boxed{\dfrac{(A\ a\ B)\vee(C\ a\ D)}{(A\vee C)\ a\ (B\vee D)}}\wedge\boxed{\dfrac{(E\ a\ F)\vee(G\ a\ H)}{(E\vee G)\ a\ (F\vee H)}}}{((A\vee C)\wedge(E\vee G))\ a\ ((B\vee D)\wedge(F\vee H))}$$

FIGURE 5. Given derivation.

$$\dfrac{\dfrac{((A\ a\ B)\vee(C\ a\ D))\ \wedge\ \boxed{\dfrac{\dfrac{(E\ a\ F)\vee(G\ a\ H)}{\phi_1\quad a\quad \phi_2}}{(\phi_3\ a\ \phi_4)\wedge(\phi_5\ a\ \phi_6)}}}{\boxed{\dfrac{(A\ a\ B)\wedge(\phi_7\ a\ \phi_8)}{(A\wedge\phi_{11})\ a\ (B\wedge\phi_{12})}}\vee\boxed{\dfrac{(C\ a\ D)\wedge(\phi_9\ a\ \phi_{10})}{(C\wedge\phi_{13})\ a\ (D\wedge\phi_{14})}}}}{\boxed{\dfrac{\phi_{15}}{(A\vee C)\wedge(E\vee G)}}\ a\ \boxed{\dfrac{\phi_{16}}{(B\vee D)\wedge(F\vee H)}}}$$

FIGURE 6. Derivation with proof variables.

and therefore stopping irrelevant trees from expanding needlessly. Clearly, the more detailed the derivation with proof variables, the more efficient the proof search will be.

The general overview of how this type of proof search works is rather intuitive, although not trivial to implement. Furthermore, there are various ways this could be implemented. The initial attempt at implementation did not use strategies and thus became far too complex and convoluted, both in the structure of the code, and the efficiency of the algorithm. Using Maude's strategy language allowed for more fine-grained control over the application of rules, and sped up the overall proof search.

Rules are applied to the conclusion, and then it is iteratively checked that the resultant derivations 'fit' proof variables in the given derivation. At each iteration any result that does not match the proof variable derivation is discarded.

$$\dfrac{\dfrac{\phi_1}{(A\vee C)\wedge\phi_2}}{(A\wedge E)\vee(C\wedge G)}$$

FIGURE 7. Derivation with two proof variables.

$$\dfrac{\dfrac{(A\wedge E)\vee(C\wedge G)}{(A\vee C)\wedge(E\vee G)}}{(A\wedge E)\vee(C\wedge G)}$$

FIGURE 8. One possible solution for the proof variable.

In Figure 7 we have a simple derivation consisting of three formulae, including two unknown proof variables. For this particular derivation, we must apply a rule to the conclusion $(A\vee C)\wedge(E\vee G)$, such that the premiss is the subformula $(A\vee C)$ and variable ϕ_2 composed by $\wedge$. Following this, any rule that can be applied to the root of the formula $(A \vee C) \wedge \phi_2$ will satisfy the proof variable ϕ_1. One solution, of the many possible solutions, is shown in Figure 8. We have applied the rule m to the formula $(A \vee C) \wedge (E \vee G)$, and obtained the formula $(A \wedge E) \vee (C \wedge G)$, and then returned to $(A \vee C) \wedge (E \vee G)$ by the rule m.

$$\begin{array}{c} (A \wedge (E \vee G)) \vee (C \wedge (E \vee G)) \\ \hline (A \wedge C) \wedge \boxed{\begin{array}{c} (E \vee G) \vee (E \vee G) \\ \| \\ (E \vee G) \end{array}} \\ \hline (A \wedge E) \vee (C \wedge G) \end{array}$$

FIGURE 9. Another possible solution for the proof variable.

So far proof variables have been shown to represent formulae, but they also represent derivations. In the general case without any restrictions, this means that the proof size is unbounded. However, in practice we only require certain specified rules to generate derivations, that are not explicitly specified. This functionality has been implemented in the system. User-specified rules are applied deep within formulae to generate derivations in the place of proof variables. Moreover, we put restrictions on the depth and number of rule applications that create these derivations. This process adds complexity to the implementation, but improves the utility of the system. An example of a proof variable representing a derivation is in Figure 9. We rewrite the formulae $(A \wedge E) \vee (C \wedge G)$ using rule m, to obtain the formula $(A \wedge C) \vee (E \wedge G)$, much like the case of Figure 8. However, the rule $c\downarrow$ is then applied one level down to the subformula $E \vee G$, which replaces the proof variable ϕ_2 with a derivation. The proof is then completed by applying the rule m and replacing ϕ_1 with the formula $(A \wedge (E \vee G) \vee (C \wedge (E \vee G))$.

3.3. **Implementation in Maude.** The purpose of strategies is to control rewriting. It is effectively the strategy of rewriting from one term to another. This is extremely useful. Clearly when proof searching it is necessary to apply rewrite rules in many different ways and combinations. This is especially true when dealing with deep inference, where rules can be applied anywhere.

