Carnap: An Open Framework for Formal Reasoning in the Browser

Graham Leach-Krouse
Department of Philosophy
Kansas State University Manhattan, KS 66506
gleachkr@ksu.edu

This paper presents an overview of *Carnap*, a free and open framework for the development of formal reasoning applications. Carnap's design emphasizes flexibility, extensibility, and rapid prototyping. Carnap-based applications are written in Haskell, but can be compiled to JavaScript to run in standard web browsers. This combination of features makes Carnap ideally suited for educational applications, where ease-of-use is crucial for students and adaptability to different teaching strategies and classroom needs is crucial for instructors. The paper describes Carnap's implementation, along with its current and projected pedagogical applications.

1 Introduction

In what follows we describe the Carnap framework, a free[1] and open framework for the development of formal reasoning applications. One of the goals of the Carnap framework is to provide instructors with flexible toolkit of interactive online exercises, widgets, and activities which can introduce students to rigorous formal reasoning, while minimizing or eliminating pedagogical roadblocks extraneous to the subject matter.

To that end, Carnap's design emphasizes flexibility and rapid prototyping, incremental presentation of formal reasoning concepts, and web browser-based deployment. Carnap's flexibility makes it possible for instructors to adapt Carnap-powered applications to fit the distinctive needs of their own institutions, students, and teaching styles. Incremental presentation of concepts via interactive exercises embedded in instructional materials helps students focus on what's important rather than being overwhelmed by complexity. And browser-based delivery of applications (most of Carnap's applications are compiled from Haskell to JavaScript, allowing them to run in any standard web browser) removes the early-semester hurdles associated with software installation, and improves accessibility for technologically anxious students as well as students who rely on libraries or computer labs for hardware access.

Carnap is under continuing development and is already actively used. In the Fall semester of 2017, Carnap was the main tool for three different courses at Kansas State University (Introduction to Formal Logic, Symbolic Logic I, and Modal Logic). It has also been used supplementally at the University of Birmingham, and is currently in use at the University of North Carolina-Chapel Hill.

Anecdotal evidence suggests that Carnap is an effective tool for logic instruction. At Kansas state, student evaluations of classes incorporating Carnap have consistently been in the top decile of courses university-wide, which is unusual for a course that is both quantitative and regarded by students as fairly challenging. Of course, a larger data set and more careful study will be required before making any definite claims about pedagogical effectiveness.

[1] GPLv3-licenced.

P. Quaresma and W. Neuper (Eds.): 6th International Workshop
on Theorem proving components for Educational software (ThEdu'17)
EPTCS 267, 2018, pp. 70–88, doi:10.4204/EPTCS.267.5

However, the following advantages of the framework are clear: a web-based interface for interactive exercises is highly accessible to students and removes barriers related to compatibility and installation presented by other forms of logic-teaching software. And, a flexible core makes it possible to provide for a wide range of different forms of logic instruction, empowering instructors to teach in whatever way they feel will be most effective for their students.

In Sections 2 and 3 below, we describe the Carnap framework, together with the theoretical background that has informed its design.[2] We go on, in Section 4 to describe several current educational applications of the framework. We close in Section 5 with a discussion of the Carnap framework's longer-term prospects and projected future applications.

2 Description of the Carnap Framework

The Carnap framework consists of three main components. The first of these is *Carnap-Core*, a set of Haskell libraries which provide:

(a) data types for a generic higher-order abstract syntax, henceforth *Carnap's syntax*;

(b) functions applicable to realizations of Carnap's syntax that allow for the semantic interpretation, parsing, display, and manipulation of all languages based on this syntax;

(c) generic unification algorithms based on the manipulations provided by the functions above;

(d) data types for a number of particular languages—in particular, languages for propositional logic, propositional modal logic, first-order logic, quantified modal logic, monadic second-order logic, and polyadic second-order logic—all implemented as realizations of Carnap's abstract syntax;

(e) data types for a number of logics for these languages;[3] and

(f) generic proof-checking methods applicable with these logics.

The second main component of the Carnap framework is *Carnap-GHCJS*, a set of browser-based educational applications, trans-compiled from Haskell to JavaScript via the GHCJS compiler [22]. These include:

(a) an embeddable "proof-checker" widget, which uses Carnap's core libraries to check the correctness of proofs in any of the logics provided by the core, and to provide continuous line-by-line feedback on proofs as they are constructed;

(b) a "rule-builder" widget which allows students to create and save derived inference rules for use in later work;

(c) a JSON API provider, which allows native JavaScript applications to pass JSON proof representations to Carnap for analysis; and

(d) several semantic exercise widgets, including widgets for checking student formalizations of natural language statements, and for checking student application of semantic methods (e.g. truth tables).

The third component of the framework is *Carnap-Server*, a web-server application based on the Yesod web-development framework [21]. This application hosts an interactive textbook and instructor-generated assignments containing interactive exercises, both incorporating the widgets above. It also manages user

[2] Interactive demonstrations of a few applications can be found at [14].

[3] By *logic* here, we mean a set of inference rules, rather than a set of theorems. So, there are many different propositional logics which can prove all and only the arguments whose validity can also be demonstrated by truth tables.

accounts for students and instructors, and collects, organizes, and grades submitted student work. In classes at Kansas state, students generally work problems by hand in the classroom, where it's possible for the instructor to provide immediate feedback.[4] Outside-of-class assignments are provided by Carnap-Server, with the Carnap-GHCJS applications supplying rapid feedback and encouraging students to correct errors before submitting work.

The libraries provided by Carnap-Core have made it possible to rapidly extend the Carnap Framework to support a wide range of languages and logics. At the time of writing, Carnap can check proofs in three different styles of natural deduction—Fitch style proofs, Montegue style proofs, and Hardegree style proofs—in nine different formal languages, with support for nineteen different logics, including the propositional and first-order logics of several well-known open-source logic textbooks and a wide range of relative modal logics.[5]

3 Theoretical Background

In this section, we briefly describe the theoretical background that has informed Carnap's design. In particular, we will detail the design principles that have motivated several of the choices mentioned above, and will expand upon the implementation of Carnap's generic higher-order abstract syntax and unification methods.

3.1 Design Principles

The primary motivation for the development of the Carnap framework has been the desire to provide an educational proof assistant that is simultaneously *flexible* enough to provide for the needs of a broad curriculum in formal reasoning and *accessible* enough for use both by students at the very introductory levels where the majority of logic instruction takes place and by educators who are not necessarily also software developers.

The demand for flexibility motivates an emphasis on genericity in the basic components of the framework. If our goal is to cover not just elementary propositional and first-order logic, but also perennial topics in philosophical logic, such as modality, plural and higher-order quantification, set theoretic foundations of mathematics, abstraction-theoretic foundations of mathematics, formal theories of truth, and so on, we will need to be able to handle a wide variety of different syntaxes and rules of inference. Writing ad-hoc proof-checking methods for each of these different logics would be exhausting and would result in bloated and unmaintainable software. Hence, there is a need for a notion of syntactical data general enough to encompass the many formal reasoning systems that Carnap aims to accommodate, and for proof-checking algorithms generic enough to be applied in combination with any of the sets of rules of inference that occur in the target systems.

