Machine Involvement in Formal Reasoning
Simulated Contexts and an Interface Layer for Formal Verification

Andrei Lapets
April 25, 2010
Contents

I Machine Involvement in Formal Reasoning 5

1 Machine Involvement in Formal Reasoning 7
   1.1 Formal and Semi-formal Reasoning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
   1.2 Machine Involvement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
   1.3 The Verifier Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2 An Abstraction Layer for Machine Verification Systems 13
   2.1 Concrete Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
      2.1.1 Backward Compatibility: Simple and Familiar Syntax . . . . . . . . . . . . . . . . . . 13
      2.1.2 Forward Compatibility: No Explicit Annotations . . . . . . . . . . . . . . . . . . . . . 14
   2.2 Libraries for Syntactic Idioms and Logical Propositions . . . . . . . . . . . . . . . . . . . . 15
      2.2.1 A Rich Ontology of Concepts, Definitions, and Propositions . . . . . . . . . . . . . 16
      2.2.2 Accessing and Employing the Ontology in Real Time . . . . . . . . . . . . . . . . . . 16
   2.3 Interaction and Real-time Assistance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
      2.3.1 Automatic Hints for Supported Syntactic Idioms . . . . . . . . . . . . . . . . . . . . . 17
      2.3.2 Flexible and Lightweight Validation Capabilities . . . . . . . . . . . . . . . . . . . . . 17
   2.4 Inference, Search, and Automation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
   2.5 Implementing an Interface Layer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

3 An Automated Assistant for Formal Reasoning 19
   3.1 Concrete Syntax and Abstract Representation . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
      3.1.1 Concrete Syntax and Parsing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
      3.1.2 Abstract Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
   3.2 The Simulated Context (Library) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
      3.2.1 Static Context . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
   3.3 Interaction and Real-time Assistance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
      3.3.1 Argument Submission and Validation Feedback . . . . . . . . . . . . . . . . . . . . . . 25
      3.3.2 Real-time Syntactic Idiom Lookup and Hints . . . . . . . . . . . . . . . . . . . . . . . 25
   3.4 Inference, Search, and Automation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
      3.4.1 Inference Rule Template for Expressions . . . . . . . . . . . . . . . . . . . . . . . . . . 27
      3.4.2 Inference Rules for Statements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
      3.4.3 Dynamic Context . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

II Use Cases, Applications, and Integrations 37

4 Evaluation of Machine Verifiers 39
5 Classroom Instruction of Formal Reasoning 41
  5.1 Classroom Instruction of Reasoning about Pure Functional Programs . . . . . . . . . . . . . . 41
    5.1.1 Related Work in Instruction of Reasoning about Functional Programming . . . . . . . . 41
    5.1.2 Examples of Verifiable Formal Proofs . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
  5.2 Classroom Instruction of Linear Algebra . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
  5.3 Evaluation of the Capabilities of the System . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
    5.3.1 Logical Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
    5.3.2 Properties of Equality . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
    5.3.3 Algebraic Manipulations with Arithmetic Operators . . . . . . . . . . . . . . . . . . . . 52
    5.3.4 Improvements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
  5.4 Pedagogical and Administrative Consequences . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
    5.4.1 Instruction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
    5.4.2 Usability from the Perspective of Students . . . . . . . . . . . . . . . . . . . . . . . . . . 54
    5.4.3 Grading . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

6 Safe Compositional Network Sketches 57
  6.1 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
    6.1.1 Usability and Application of Formal Reasoning Systems . . . . . . . . . . . . . . . . . . 57
    6.1.2 Formalisms for Modelling Constrained-flow Networks . . . . . . . . . . . . . . . . . . . 58
  6.2 Soundness of the NetSketch Formalism . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
    6.2.1 Modules: Untyped and Typed . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
    6.2.2 Semantics of Network Typings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
    6.2.3 Typed Network Sketches . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
    6.2.4 Soundness . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
  6.3 NetSketch and the Integrated Automated Assistant . . . . . . . . . . . . . . . . . . . . . . . . . 72
    6.3.1 Module Design and Analysis with an Automated Assistant . . . . . . . . . . . . . . . . . 72
    6.3.2 A NetSketch Use Case and the Automated Assistant . . . . . . . . . . . . . . . . . . . . 73

7 A Formal Framework for Service-Level Agreements 77
  7.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
  7.2 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
Part I

Machine Involvement in Formal Reasoning
Chapter 1

Machine Involvement in Formal Reasoning

In a variety of human activities with a specific goal, machine assistance can be advantageous along at least two dimensions. It can improve the efficiency of the human (the amount of work the human can accomplish per unit time), and the accuracy of the human (the quality of the work produced when measured using some defined metric). If we consider the activity that might be described as computation, wherein a human must perform a mathematical calculation according to a specified set of rules, machine involvement demonstrably improves both the human’s capacity and accuracy. However, this is not yet the case for the broader category of human activity that can be described as “formal reasoning”.