However, as mentioned previously, rewriting without any restrictions has limited use. Certain combinations of rules, and the application of rules at certain depths might not be desirable. Furthermore, these undesirable scenarios can increase the computation time of proof search exponentially. This is where strategies are useful, because they restrict rewriting applications to avoid such situations. Moreover, the details of what these undesirable situations are, and when they occur, can be adapted and edited by the user of the system.

3.3.1. *Maude Definitions.* The Maude implementation of proof systems used with the calculus of structures was proposed and implemented by Kahramanoğulları [4]. Extensions were made to this by Schäfer with the development of

GraPE [13]. The concept is that the grammar of a proof system is defined in a functional module. Functional modules 'define data types and operations on them by means of equational theories' [7]. In the case of SAKS, the relations in the functional module have been defined as

```
op [_,_] : Structure Structure -> Structure [comm assoc] .
op {_,_} : Structure Structure -> Structure [comm assoc] .
op ___   : Structure Atom Structure -> Structure .
```

where the operations `[_,_]` and `{_,_}` represent the connectives $\vee$ and $\wedge$, respectively. Additionally the operation `___` represents a subatomic formula such as $(\mathrm{t} \vee \mathrm{f})\ b\ \mathrm{f}$. A system module 'specifies a rewrite theory' [7], and thus we define rewrite rules in a system module to represent inference rules. One such rule in SAKS would be

```
rl [a_down] : [ A atm C , B atm D ] => [ A , B ] atm [ C , D ] .
```

Furthermore, the strategy module represents the strategy expressions used during rewriting. They extend the expressiveness of the language by means of recursive and mutually recursive strategies. We implement a new operation core in Maude to represent open deduction derivations. This is based on the concept of `premiss >[ rule ]> conclusion`, and is the core of how strategies work with the whole derivation.

```
op _>[_]>_ : Structure Qid Structure -> Deriv [gather (e & E)] .
```

The reasoning for this is that now derivations can be dealt with directly in Maude. GraPE and previous versions of this proof editor, used Maude to work with single formulae. Strategies are declared by the keyword `strat` and defined by `sd`. They consist of rule applications, however further restrictions can be made. It is also possible to test a condition on the subject term by

`match Term s.t. Condition`

There are further capabilities such as `matchrew` which allows rewriting of specified subterms. These abilities are complemented by the various control combinators, such as $\alpha;\beta$ which executes strategy α and then strategy β. Additionally, $\alpha|\beta$ executes α or β, while $\alpha\,?\,\beta : \gamma$ is a conditional ternary operator. To simplify, some common ternary operator scenarios are replaced by keywords such as `or-else`, equivalently $\alpha\,?\,\mathit{idle} : \gamma$, where idle returns the term with no change. Therefore, the strategy `rles` can be understood as attempting to apply all the specified rules of SAKS to the subject term. As with all strategies, it then returns the set of all results.

```
strat rles : @ Structure .
sd rles := a_down | a_up | conj_down | disj_up | m | ac | acc | eq .
```

3.3.2. *Strategies.* Given a derivation with proof variables, the software will attempt to build a new derivation that fills in the proof variables with sound formulae. When this built derivation fits with the given derivation, and has the same conclusion and premiss, then the solution has been found. The user can enter a derivation as defined in Maude,

```
((\{[a,c],[e,g]\} >[Q1]> phi1) a (\{[b,d],[f,h]\} >[Q2]> phi2))
    >[Q3]> [\{a,[e,g]\} a \{b,[f,h]\},\{c,[e,g]\} a \{d,[f,h]\}]
```

which represents

$$\frac{((a \wedge (e \vee g))\ a\ (b \wedge (f \vee h))) \vee ((c \wedge (e \vee g))\ a\ (d \wedge (f \vee h)))}{\boxed{\dfrac{\phi_1}{(a \vee c) \wedge (e \vee g)}}\ a\ \boxed{\dfrac{\phi_2}{(b \vee d) \wedge (f \vee h)}}}\ .$$

FIGURE 10. Derivation with proof variables to be solved.

Initially, the meta strategy rewrite equation `metaSrewrite` is used to call the strategy init on the meta-level term of the conclusion.

```
metaSrewrite(['SamStr], upTerm(enwrite(conclusion)),
    'init[[upTerm(upTerm(deriv)),upTerm(0)]], breadthFirst, N)
```

The full derivation in Figure 10 is also passed to the strategy as an argument. In addition to these, breadth-first searching is specified, as opposed to depth-first. The following strategies are then applied.

```
sd init( Trm, N ) := solve(Trm) ; init(Trm, N + 1) .
sd solve( Trm ) := isSoln( Trm ) or-else
    (app ; match S s.t. isMatch(upTerm(unwrite(S)), expand(Trm))) .
sd isSoln( Trm ) := match S s.t. isMatch(upTerm(unwrite(S)), Trm) .
```

It is important to note that the strategy `init` is being executed on the subject term (the conclusion), which is not referenced by a variable. In comparison, the full proof variable derivation is passed to the strategy as the variable `Trm`. The strategy `solve` checks if the subject term is a solution. If so then we are done, if not then `app` applies rules. We use the equation `isMatch` to iteratively check if the resultant derivations are solutions, and whether they are compatible with the proof variables in the given derivation.