The demand for accessibility motivates the browser-first approach that the Carnap framework favors for user interactions, rather than the command-line or dedicated application approaches used by the majority of proof-assistants.[6] Even advanced students can have trouble installing and configuring new software. With a large class of beginning students, simply getting everyone started with a proof-checking

[4] A few have taken to using Carnap in the classroom, on tablets or laptops, for a little extra guidance while working out problems. This has been permitted.

[5] The systems are described in more detail in Hardegree's freely available online text, Introduction to Modal Logic [10].

[6] One notable exception to this generalization is documented in [20]. However, even this project requires the installation of Flash and Java browser plugins, which are increasingly omitted from modern browsers because of their long and well-publicized history of security problems.

program can take days and can be a constant source of friction throughout the semester. Directing students to access a webpage, on the other hand, is quite painless. Introducing a proof assistant through a series of widgets with increasingly extensive functionality also seems to expose students to much less shock and anxiety than providing them with a single monolithic interface right out of the box.

Ordinarily these two demands—flexibility and accessibility—and the approaches adopted to address them—genericity and a browser-first approach to user interactions—would be in direct tension with one another. The main programming language currently available for browser-based applications is JavaScript. But JavaScript is not designed for generic programming or for the development of large scale applications.[7] One possible remedy would be to do the proof-checking work server-side, rather than client-side. But proof-checking can be computationally demanding, and genericity often comes at the cost of efficiency. So the server-side solution will not scale.

Carnap's use of Haskell overcomes this apparent tension. Haskell's type system includes a number of features that support highly generic programming. The Haskell ecosystem also provides excellent support for basic tasks like parsing and non-deterministic computation that are fundamental to proof-checking. Recently, several projects have attempted to trans-compile Haskell code to JavaScript.[8] The GHCJS compiler is currently the most ambitious of these. It allows for most of Haskell's libraries and any language extension supported by the GHC compiler to be used in writing applications that run in standard web-browsers. By using Haskell in combination with the GHCJS compiler, Carnap achieves both genericity and accessibility. And by distributing the computational load of generic proof-checking across a large number of clients, Carnap's basic model can smoothly scale to accommodate very large numbers of students without any noticeable degradation in performance.

3.2 Syntactic Data

In this section, we explain Carnap's generic syntax, showing how it provides a framework capable of accommodating a wide variety of formal languages. In the first part, we informally describe the basics in the abstract, without requiring any particular familiarity with Haskell or functional programming. In the next two parts, we describe the implementation of the generic syntax. Care has been taken to avoid as much technicality as possible in the latter two parts. However, some familiarity with the Haskell programming language is assumed. These more detailed discussions can be skipped without loss of continuity.

3.2.1 General Approach

An abstract syntax of propositional logic might be informally presented in the following way:

> The language of propositional logic has a lexicon consisting of a countable infinity of sentence letters and the five Boolean connectives: $\wedge$,$\vee$,$\rightarrow$, $\leftrightarrow$, and $\neg$.
>
> The main syntactic category of our language is *formula*.
>
> A formula is either: a single sentence letter, the result of applying one of the binary Boolean connectives $\wedge$,$\vee$,$\rightarrow$ or $\leftrightarrow$ to a pair of formulas, or the result of applying the unary connective $\neg$ to a formula

[7] For example, `import` statements were only introduced to the ECMAScript standard as recently as 2015, and are only now beginning to be implemented natively in browsers

[8] Besides GHCJS, there is also the Fay compiler [4] and the Haste compiler [8]. And besides these there are languages that attempt to be imitate Haskell's most important features while still compiling to JavaScript, like PureScript [9] and Elm [6].

It is tempting to analyze the above as an informal specification of a simple recursive data type, with each lexical item a constructor for that type. However, consider that an informal presentation of pure (monadic) first-order logic might then proceed to specify:

> The language of pure first-order logic extends the lexicon of propositional logic with a countable infinity of predicate letters, the quantifiers $\forall$ and $\exists$, and a countable infinity of term variables.
>
> The syntactic categories of first-order logic are *formula* and *term*
>
> All term variables are terms. Formulas are built as in propositional logic, and also by applying a predicate to a term, or applying a quantifier to bind a variable in a given formula.

Putting this second definition alongside the first calls our attention to a problem with the tempting analysis above. When, in the previous analysis, we took the language of propositional logic for a recursive data type and took, e.g. the connective $\neg$ for a constructor, we committed ourselves to thinking of $\neg$ as a function taking a formula of propositional logic as input and returning a formula of propositional logic. But in the language of first-order logic, at least informally, we find that same connective being applied, now to formulas of *first-order* logic, and now returning formulas of first-order logic.

Of course, we could hold on to our original analysis and deny that the $\neg$ of propositional logic is the same constructor as the $\neg$ of first-order logic, regarding talk about "extending the lexicon" of propositional logic as a mere façon de parler. But this will lead to a peculiar case of conceptual double-vision, as we're constantly forced to distinguish the two almost indistinguishable negations. And, in the context of designing a computer program, we'll have an ugly duplication of code that will only worsen as we consider other languages containing negation.

Carnap's generic syntax avoids this problem by rejecting the analysis of $\neg$ as the constructor of a recursive data type and, correlatively, rejecting the analysis of languages as simple recursive data types. Instead, Carnap attempts to mirror the informal practice (suggested by the language specifications above) of treating different connectives and lexical categories as portable across languages. Carnap does this by treating a connective like $\neg$ as the inhabitant of a non-recursive type[9] of Boolean connectives. Features of the connective that are invariant across languages (it applies to elements of the syntactic category "formula"; it is rendered prefix as "$\neg$"; in first-order semantics it reverses truth value... etc.) are encoded at the level of this type. Other lexical items are treated similarly—each lexical category (categories like *Boolean connective* and *sentence letter*) can be thought of as a "module", to be freely combined with other modules. Languages are produced by "snapping together" modules[10] and then transforming the resulting collection of non-recursive data-types into a single recursive data-type using a fixed-point construction (detailed in Section 3.2.3).

Compared to the analysis of a language as a recursive data type, Carnap's modular approach is somewhat more cumbersome when dealing with just a single language. But the Carnap framework is intended to support *multiple* languages. In this situation, the modular approach greatly reduces duplication of code. Instead of specifying what each language is, we specify what a variety of language modules are, and then specify a language by simply stipulating which modules are "snapped together" to make that language. Algorithms for working on languages can also be designed generically, either at the level of individual modules (for example, a conjunctive-normal-form algorithm that can be applied to any language containing Boolean connectives), or even at the level of the structure that all languages have in common in virtue of the way that they are constructed from modules (for example, a unification algorithm that can be applied to any language—see Section 3.3).

[9] Technically, not a concrete type, but a higher-order type-constructor.

[10] Essentially, by taking a direct sum of type-constructors.

3.2.2 Two examples

To flesh out the general remarks of the previous section, we present two illustrations of how Carnap represents formal languages: the language of propositional logic, and the language of first order logic.

Carnap's representation of a propositional language like the one specified in the previous section would look like this:

```
data Formula

data Connective lang type where
  And :: Connective lang (Formula → Formula → Formula)
  Or  :: Connective lang (Formula → Formula → Formula)
  If  :: Connective lang (Formula → Formula → Formula)
  Iff :: Connective lang (Formula → Formula → Formula)
  Neg :: Connective lang (Formula → Formula)

data Sentence lang type where
  Sentence :: Int → Sentence lang Formula

type PropositionalLexicon = Sentence :|: Connective

type PropositionalLanguage = FixLang PropositionalLexicon
```

We can see a rough correspondence between the informal presentation just given and the code here. We have two sorts of lexical items (sentences and connectives) in the informal definition. In the code, each of those two is given a data type (`Sentence lang type`, and `Connective lang type` respectively). We had one basic syntactic category—Formula—in the informal presentation, which corresponds here to the data type `Formula`.[11] The syntactic type of each lexical item is visible in the second argument of its Haskell type. So the syntactic type of a sentence is `Formula`, while the syntactic type of a binary connective, which takes two formulas to make a formula, is `Formula → Formula → Formula`. The two lexical data types are combined into a *lexicon* using the `:|:` operator. The language is then produced by applying the `FixLang` operator to the lexicon, which performs the fixed-point construction that turns the non-recursive lexicon type into a language with recursive formation rules.

We'll save investigation of `Fixlang` and the `:|:` operator for the next section. For now, we'll instead focus on how treating each sort of lexical item as a separate "module" and treating a language as a sum of different modules allows us to streamline our code and avoid repetition and boilerplate.

The informal presentation of the language of first-order logic described it as extending the lexicon of propositional logic with a few new constructors. Carnap's representation can have first-order logic be exactly that:

```
data Term

data Predicate lang type where
  Pred :: Int → Predicate lang (Term → Formula)

data Quantifier lang type where
  All :: Quantifier ((Term → Formula) → Formula)
  Some :: Quantifier ((Term → Formula) → Formula)
```

[11] Notice that this type has no constructors. It's a "phantom" type, used exclusively for organizing other types

```
data Variable lang type where
  Var :: Int → Variable lang Term

type FirstOrderLexicon =  PropositionalLexicon
                      :|: Predicate
                      :|: Quantifier
                      :|: Variable

type FirstOrderLanguage = FixLang FirstOrderLexicon
```

The fact that the syntactic categories of the language of first-order logic (namely `Term` and `Formula`) extend those of our propositional language allows Carnap to automatically infer how the Boolean connectives that make up part of `PropositionalLexicon` extend their domain when they are lifted to `FirstOrderLexicon`. Hence, we avoid the repetitive boilerplate involved in respecifying the propositional parts of the language. Furthermore, functions defined on `Connective` and lifted from there to `PropositionalLanguage` can now be lifted to `FirstOrderLanguage` automatically. We also inherit, for both languages, all functions that have been defined to apply to the inhabitants of arbitrary fix-points `FixLang a`.

3.2.3 The fundamental data types

To see how what was described in the previous section is possible, we'll need to do a bit more to dissect `:|:` and `Fixlang`. This section will contain a certain amount of unavoidable technicality, for which we beg the reader's patience.

The three fundamental data types (simplified for the sake of readability—in particular, we omit a lambda-abstraction operator from `Copula` that adds some distracting complexity) supporting Carnap's generic syntax are as follows:[12]

```
data Copula lang t where
    (:$:) :: lang (t → t') → lang t → Copula lang t'

data (:|:) f g lang t where
    FLeft :: f lang t → (f :|: g) lang t
    FRight :: g lang t → (f :|: g) lang t

data Fix f t where
    Fx :: f (Fix f) t → Fix f t
```

Suppose we are given the language `PropositionalLanguage` described in the previous section. (Henceforth, for brevity, we'll refer to this language as `Lang`, we will refer to to the corresponding lexicon `PropositionalLex` as `Lex`, and we will refer to to the syntactic type `Formula` as `Form`.) Individual lexical items of `Lang` are of type `Lang` *a*, where *a* is determined according to how these lexical items combine syntactically. Suppose we are given two lexical items from `Lang`: a sentence letter `P` of type `Lang Form`, and a negation sign ¬ of type `Lang (Form → Form)`. The constructor `:$:` for the type `Copula Lang Form` can then be applied thus: `¬:$:P`, yielding a value of type `Copula Lang Form`.

[12] Carnap's generic syntax is based on an idea roughly analogous to the one presented in [23], but adds to this idea a phantom type which is used for keeping track of the syntactic category of the constructor.

Visually:

$$\dfrac{\dfrac{\neg :: \texttt{Lang}\ (\texttt{Form} \to \texttt{Form}) \qquad (\texttt{:\$:}) :: \texttt{lang}\ (\texttt{t} \to \texttt{t}') \to \texttt{lang}\ \texttt{t} \to \texttt{Copula lang t}'}{(\texttt{:\$:})\neg :: \texttt{Lang Form} \to \texttt{Copula Lang}\ (\texttt{Form} \to \texttt{Form})} \qquad \texttt{P} :: \texttt{Lang Form}}{\neg\texttt{:\$:P} :: \texttt{Copula Lang Form}}$$

Of course, one would expect that applying negation to an atomic formula should yield something of the same type as the original formula. The trick here is as follows. Having constructed `¬:$:P`, we then apply `FLeft` to it, yielding a value of type `(Copula :|: g) Lang Form`. And, recall from the definitions above, `Lang` is in fact an alias for the `FixLang Lex`, which is again an alias for the type `Fix (Copula :|: Lex)`. Hence, we can take our value of type `(Copula :|: Lex) Lang Form`, and apply the constructor `Fx`, yielding the desired value of type `Lang Form`. Visually (with dotted lines representing the rewriting of aliases):

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg\texttt{:\$:P} :: \texttt{Copula Lang Form} \qquad \texttt{FLeft} :: \texttt{f lang t} \to (\texttt{f} :|: \texttt{g})\ \texttt{lang t}}{\texttt{FLeft}(\neg\texttt{:\$:P}) :: (\texttt{Copula} :|: \texttt{g})\ \texttt{Lang Form}}}{\texttt{FLeft}(\neg\texttt{:\$:P}) :: (\texttt{Copula} :|: \texttt{g})\ (\texttt{Fix}\ (\texttt{Copula} :|: \texttt{Lex}))\ \texttt{Form}} \qquad \texttt{Fx} :: \texttt{f}\ (\texttt{Fix f})\ \texttt{t} \to \texttt{Fix f t}}{\texttt{Fx}(\texttt{FLeft}(\neg\texttt{:\$:P})) :: \texttt{Fix}\ (\texttt{Copula} :|: \texttt{Lex})\ \texttt{Form}}}{\texttt{Fx}(\texttt{FLeft}(\neg\texttt{:\$:P})) :: \texttt{Lang Form}}$$

This does seem like rather a lot of circumlocution. But Haskell's typechecking is static, so the majority of this work is done at compile time, and has no noticeable effect on performance.