Within `app`, rule applications and strategies are applied. For example, in Figure 10, `rles` applies the inference rules of SAKS on the subject term $((a \vee c) \wedge (e \vee g))\ a\ ((b \vee d) \wedge (f \vee h))$ and returns the set of possible derivations. At the same time, two further strategies are also applied to this subject term, namely `depth` and `applyRles`.

```
sd app := rles | ( depth(2) ; applyRles ) .
```

These strategies are applied sequentially, and are used to reduce the time complexity of the proof search. In particular, `depth` limits how deep the rules contraction and co-contraction of SAKS are applied within subformulae.

Truncated versions of the Maude strategies for bounding the depth of these rules are below.

```
csd depthSch( N, K ) := der( N, K ) or-else binop( N, K )
                or-else matchrew F:Formula by F using spec
    if N =/= K .
sd der( N , K ) :=
    matchrew S >[Q]> T by
      T using  der(N,K) or-else (spec | depthSch(N + 1,K)) .
sd binop( N , K ) :=
    matchrew [S,T] by
      S using (der(N,K) or-else (spec | depthSch(N + 1,K))) | idle,
      T using der(N,K) or-else (spec | depthSch(N + 1,K))   .
sd binop( N , K ) :=
    matchrew {S,T} by
      S using (der(N,K) or-else (spec | depthSch(N + 1,K))) | idle,
      T using der(N,K) or-else (spec | depthSch(N + 1,K))
```

The strategy **depthSch** uses **matchrew**. The syntax of which is

$$\texttt{matchrew}\ X_1, ..., X_n\ \texttt{s.t. by}\ X_1\ \texttt{using}\ \alpha, \texttt{...}\ , X_n\ \texttt{using}\ \gamma.$$

where different strategies are applied to each subterm in $X_1, ..., X_n$. Variations such as `amatchrew` exist for matching subterms at depth. However the ability for bounding the depth is not currently supported. Therefore, with `binop` we match binary operators such as `[_,_]`, at the top level of the subject term. We then apply the selected rewrite rules to the matched subterms, and recursively do so at each level of depth. How the depth of the strategy search is bound by `N` and `K`. Once this process has been completed, and `app` has been fully applied, there will be a set of solutions. For example,

$$\boxed{m\,\frac{(a\wedge e)\vee(c\wedge g)}{(a\vee c)\wedge(e\vee g)}}\;\; a\;\; \boxed{m\,\frac{(b\wedge f)\vee(d\wedge h)}{(b\vee d)\wedge(f\vee h)}}$$

and,

$$\boxed{m\,\frac{(a\wedge(e\vee g))\vee(c\wedge(e\vee g))}{(a\wedge c)\wedge\boxed{\begin{array}{c}(e\vee g)\vee(e\vee g)\\ c\!\downarrow\!\Vert\\ e\vee g\end{array}}}}\;\; a\;\; \boxed{m\,\frac{(b\wedge(f\vee h))\vee(d\wedge(f\vee h))}{(a\wedge c)\wedge\boxed{\begin{array}{c}(f\vee h)\vee(f\vee h)\\ c\!\downarrow\!\Vert\\ f\vee h\end{array}}}}\;.$$

Solutions incompatible with the proof variable derivation are then discarded. This process is recursively applied to the derivations that were possible solutions, until a match for the given proof variable derivation is found.

Various other methods were used to help reduce the time complexity of proof search. For example the strategy `applyRles` is used after `depth`, instead of `rles`. This ensures that solutions of `depth` are always at least partially included in term being rewritten. The test `amatchrew` is used to mimic the normal application of rules. It applies rules to the root of any subformula, except if the subformula does not include a result of `depth`.

```
sd applyRles:=amatchrew S s.t.(not S::Rewritable) by S using toprles
```

3.3.3. *Meta-level matching.* The meta-level equation `metaMatch` is at the core of how we compare the set of possible solutions with the given proof variable derivation. In particular, the equation `expand` defined in the strategy module strips down the proof variable derivation. This is in order to prepare it to be iteratively matched against the set of solutions given by the strategies.

```
op expand : Term -> TermList .
eq expand( '_>`[_`]>_[ Trm1 , Q , Trm2 ] ) =
   unpack-deriv(expand(Trm1), Q, (expand(Trm2),upTerm(nowt)), 0) .
eq expand( op [ Trm1, Trm2 ] ) =
   unpack-structure(op, expand(Trm1), expand(Trm2), 0) .
eq expand('___[ Trm1, cnst, Trm2 ] ) =
   unpack-subatom(expand(Trm1), cnst, expand(Trm2), 0) .
eq expand( Trm ) = Trm .
```