We see now how languages can be put together. There's a contribution both from a universal component, `FixLang`, which ensures that all languages contain a copula for applying different lexical items to one another, and from a particularizing component: the lexicon specific to the language in question—in the case above, the propositional lexicon `Lex`.

This shows us how generic programming can be done over all languages constructed in Carnap. Since all lexical items have types that unify with `FixLang lex a`, generic functions acting on this type can be applied to all lexical items. Generic functions like this can be derived using Haskell's typeclass mechanisms. For example, a `show` function, rendering a lexical item as a string, could be derived generically for our hypothetical language using a little bit of boilerplate (written only once for all langauges) like this:

```
class Schematize f where
  schematize :: f a → [String] → String

instance Schematize (Fix lex) ⇒ Show (Fix lex a) where
  show x = schematize x []

instance Schematize (lex (Fix lex)) ⇒ Show (Fix lex) where
  schematize (Fx x) = schematize x

instance (Schematize (lex1 lang), Schematize (lex1 lang)) ⇒
  Schematize ((lex1 :|: lex2) lang a) where

  schematize (FLeft x) = schematize x
  schematize (FRight x) = schematize x
```

```
instance Schematize lang ⇒ Schematize (Copula lang) where
  schematize (x :$: y) xs = schematize x (schematize y [] : xs)
```

together with particular Schematize instances for particular lexical categories:

```
instance Schematize (Connective lang) where
  schematize(And) = λ(x:y:xs) → x ++ " ∧ " ++ y
  schematize(If)  = λ(x:y:xs) → x ++ "→" ++ y
  schematize(Or)  = λ(x:y:xs) → x ++ " ∨ " ++ y
  schematize(Iff) = λ(x:y:xs) → x ++ "↔" ++ y
  schematize(Neg) = λ(x:y:xs) → " ¬ " ++ x

instance Schematize (Sentence lang) where
  schematize(Sentence n) = λ_ → "P_" ++ show n
```

The key observation here is that now we can declare how to print different connectives once and for all, at the level of lexical modules—languages will then automatically inherit this information on the basis of the modules that make them up.

A great deal of functionality can be automatically derived in this way, including not just rendering of formulas, but also parsing, functions for semantic evaluation and, most importantly from a proof-checking perspective, algorithms for unification.

3.3 Unification Algorithms

From the perspective of proof-checking, the most significant piece of generic functionality derived for Carnap's languages is support for higher-order unification.[13] In this section, we explain how Carnap applies higher-order unification in order to allow for the declarative specification of logics. As in the previous section, a small amount of familiarity with Haskell will be assumed. Familiarity with basic logic and the simply typed λ-calculus will also be helpful. As above, however, this section can be skipped without loss of continuity.

The generic deriving mechanism described in the previous section suffices to provide a wide range of basic syntactic operations—including substitution, application, abstraction, η-expansion and β-normalization—for any language defined using `FixLang`. Carnap incorporates an implementation of Huet's algorithm as given by Dowek in [7]. The algorithm is written exclusively in terms of the basic syntactic operations that are generically provided for any language defined using `FixLang`.

Huet's algorithm is a (constructive) semi-decision procedure for *higher-order unification problems*, i.e. problems of the following form:

> Given a finite set of equations $t_i = u_i$ between terms of the simply-typed λ-calculus, does there exist a substitution θ, assigning terms to schematic variables, such that for each i, the results of applying θ to t_i, u_i have the same normal form?

The naive presentation of this algorithm is as a non-deterministic type-directed stateful computation. The non-determinism and stateful aspects of the algorithm can be faithfully reproduced as a pure computation in Haskell using a combination of the `LogicT` monad transformer [13] and the `State` monad. The type-directedness of the algorithm can be reproduced by using the syntactic types[14] described in the previous section, along with Haskell's `Data.Typeable` library. Huet's algorithm is the best possible, in the sense that the general problem of higher-order unification is only semi-decidable. However, the cases we deal

[13] Haskell has a standard library for first-order unification—`unification-fd`—but not for higher-order unification.

[14] E.g. `Formula` for sentence letters, and `Formula → Formula` for unary connectives.

with in proof checking (pattern-matching unification, with fairly simple terms) are in fact decidable, and in practice Huet's algorithm solves them quickly.

How is this applied to proof checking? Consider the naive classroom presentation of a common pair of inference rules, modus ponens and universal instantiation:

$$(\text{MP})\,\frac{\phi \to \psi \quad \phi}{\psi} \qquad (\text{UI})\,\frac{\forall x \phi(x)}{\phi(\tau)}$$

Verifying that a particular inference instantiates the rule MP is a matter of finding a solution to one of a pair of unification problems determined by the inference. For example, given the inference

$$\frac{P \wedge R \to Q \vee S \quad P \wedge R}{Q \vee S}$$

We have the unification problems:

$$\begin{aligned} P \wedge R \to Q \vee S &= \phi \to \psi \\ P \wedge R &= \phi \\ Q \vee S &= \psi \end{aligned} \qquad\qquad \begin{aligned} P \wedge R \to Q \vee S &= \phi \\ P \wedge R &= \phi \to \psi \\ Q \vee S &= \psi \end{aligned}$$

If we view ϕ, ψ as schematic variables and $P, Q, R, S, \to$ as constants[15] then the inference is an instance of the rule MP, so long as one of these unification problems has a solution—as the first obviously does, namely:

$$\phi \rightsquigarrow P \wedge R \qquad\qquad \psi \rightsquigarrow Q \vee S$$

Of course, this case required only first-order unification. We only encounter higher-order terms once we begin verifying inferences that are supported by essentially first-order inference rules. For example, the inference from $\forall x(R(x) \wedge Q(x))$ to $R(f(a)) \wedge Q(f(a))$ is an instance of UI just in case the unification problem

$$\begin{aligned} \forall x(R(x) \wedge Q(x)) &= \forall x \phi(x) \\ R(f(a)) \wedge Q(f(a)) &= \phi(\tau) \end{aligned}$$

has a solution, when ϕ, τ are regarded as schematic variables, $\forall, R, f, a$ as constants. Which it does:

$$\phi \rightsquigarrow \lambda x(R(x) \wedge Q(x)) \qquad\qquad \tau \rightsquigarrow f(a)$$

since $\lambda x(R(x) \wedge Q(x))(f(a))$ normalizes to $R(f(a)) \wedge Q(f(a))$.

So, the problem of checking whether inferences are instances of rules reduces to the problem of checking whether certain simple higher-order unification problems have solutions. As a result, it can be solved in complete generality by an application of Huet's algorithm.

Against the background of a generic implementation of higher-order unification, developing a checker for a new logic mostly amounts to simply declaring the inference rules that the logic contains. Carnap approaches this problem by extending the language of the logic with lexical modules for one or more lexical categories of schematic variables. So the lexicon of propositional logic is extended to

[15] Some of which, namely P, Q, R, S are essentially of the type `Formula`, and others, namely $\to$, are of higher types, in this case `Formula`→`Formula`→`Formula`.

```
type PropositionalLexicon = Sentence :|: Connective :|: SchematicSentence
```

In the actual implementation, we also keep track of a context for each proof, in order to handle assumptions that are later discharged, e.g. in conditional proofs. However, besides the addition of a schematic variable for the assumptions currently in scope, rules can be declared in Carnap in a way that more or less mirrors how you might write them on the board in a logic classroom. For example, here are the two rules from above (MP and UI)[16]

```
modusPonens = [ GammaV 1 ⊢ phin 1 .→. phin 2
              , GammaV 2 ⊢ phin 1
              ] ∴ GammaV 1 :+: GammaV 2 ⊢ phin 2

universalInstantiation = [ GammaV 1 ⊢ lall "v" (phi 1)]
                         ∴ GammaV 1 ⊢ phi 1 tau
```

In `modusPonens`, by using functions like `phin` (provided by generically derived typeclasses) for our various lexical items, we end up with a rule that can be applied to any language incorporating the lexical module that provides schematic sentence letters and the lexical module that provides Boolean connectives. Similarly for `universalInstantiation`.

The generic approach to inference checking makes it possible to rapidly develop a wide variety of formal systems (since one essentially just has to transcribe the rules from a textbook). This makes it practical to support systems of logic from a wide variety of common textbooks, including free and open-source textbooks which generally do not provide complementary proof-checking software.

4 Applications

In this section, we'll describe some current applications of Carnap. In the first part, we describe where and how Carnap is currently used, and how it has been received by students. In the second part, we give more detailed descriptions of several of the widgets provided by Carnap-GHCJS.

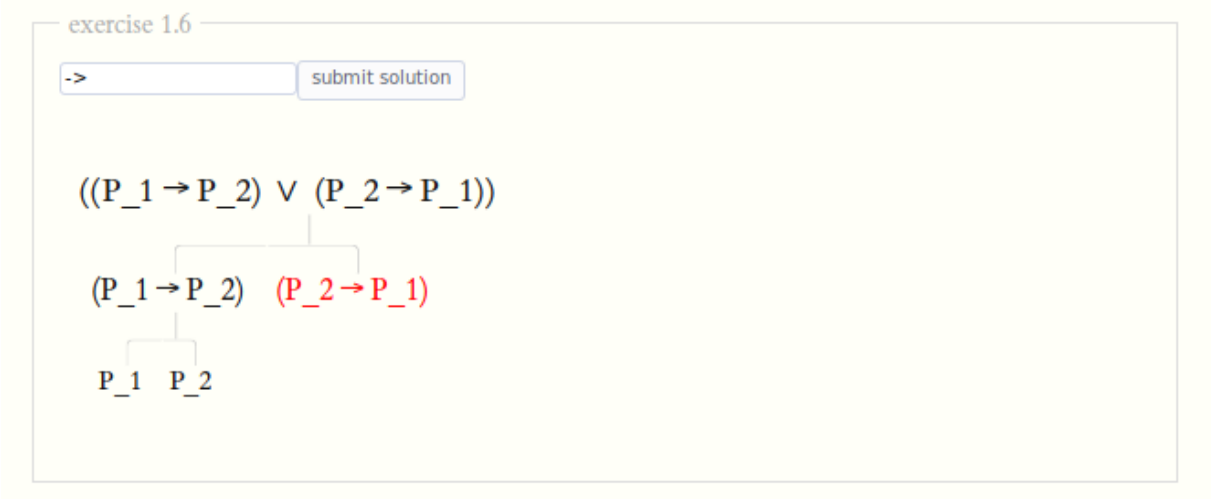
4.1 Adoption and Reception

Carnap has been used successfully in the philosophy department at Kansas State university to teach a majors-level symbolic logic course (PHILO320), an introductory course in logic aimed at non-majors (PHILO110), and an advanced independent study on modal logic (PHILO680). It has also been used and as a supplementary tool for teaching logic at the University of Birmingham in the UK, and is currently being used as the central tool for a course in mathematical logic at the University of North Carolina-Chapel Hill.

In PHILO320 and PHILO110, students are introduced to Carnap through an online textbook, *The Carnap Book* [16], provided by Carnap-Server at Carnap's main website. The chapters of the book are web-pages which incorporate JavaScript widgets generated by Carnap-GHCJS. These widgets present students with interactive exercises that challenge them to apply the concepts they acquire from the textbook as soon as possible after reading. The Carnap Book's interactive exercises offer rapid feedback on work, helping students zero in on the correct understanding of the ideas presented in the textbook while these ideas are still fresh. In PHILO680, assignments are provided to students on problem-sheet web-pages that also incorporate Carnap-GHCJS's widgets.

[16] We use Unicode function symbols to emphasize the similarity between Carnap's code for a rule and the informal blackboard presentation that that rule might receive.

Figure 1: Syntax-Checker Exercise



Both the assignment sheets for PHILO680, and the chapters of the Carnap Book, are generated by Carnap-Server from text files written in a version of the pandoc markdown language [17], extended with a small amount of extra syntax used to indicate the insertion of exercises. Using a markdown language greatly simplifies the process of generating and modifying assignments.

Student feedback at Kansas State has been extremely positive, with the course consistently receiving scores in the top decile for every category of evaluation.

4.2 Widgets and The Carnap Book

Students interact with The Carnap Book through a range of different widgets provided by Carnap-GHCJS, which are embedded in web-pages served by Carnap-Server. At the moment, four main widgets are used in teaching.

The first of these is a "syntax-checker" widget, which The Carnap Book uses to familiarize beginning students with the five standard propositional connectives, and to help them acquire the concepts *main connective* and *parsing tree* that they will need for the rest of the course. The task is to enter the main connective of a highlighted formula into an input field. When the task is successfully completed, this will "break down" the formula into its parts. The parts are then highlighted and broken down by the student, until the full parsing tree of the formula is generated. The widget has two variants. The first variant (depicted in Figure 1) asks students to break down formulas with fully visible parentheses. The second variant familiarizes them with fixity conventions by asking them to break down formulas with some implicit parentheses.

Carnap's proof-checking functionality is exposed to students through a proof-checker widget. This widget consists of an HTML textarea where students enter proofs, decorated with a overlay for displaying line numbers and interactive feedback. In the spirit of flexibility—one of the design objectives described in Section 3.1—the proof-checker widget allows for three different styles of natural deduction: Hardegree (Figure 2), Fitch (Figure 3), and Montegue (Figure 4). Feedback on individual proof lines is available by means of a column, placed to the right of the proof, which displays a "+" for correctly derived lines, and which displays various icons for different types of error, with more detailed descriptions of problems visible upon mousing over the icon. Optionally, Carnap can generate a rendering of the proof closer

Figure 2: Hardegree-Style Proof (Currying a proposition)