The following block of code is the `isMatch` equation that is used in the strategies `solve` and `isSoln` seen in Section 3.3.2. It checks if the solution has been found, and with `isMatch(upTerm(unwrite(S)),expand(Trm))` also checks if derivations in the set of solutions are valid.

```
op isMatch : Term TermList QidList -> Bool .
eq isMatch( Trm, TL, Rles ) =
    itr(remove-specs(Trm1, Rles, 0), TL,0) .

op itr : Term TermList Nat -> Bool .
eq itr( Trm1, empty, N ) = false .
eq itr( Trm1, (Trm,TL), N ) =
   if metaMatch(['SamStr], Trm, Trm1, nil, 0) =/= noMatch then
       true
   else
       itr( Trm1, TL, N + 1 )
   fi .
```

This whole process, when applied to the derivation with proof variables in Figure 6, results in the completed derivation in Figure 11. Furthermore, in Figure 12 we can see the equivalent output of the derivation in the Maude command line. Where `P:Structure` are called variables in Maude, and represent the proof variables of our derivation.

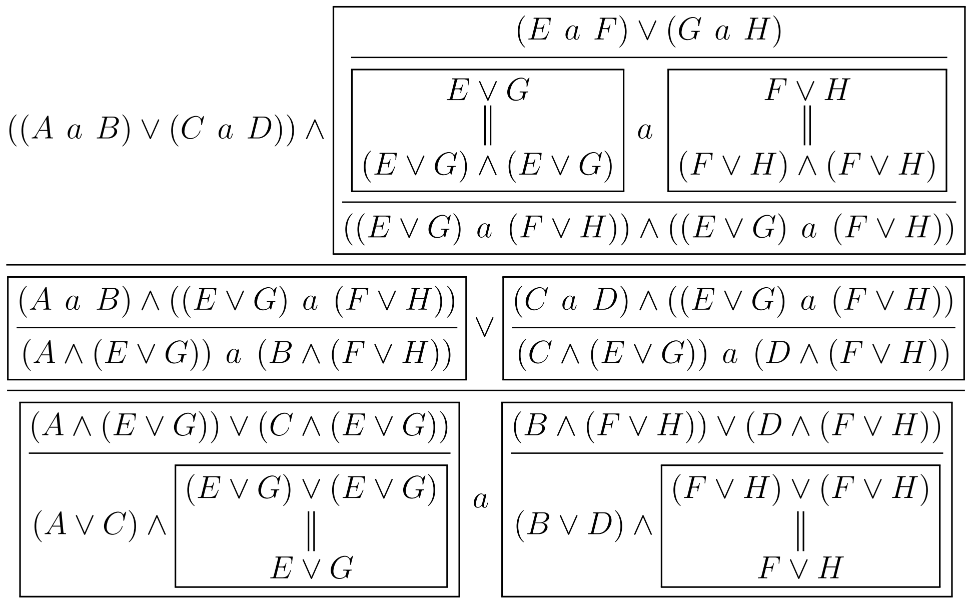
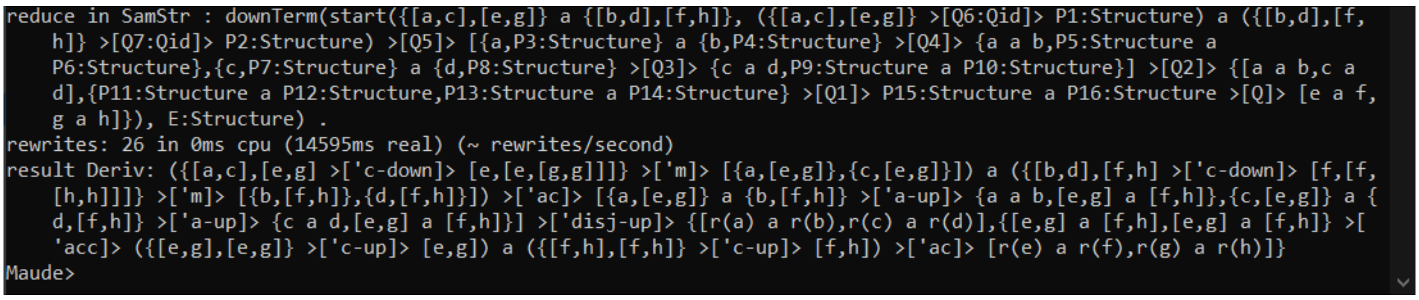


FIGURE 11. Solution to derivation with proof variables.



```
reduce in SamStr : downTerm(start({[a,c],[e,g]} a {[b,d],[f,h]}, ({[a,c],[e,g]} >[Q6:Qid]> P1:Structure) a ({[b,d],[f,
    h]} >[Q7:Qid]> P2:Structure) >[Q5]> [{a,P3:Structure} a {b,P4:Structure} >[Q4]> {a a b,P5:Structure a
    P6:Structure},{c,P7:Structure} a {d,P8:Structure} >[Q3]> {c a d,P9:Structure a P10:Structure}] >[Q2]> {[a a b,c a
    d],{P11:Structure a P12:Structure,P13:Structure a P14:Structure} >[Q1]> P15:Structure a P16:Structure >[Q]> [e a f,
    g a h]}), E:Structure) .
rewrites: 26 in 0ms cpu (14595ms real) (~ rewrites/second)
result Deriv: ({[a,c],[e,g] >['c-down]> [e,[e,[g,g]]]} >['m]> [{a,[e,g]},{c,[e,g]}]) a ({[b,d],[f,h] >['c-down]> [f,[f,
    [h,h]]]} >['m]> [{b,[f,h]},{d,[f,h]}]) >['ac]> [{a,[e,g]} a {b,[f,h]} >['a-up]> {a a b,[e,g] a [f,h]},{c,[e,g]} a {
    d,[f,h]} >['a-up]> {c a d,[e,g] a [f,h]}] >['disj-up]> {[r(a) a r(b),r(c) a r(d)],{[e,g] a [f,h],[e,g] a [f,h]} >[
    'acc]> ({[e,g],[e,g]} >['c-up]> [e,g]) a ({[f,h],[f,h]} >['c-up]> [f,h]) >['ac]> [r(e) a r(f),r(g) a r(h)]}
Maude>
```

FIGURE 12. Proof search with proof variables in the Maude command line interface.

4. GUI

The GUI is capable of displaying and deconstructing open deduction derivations, in user-defined proof systems. A proof system is implemented as a Maude and XML file. At the frontend, the derivations are stored in Java as tree data structures, and are graphically displayed with Java Swing [11]. When a user performs an operation, such as applying an inference rule, the derivation trees are parsed to be Maude-readable. Maude then computes the relevant operations on the derivation, and returns the result to be displayed.

Various features have been added to the GUI that were previously missing. These includes the ability to undo and redo while constructing proofs, and the ability to switch between the 'regular' view and subatomic view of a derivation. Proof search for derivations with proof variables has also been implemented, by the press of a button. To support this feature, derivations with or without proof variables can now be entered as input and directly graphically displayed.

4.1. **Derivations as input.** The proof editor has always dealt with derivations, but required the user to enter a formula and construct them. In Section 3.3.1 we introduced a new operation in Maude

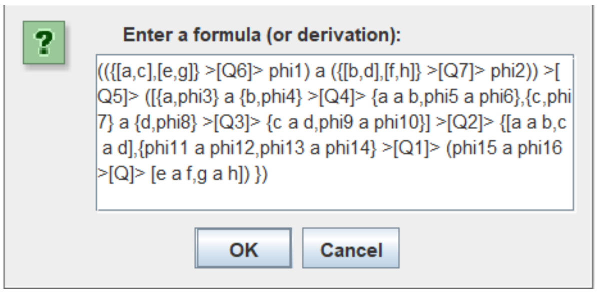
```
_>[_]_> : Structure Qid Structure -> Deriv .
```

The ability to parse this and correctly integrate it with the current system has been implemented. A consequence of this is that we can now directly enter a derivation as input and view it graphically. For example the user may enter the derivation `a >['c_down]> [a,a]`.

4.2. **Proof Search.** The ability to input derivations complements the implementation of proof search with proof variables nicely. Further functionality has been added to both Maude and the GUI, such that the proof editor can correctly deal with proof variables and unknown rules. An example of a derivation being directly entered is in Figure 13. The derivation entered is:

```
(({[a,c],[e,g]} >[Q6]> phi1) a ({[b,d],[f,h]} >[Q7]> phi2)) >[Q5]>
([{a,phi3} a {b,phi4} >[Q4]> {a a b,phi5 a phi6},{c,phi7} a {d,phi8}
>[Q3]> {c a d,phi9 a phi10}] >[Q2]> {[a a b,c a d],{phi11 a phi12,
phi13 a phi14} >[Q1]> (phi15 a phi16 >[Q]> [e a f,g a h])})
```

The current format is that `Q1`, . . ., `Qn` denote unknown rules, while `phi1`, . . ., `phin` denote proof variables. Note that `a, b, c, d, e, f, g, h` are all atoms here, and can each be replaced by any formula, such as `[tt,{a,ff}]`, or simply the units `tt` and `ff`.



Enter a formula (or derivation):

(({[a,c],[e,g]} >[Q6]> phi1) a ({[b,d],[f,h]} >[Q7]> phi2)) >[
Q5]> ([{a,phi3} a {b,phi4} >[Q4]> {a a b,phi5 a phi6},{c,phi
7} a {d,phi8} >[Q3]> {c a d,phi9 a phi10}] >[Q2]> {[a a b,c
a d],{phi11 a phi12,phi13 a phi14} >[Q1]> (phi15 a phi16
>[Q]> [e a f,g a h]) })

FIGURE 13. Entering proof variable derivation.

The derivation is then normalised by regular expressions in Java and equational theories in Maude, before being graphically displayed, as in Figure 14.

The user may then press the 'Proof Search' button, which uses Maude at the backend to apply the proof search techniques discussed in Section 3.3.2. While searching the status bar changes, the cursor becomes busy, and the user has the ability to stop the search by pressing the red circle. When the solution is found, the derivation is displayed, this can be seen in Figure 15.