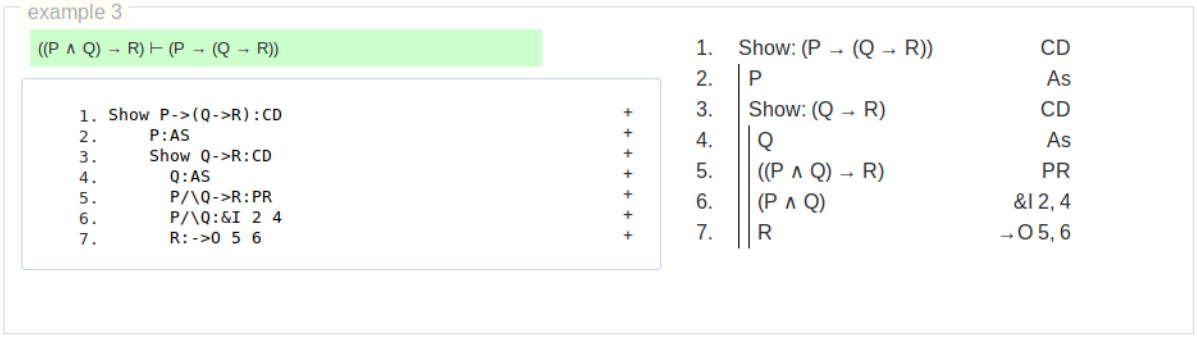
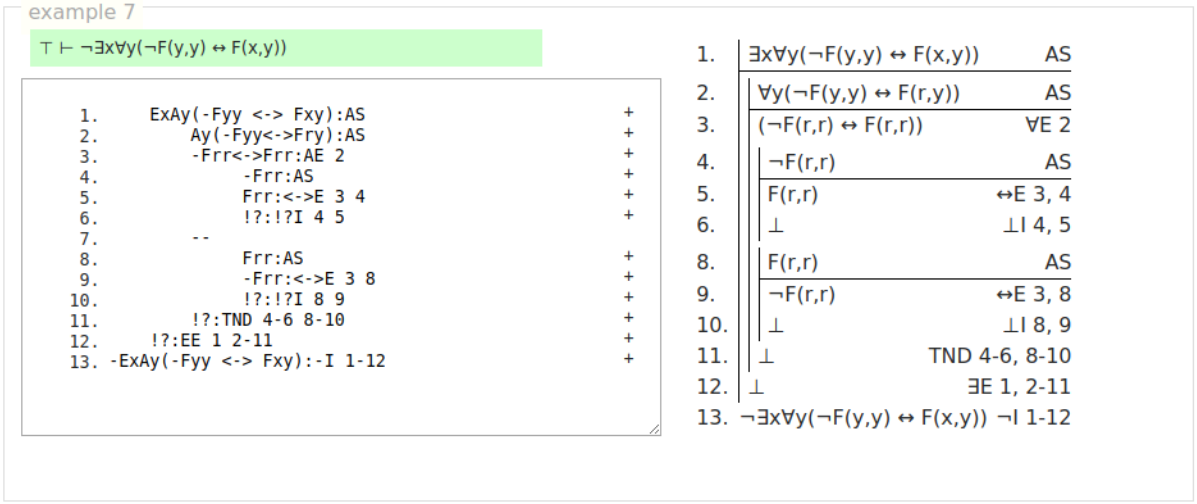


Figure 3: Fitch-Style Proof (Russell's Paradox)



to what might appear in a textbook, and display the rendering alongside the student proof (visible in Figures 2, 3, 4). Experimentally, Carnap can also overlay other feedback on the textarea, e.g. vertical guidelines to help with the proper indentation of subproofs.

To introduce students to semantic methods, The Carnap Book uses a simple truth table widget (Figure 5). Students are able to fill in the entire truth table to demonstrate the validity of a given argument, or they can present a single counterexample in order to demonstrate the invalidity of the argument. The semantic functions here—checking that a particular sentence is true relative to a certain assignment of truth values to its sentence letters—is derived from semantic functions at the level of the lexical modules that go into the construction of Carnap's propositional language. It would be possible to use the same method to construct exercises for checking counterexamples to arguments formulated in the languages of first-order or modal logic, although this functionality hasn't yet been required.

Finally, with a translation widget, The Carnap Book supplies two forms of translation exercise, asking students to translate English sentences into the languages of propositional and first-order logic. In the propositional case, Carnap can check solutions for correctness up to logical equivalence with the intended translation. In the first-order case, translations are currently required to be correct verbatim. Overcoming this limitation will require introducing a check for equivalence of sentences in the language of first-order

Figure 4: Montegue-Style Proof, in a system of second-order logic

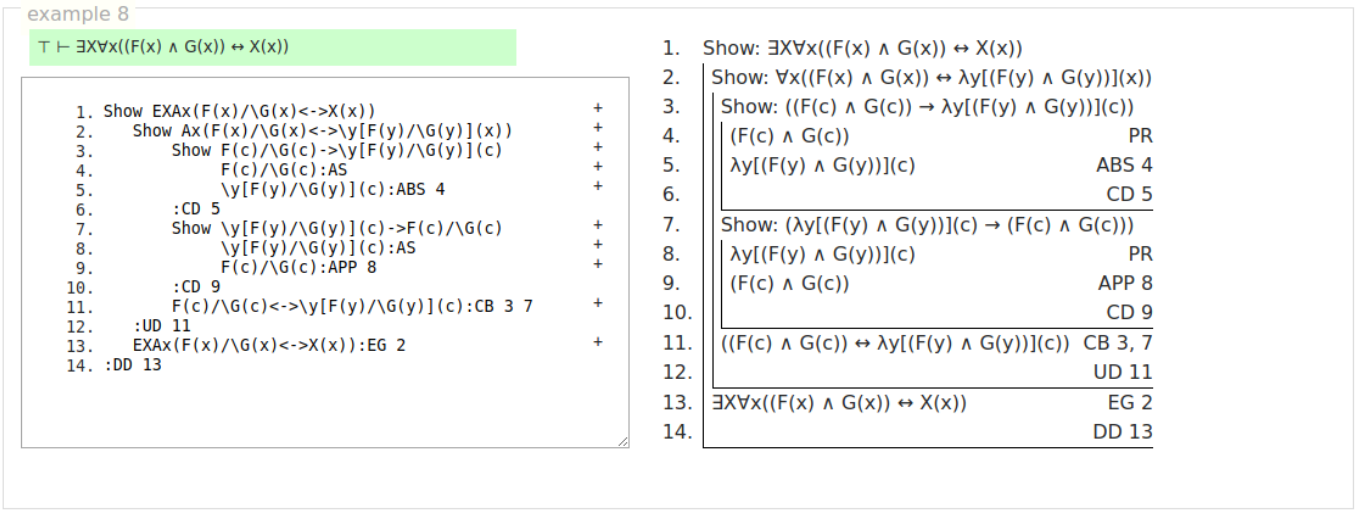
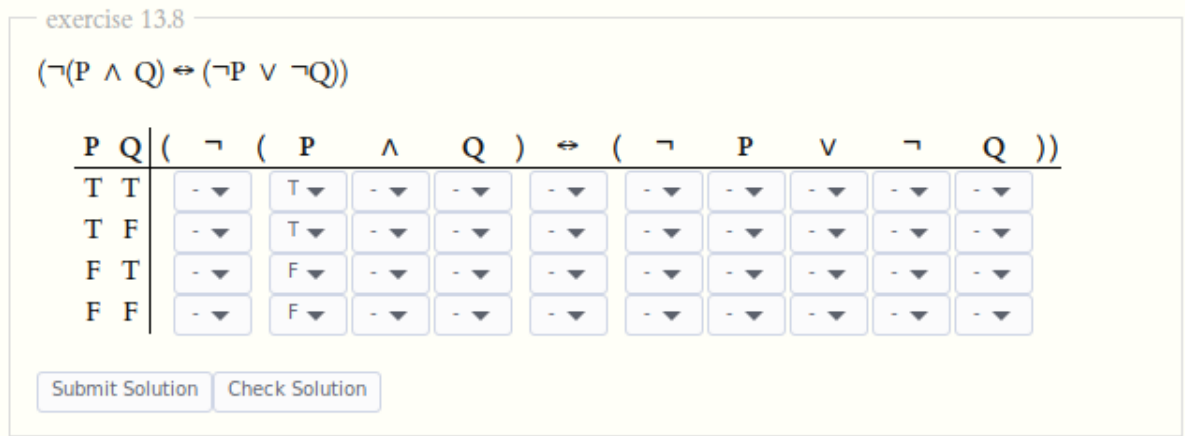