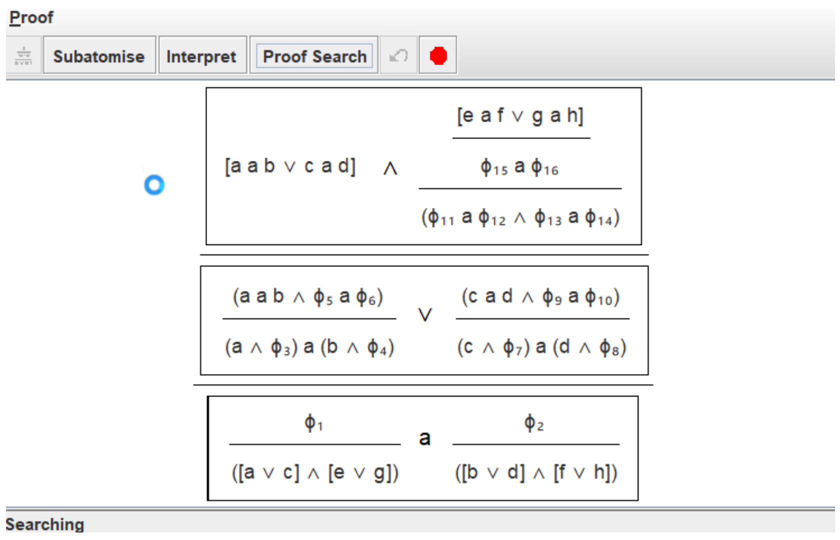


FIGURE 14. Graphical derivation with proof variables.

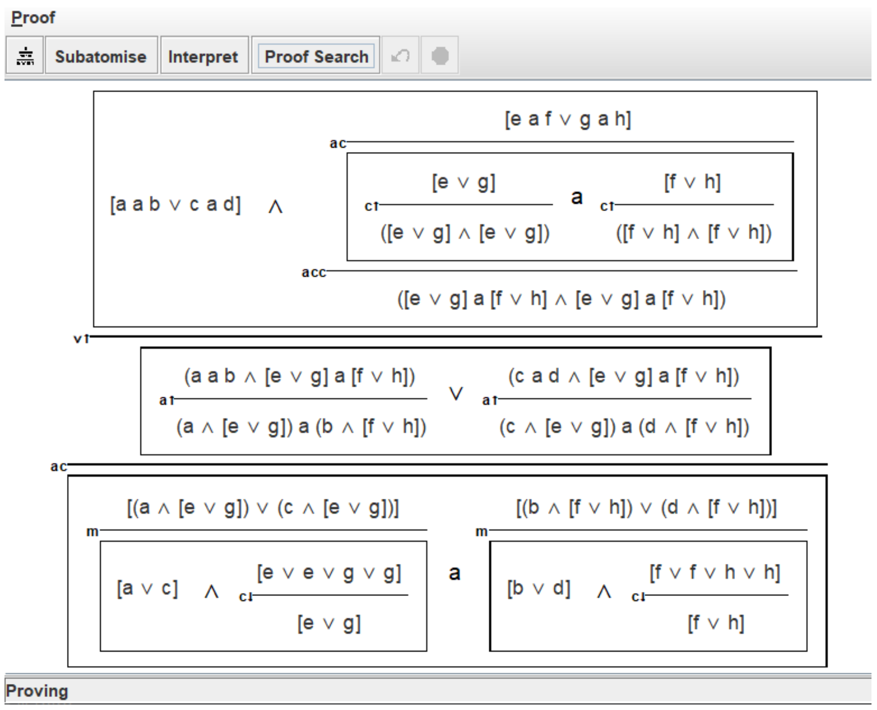


FIGURE 15. Graphical derivation of the solution.

4.3. **Subatomic Logic.** The implementation of SAKS consisted of creating the relevant Maude and XML files, and integrating these with Java. The key changes made to the Java code was to support the transition between regular and subatomic logic, SKS and SAKS, respectively. There are now buttons in the top left to swap between the regular and subatomic view of the derivation.

If the user enters a formula `[{a,b},-a]`, and presses the ‘subatomise’ button, the formula is replaced with `[{ff a tt,ff b tt},tt a ff]`. The converse occurs with the interpret button. When one of these buttons is pressed, the derivation is passed to Maude, where meta-level equations are applied. The resultant derivation is then passed back to the frontend to be displayed.

The interpretation and representation maps discussed in Section 2.2, have been implemented into Maude as functional modules. The subatomic derivation of meta-terms, passed from Java, is reduced within these modules. The action of translating from SKS to SAKS was slightly more complex. This is because we attempt to replace each rule in SKS with its equivalent in SAKS. Therefore, the process consisted of using the representation mapping [15], along with meta-level equations to check for the validity of the derivations that had been found. For example, we can construct the proof in Figure 16, then press the subatomise button and represent it as Figure 17.

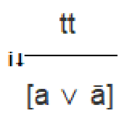


FIGURE 16. Rule instance in SKS.

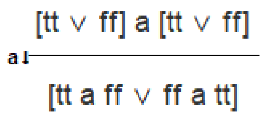


FIGURE 17. Rule instance in SAKS.