Figure 5: Truth-Table Exercise



logic, or some fragment thereof—likely through the introduction of an external theorem-prover. (See the next section for discussion of this possibility.)

5 Longer Term Prospects

In this section, we discuss the longer term prospects of the Carnap framework. In Subsection 5.1, we discuss new features we hope to add to the framework, new topics that we hope to incorporate into educational materials available on the Carnap website, and changes to the underlying technologies that we may need to address in the future. In Subsection 5.2, we discuss some potential future applications that we envision for the framework.

5.1 Projected Development

Carnap's successful use in the classroom and as a platform for the rapid development of a range of different proof-checking tools suggests that the basic architecture of the project is sound, and that the fundamental data types underlying Carnap's abstract syntax are sufficiently flexible to support the objectives of the project. Ultimately, depending on future development of the GHCJS compiler, it may make sense to move to the emerging WebAssembly standard, as it seems likely[17] that Haskell compiled to WebAssembly will eventually be efficiently executable in the browser. However, major revisions to the build process or to the fundamental parts of Carnap-Core seem unlikely in the relative short term. Most near future development will instead focus on extending Carnap's capacities, and improving the user-facing parts of the framework.

We are particularly concerned to:

(a) implement new unification algorithms;

(b) continue to extend Carnap's support for a wide range of languages and logics;

(c) develop novel web-based interfaces for proof-construction and semantic reasoning; and

(d) improve the Carnap-Server interface for instructors.

We'll discuss each of these objectives in turn.

First—We are concerned to implement new unification algorithms. While Huet's algorithm is extremely general and performs reasonably well for the type of problem that it is applied to in Carnap's proof checking,[18] there are other unification algorithms which are more efficient. It would be good to have them available in case efficiency becomes a problem in some future application. A likely target for a more efficient but still powerful unification algorithm would unification for higher-order patterns (see [7, p1041].) In the near future, we hope to implement this functionality generically for all of Carnap's languages, likely along the lines of the implementation described in [18].

Our second objective is to extend the range of languages and logics that Carnap supports. While Carnap's current capacities are sufficient for teaching logic at the elementary level, there are still various mathematically and philosophically interesting formalisms for which we still do not have proof systems.

[17] On the basis of e.g. the high-level goals of the WebAssembly team [1], and the existence of projects like the recently funded WebGHC [2]

[18] The class of unification problems that we use the algorithm for at the moment is restricted to higher-order pattern matching (not to be confused with unification of higher-order patterns). However, it is worth noting that even a very limited subset of this family of unification problems—namely second-order pattern matching—is NP complete. [3]

In order to continue to improve Carnap's usefulness as part of a logically rigorous curriculum in philosophy, we would like to add support for different constructive systems of logic, for philosophically interesting extensions to the standard languages of classical logic (e.g. definite descriptions, abstraction operators, truth predicates, multi-modal logics, and so on). And in order to make Carnap useful for students learning about reasoning in a more traditionally quantitative context—a standard "introduction to proof" class in a mathematics department, for example—it would also be desirable to add support for some elementary topics in mathematics, by adding, e.g., a language for graphs, for elementary algebra, and for elementary number theory or the theory of some other inductive structure.

Our third objective is to continue to develop novel web-based interfaces for proof construction and semantic reasoning. Several recent projects show the potential of the modern web for creating compelling interfaces and visualizations that enable students to bring visual insight to bear on logical problems, in something like the way they might bring it to bear on problems in calculus and geometry.[19] Most of these projects, however, have focused on the interface and left the underlying logical engine relatively simple. We hope to bring similar interfaces to Carnap. A first step will be to extend the JSON API mentioned in Section 2 to better expose the full functionality provided by Carnap-GHCJS. This will make it possible to build on existing open-source work of the kind mentioned above by adapting the interfaces they provide to plug into Carnap. It will also hopefully smooth the path for future front-end developers to easily make use of Carnap in similar explorations of visual reasoning, simplifying such projects by providing a ready-to-go client-side logic engine for JavaScript-based GUIs.

Our final objective is to improve the Carnap-Server interface for instructors. At the moment, since Carnap has been used primarily by its developers at Kansas State University, it isn't possible to remove and modify student accounts or to set due dates for problem sets from The Carnap Book through the Carnap-Server web interface. In order to make Carnap-Server more accessible to instructors outside of the university, it will be necessary to make these classroom management tasks easy to perform through the Carnap website. The Carnap website's content-creation functions could also be significantly improved. While at the moment, it's possible to create custom problem-sheets for one's class, instructors would benefit from some mechanism for creating other instructional materials as well: HTML slides with embedded logic widgets,[20] interactive textbooks and lecture notes similar to The Carnap Book, and automatically graded exams. Once better content-creation functionality is in place, it also seems desirable to allow instructors to optionally share their assignments publicly, or with selected others instructors.

5.2 Projected Applications

In coming decades, as computer-assisted proving moves into the mainstream in mathematics and computer science, students who wish to prove theorems or to produce formally verified software will need to be familiar with the idea of formal proof and with the capacities and limitations of proof-assistants. Researchers outside of mathematics and computer science departments may also eventually come to benefit from the higher standards of rigor, clarity, and precision that computer-assisted proving enforces. It is our hope that Carnap will eventually be able to serve as a bridge, both from standard introductory logic and discrete mathematics courses to more advanced computer-assisted logic and mathematics, and from the study of traditional problems in philosophy to rigorous research in philosophical logic. In this section, we conclude this paper by briefly describing those two projected applications.

[19] Two notable examples are The Incredible Proof Machine [5], and The Modal Logic Playground [12]

[20] Carnap's widgets can be embedded in HTML slides for presentations—see the slides associated with [15] for an example—but at the moment the slides must be created by hand.