The meta-level equations are used to attempt to correctly translate the derivation. The unit `tt` in Figure 16 is rewritten to a subatomic formula that fits the subatomic rule in SAKS. This is instead of being simply rewritten to `tt`.

An important thing to note is that the GUI makes the underlying proof system change when translating between the types of logic. This lets us convert the regular formula `[{a,b},-a]` to the subatomic formula `[[{f a t,f b t},t a f],tt]`. We can then construct Figure 18, and finally interpret this back to the SKS derivation in Figure 19.

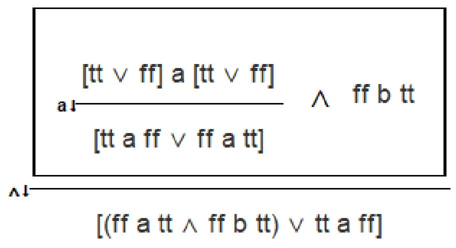


FIGURE 18. SAKS derivation.

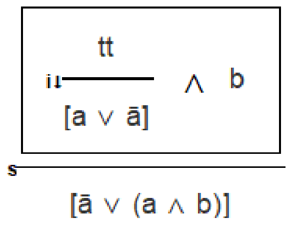


FIGURE 19. SKS derivation.

In addition to this, the derivation in Figure 21 can be constructed through the application of two inference rules. The ’subatomise’ button may then be pressed to produce the derivation in Figure 20. This is an example of where the system attempts to find a SAKS derivation that is a valid representation.

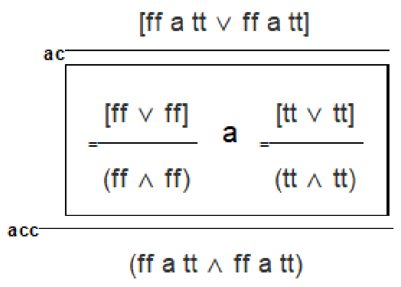


FIGURE 20. SAKS derivation.

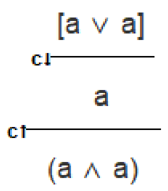


FIGURE 21. SKS derivation.

There are however currently limitations on applying the representation mapping to all forms of derivations in SKS. This in particular includes the weakness rules, as well as various other derivations.

4.4. **Undo and Redo.** An undo button was present in GraPE [13] for derivations of the calculus of structures, but was previously lacking from the open deduction proof editor. Therefore, the ability to undo, and now also redo, rule applications has been implemented. This is clearly a vital tool for constructing derivations step by step. These actions are performed by the arrow buttons that can be seen in Figure 15.

5. FUTURE WORK

In terms of future work, there are a wide range of opportunities to extend the results discussed here. In particular, the implementation of proof search in Maude could be improved by refining the algorithms. This in turn would reduce the amount of time to find a solution. While the proof editor is already generalised for proof systems, the proof search functionality does not yet support this. Proof search with proof variables is currently implemented for SAKS. Therefore, a potential extension is rewriting strategies at the meta-level and removing dependencies of any one proof system.

Additional enhancements could be made to the written Maude code. It is fair to say that there is a learning curve, and that 'Maude is challenging' [8]. Therefore, a better separation between the modules defining proof systems and the meta-level code can be enforced. This would make the system more developer-friendly, for those who want to implement their own proof systems.

Currently the system has limitations when transitioning between 'regular' and subatomic logic. Therefore, more development on the representation and interpretation of derivations would add value to the proof editor.

Regarding the graphical user interface, there are a number of features that could be added. Traditional proof searching without proof variables could be added as a functionality. Furthermore, this proof editor relies on some of the architecture of GraPE, which supported the ability to produce derivations compilable by TeX. Therefore, this functionality could be adapted such that it works with the open deduction proof editor. More complex extensions could be added, such as keeping track of subformulae during rule applications in order to produce atomic flows.

References

[1] A. Guglielmi, T. Gundersen, M. Parigot. 'A Proof Calculus Which Reduces Syntactic Bureaucracy'. In: *Leibniz International Proceedings in Informatics (LIPIcs)* 6 (2010). Ed. by C. Lynch. Proceedings of the 21st International Conference on Rewriting Techniques and Applications, pp. 135–150. ISSN: 1868-8969. DOI: `10.4230/LIPIcs.RTA.2010.135`.

[2] T. Barrett. *Confirmation report.* Personal Communication. University of Bath, May 2020.

[3] A. Guglielmi. 'Deep Inference'. In: *All About Proofs, Proof for All.* Mathematical Logic and Foundations 55 (2015). Ed. by B. Paleo and D. Delahaye, pp. 164–172. DOI: `10.1017/S1471068415000125`.

[4] O. Kahramanoğulları. 'Maude as a Platform for Designing and Implementing Deep Inference Systems'. In: *Electronic Notes in Theoretical Computer Science* 219 (2008). Proceedings of the Eighth International Workshop on Rule Based Programming (RULE 2007), pp. 35–50. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2008.10.033`.

[5] K. Korovin. 'iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description)'. In: *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings.* Ed. by Alessandro Armando, Peter Baumgartner and Gilles Dowek. Vol. 5195. Lecture Notes in Computer Science. Springer, 2008, pp. 292–298. DOI: `10.1007/978-3-540-71070-7\_24`.

[6] L. de Moura, S. Kong, J. Avigad, F. van Doorn, J. van Raumer. 'The Lean Theorem Prover'. In: *25th International Conference on Automated Deduction (CADE-25)* (2015). URL: `https://leanprover.github.io/papers/system.pdf`.

[7] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott. *Maude Manual 3.0.* URL: `http://maude.cs.illinois.edu/w/images/e/ee/Maude-3.0-manual.pdf`.

[8] T. McCombs. *Maude 2.0 Primer.* 2003. URL: `http://maude.cs.illinois.edu/w/images/6/63/Maude-primer.pdf`.

[9] N . Martí-Oliet, J. Meseguer, A. Verdejo. 'Towards a Strategy Language for Maude'. In: *Electronic Notes in Theoretical Computer Science* 117 (2005). Proceedings of the Fifth International Workshop on Rewriting Logic and Its Applications (WRLA 2004), pp. 417–441. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2004.06.020`.

[10] P. Bruscoli, A. Guglielmi. 'On the Proof Complexity of Deep Inference'. In: *CoRR* abs/0709.1201 (2007). arXiv: `0709.1201`. URL: `http://arxiv.org/abs/0709.1201`.

[11] R. Eckstein, M. Loy, D. Wood, J. Elliott, B. Cole. *Java Swing, 2nd Edition*. O'Reilly Media, 2009.

[12] S. Eker, N. Martí-Oliet, J. Meseguer, I. Pita, R. Rubio, A. Verdejo. *Strategy language for Maude*. URL: `http://maude.sip.ucm.es/strategies/` (visited on 25/04/2020).

[13] M. Schäfer. *The design and implementation of the GraPE Graphical Proof Editor*. 2006. URL: `http://grape.sourceforge.net/grape.pdf`.

[14] R. Statman. 'Bounds for proof-search and speed-up in the predicate calculus'. In: *Annals of Mathematical Logic* 15.3 (1978), pp. 225–287. ISSN: 0003-4843. DOI: `https://doi.org/10.1016/0003-4843(78)90011-6`. URL: `http://www.sciencedirect.com/science/article/pii/0003484378900116`.

[15] A. Aler Tubella. *A study of normalisation through subatomic logic*. 2017. URL: `http://cs.bath.ac.uk/ag/aat/phd.pdf`.

This form must be attached to the dissertation as an appendix.

Department of Computer Science

12-Point Ethics Checklist for UG and MSc Projects

Student	Joseph Lynch
Academic Year or Project Title	2019/20 - XX40211
Supervisor	Alessio Guglielmi

Does your project involve people for the collection of data other than you and your supervisor(s)? YES / [✓ NO]

If the answer to the previous question is YES, you need to answer the following questions, otherwise you can ignore them.

This document describes the 12 issues that need to be considered carefully before students or staff involve other people ('participants' or 'volunteers') for the collection of information as part of their project or research. Replace the text beneath each question with a statement of how you address the issue in your project.

1. *Have you prepared a briefing script for volunteers?* YES / NO

   Briefing means telling someone enough in advance so that they can understand what is involved and why – it is what makes informed consent informed.

2. *Will the participants be informed that they could withdraw at any time?* YES / NO

   All participants have the right to withdraw at any time during the investigation, and to withdraw their data up to the point at which it is anonymised. They should be told this in the briefing script.

3. *Is there any intentional deception of the participants?* YES / NO

   Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

4. *Will participants be de-briefed?* YES / NO

   The investigator must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. This phase might wait until after the study is completed where this is necessary to protect the integrity of the study.

5. *Will participants voluntarily give informed consent?* YES / NO
Participants MUST consent before taking part in the study, informed by the briefing sheet. Participants should give their consent explicitly and in a form that is persistent –e.g. signing a form or sending an email. Signed consent forms should be kept by the supervisor after the study is complete.

6. *Will the participants be exposed to any risks greater than those encountered in their normal work life (e.g., through the use of non-standard equipment)?* YES / NO
Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life.

7. *Are you offering any incentive to the participants?* YES / NO
The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

8. *Are you in a position of authority or influence over any of your participants?* YES / NO
A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. *Are any of your participants under the age of 16?* YES / NO
Parental consent is required for participants under the age of 16.

10. *Do any of your participants have an impairment that will limit Their understanding or communication?* YES / NO
Additional consent is required for participants with impairments.

11. *Will the participants be informed of your contact details?* YES / NO
All participants must be able to contact the investigator after the investigation. They should be given the details of the Supervisor as part of the debriefing.

12. *Do you have a data management plan for all recorded data?* YES / NO
All participant data (hard copy and soft copy) should be stored securely, and in anonymous form, on university servers (not the cloud). If the study is part of a larger study, there should be a data management plan.