Our experience with Carnap so far suggests that it is a good on-ramp for students (including the initially quantitatively anxious students found in philosophy departments) to learn about computer-assisted proof. Anecdotal evidence suggests less incremental approaches—for example teaching logic using the Coq IDE—while effective, presents significant challenges even for third or fourth year students in computer science [11, 19]. This is perhaps unsurprising. Applications like Coq are not designed with students in mind. Some of their most impressive features (all-encompassing scope, a tight relation to the production of formally verified software, a small trusted kernel) may even be counterproductive to pedagogical applications, where one wants to move in small steps and initially restrict degrees of freedom to those that are actually relevant to the concepts students are acquiring.

Hence, one possible application for the Carnap framework in the future will be the development of specialized "baby" proof assistants, designed to help novice students acquire basic concepts—formal inference, structural induction, Boolean operations, syntactic types—that will make the notation and operations of "full-grown" proof assistants intelligible, smoothing the learning curve in more advanced classes. Ideally, this familiarization could be achieved either by introducing these concepts (in particular, induction) within a second-semester logic course, or by adapting an introductory discrete mathematics course—where students often encounter induction and rigorous demonstration for the first time—both to incorporate an early unit on formal proof and to subsequently use Carnap as a tool for verifying proofs throughout the semester.

Developing an application for this purpose will require the completion of some parts of goal (b) from 5.1 above. In particular, it will be necessary to develop languages and logics that allow for reasoning by structural induction, and ideally that make use of notation resembling the notation of existing proof-assistants. It will also be helpful to create educational materials to designed to support instructors hoping to use Carnap to ease their students into computer-assisted proving. Much work remains to be done here, and much remains to be learned about the best approach to this particular pedagogical challenge. However, given the potential payoff for students of early exposure to computer-assisted proving, this application is worth exploring.

The second projected application mentioned above was the development of Carnap as a tool not just for students of logic but also for researchers in logic-adjacent areas of philosophy—those studying the foundations of mathematics, the semantics of natural language, truth predicates, reasoning with imprecise predicates, and so on. Just as researchers in mathematics can prevent errors and improve the precise communicability of their results by working in machine-checkable formalisms, philosophers whose work involves proof stand to benefit from the use of formal verification technologies. However, logical work in philosophy differs from research in pure mathematics in various ways. In particular, the work is more "comparative". In addition to proving theorems, philosophical logicians tend to be interested in finding formal models for phenomena of philosophical interest, which often requires looking at the behavior of a variety of different formalisms or developing entirely new formalisms. A framework like Carnap, with its emphasis on flexibility and on lowering the barrier to entry for users, would be well-suited to philosophical research.

Adapting Carnap for this application will again require the completion of some parts of goal (b) from Section 5.1. In this case, it will be important to provide support for formal languages studied in philosophical logic, and to improve Carnap's interface in a way that makes it suitable for sustained reasoning and not just solving classroom problems. In particular, it will be desirable to add formatting tools that will ease the construction and organization of longer proofs or of reasoning that adopts a theorem/lemma organization, and to integrate with one or more theorem-provers in order to shorten some of the more tedious and routine parts of proof-construction. Again, much work remains to be done, and much remains to be learned about the best approach to integrating computer-assisted proving into philo-

sophical research. But the potential benefits—in terms of improved rigour, improved communicability, and potential reorientation of some forms of research—make this application worth exploring.

References

[1] (2017): *WebAssembly: High Level Goals*. Available at `http://webassembly.org/docs/high-level-goals/`.

[2] (2017): *WebGHC*. Available at `https://webghc.github.io/`.

[3] L.D. Baxter (1977): *The Complexity of Unification*. Ph.D. thesis, University of Waterloo, New York.

[4] A. Bergmark (2010): *Fay*. Available at `https://github.com/faylang/fay/wiki`.

[5] J. Breitner (2016): *Visual Theorem Proving with the Incredible Proof Machine*. In Blanchette J. & Merz S., editors: *Interactive Theorem Proving—ITP 2016*, *Lecture Notes in Computer Science* 9807, Springer, pp. 123–139, doi:10.1007/978-3-319-43144-4_8.

[6] E. Czaplicki & S. Chong (2013): *Asynchronous Functional Reactive Programming for GUIs*. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation—PLDI*, ACM Press, pp. 411–422, doi:10.1145/2491956.2462161.

[7] G. Dowek (2001): *Higher-Order Unification and Matching*. In A.J Robinson & A. Voronkov, editors: *Handbook of Automated Reasoning*, Elsevier, pp. 1009–1062, doi:10.1016/B978-044450813-3/50018-7.

[8] A. Ekbald (2015): *A Distributed Haskell for the Modern Web*. Licentiate thesis, Chalmers University of Technology and Göteborg University.

[9] P. Freeman (2017): *PureScript By Example*. Leanpub.

[10] G. Hardegree (2016): *Introduction to Modal Logic*. Online Textbook. Available at `http://courses.umass.edu/phil511-gmh/text.htm`.

[11] M. Henz & J. Hobor (2011): *Teaching Experience: Logic and Formal Methods with Coq*. In J. Jouannaud & Z. Shao, editors: *Certified Programs and Proofs—CPP 2011*, *Lecture Notes in Computer Science* 7086, Springer, pp. 199–215, doi:10.1007/978-3-642-25379-9_16.

[12] R. Kirsling (2017): *The Modal Logic Playground*. Available at `https://rkirsling.github.io/modallogic/`.

[13] O. Kiselyov, C. C. Shan, D. P. Friedman & A. Sabry (2005): *Backtracking, interleaving, and terminating monad transformers*. *ACM SIGPLAN Notices* 40(9), pp. 192–203, doi:10.1145/1090189.1086390.

[14] G. Leach-Krouse (2017): *About Carnap*. Available at `http://carnap.io/about`.

[15] G. Leach-Krouse (2017): *Carnap @ ThEdu2017*. Presentation at ThEdu2017, Göteborg, Sweden. Available at `https://gleachkr.github.io/CADEslides/`.

[16] G. Leach-Krouse (2017): *The Carnap Book*. Online Textbook. Available at `http://carnap.io/book`.

[17] J. MacFarlane (2017): *Pandoc*. Available at `https://www.pandoc.org/`.

[18] T. Nipkow (1993): *Functional unification of higher-order patterns*. In: *Proceedings of Eighth Annual IEEE Symposium on Logic in Computer Science, 1993—LICS'93*, IEEE, pp. 64–74, doi:10.1109/LICS.1993.287599.

[19] B. Peirce (2009): *Lambda: The Ultimate TA*. In: *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming—ICFP'09*, ACM, pp. 121–122, doi:10.1145/1596550.1596552.

[20] W. Sieg (2007): *The AProS project: Strategic thinking & computational logic*. *Logic Journal of IGPL* 15(4), pp. 359–368, doi:10.1093/jigpal/jzm026.

[21] M. Snoyman (2012): *Developing web applications with Haskell and Yesod*. O'Reilly Media, Inc.

[22] L. Stegeman (2013): *Concurrent Haskell in the Browser with GHCJS*. Presentation at ZuriHac 2013, Zurich, Switzerland. Available at `https://wiki.haskell.org/ZuriHac2013#FP_Afternoon`.

[23] W. Swierstra (2008): *Data types à la carte*. *Journal of functional programming* 18(4), pp. 423–436, doi:10.1017/S0956796808006758.