Machine verification of formal (or even semi-formal) arguments can only increase confidence in the accuracy and correctness of those arguments. Evidence of this lies in the benefits of adopting a formal representation amenable to machine verification in research efforts involving mathematical rigor, as well as in mathematical instruction. These benefits include reusability, automatic evaluation of examples, and the opportunity to employ domain-specific machine verification. Machine verification can offer anything from detection of basic errors, such as the presence of unbound variables or type mismatches, to full confidence in an argument because it is consistently constructed using the fundamental principles of a particular mathematical logic. There exist a variety of such machine verification systems, and some of these have been surveyed and compared along a variety of dimensions [Wie03].

Unfortunately, the costs of employing machine verification still outweigh the benefits in a variety of formal reasoning activities. While it is by restricting a user to correct arguments that a machine verifier serves its purpose, such restrictions can inhibit even an expert user’s productivity when they are reflected in the machine verifier’s interface. To date, broad accessibility and quality interface design have not been a priority in the design of machine verification systems. On the contrary, a researcher hoping to enjoy the benefits of formal verification is presented with a variety of obstacles, both superficial and fundamental. Thus, while employing a machine verification system can improve the accuracy of formal reasoning activities, doing so also impairs the capacity and productivity of the user.

As a consequence of this tradeoff, many instructors and researchers today still choose to ignore such systems. In the literature in most domains of computer science and mathematics there are only isolated attempts to include machine-verified proofs of novel research results, and in only a few mathematics and computer science courses are such systems employed for presenting material or authoring solutions to assignments.

Moderating the conflicting constraints inherent to the contemporary design of machine verification systems is essential. Any improvement over these circumstances not only brings the benefits of such tools to a broader audience, but advances the state of the art in machine verification systems. Furthermore, in any market in which there are many competing machine verification systems that are already being utilized, a system with a superior user interface will have a competitive advantage. The ideal circumstances towards which further work in the design of machine verification systems should strive is one in which machine involvement improves
1.1 Formal and Semi-formal Reasoning

It can be difficult to delineate formal reasoning as an activity that is strictly distinct from other kinds of human reasoning. Even an argument that most would agree is formal and rigorous often does not specify the vast context of knowledge and experiences that gives the argument any sort of meaning, let alone allows it to be judged as “correct”.

In this work, we call formal reasoning any human reasoning activity that involves mental manipulation of concepts according to consistent or nearly consistent rules. These concepts are assumed to be those that are traditionally employed within the pure and applied sciences, and in particular those that arise frequently in engineering, computer science, and mathematics. Examples of this kind of formal reasoning include proof sketching, composition of rigorous mathematical proofs, modelling of physical systems, and programming of devices (abstract or concrete) that may or may not have a fully-specified semantics.

A notion of formal reasoning outlined in this way is quite broad, so it is worth emphasizing that the tools and techniques described in this work address only a portion of these formal reasoning activities. The focus is directed on formal reasoning activities that involve describing and manipulating ubiquitous and intuitive formal concepts like numbers, sets, sequences, vectors, sums, boundaries, and graphs.

1.2 Machine Involvement

In this work, we study and address issues related to machine involvement in human formal reasoning activities. In order to mitigate the fluid nature of the problem being addressed, we describe in detail the scenario that is under consideration: a human engages in formal reasoning with the assistance of a machine. We introduce several terms labelling the elements in this scenario. This terminology will be used consistently throughout this work and is illustrated in Figures 1.1 and 1.2.

A user is a human involved in some formal reasoning (or reasoning for short) activity within the context of some domain. Examples of domains include set theory, C++ programs, circuits, constrained-flow networks, and so on. A domain is typically associated with a domain model, which is a collection of facts and definitions that describe the domain and can be used to determine the accuracy of arguments about the domain.

When the user chooses to employ a machine verification system (or machine verifier, or simply verifier), she converts her reasoning into a formal argument (or argument) in the input format of the verifier and
1.2. MACHINE INVOLVEMENT

submits it for processing. After processing the argument and determining how consistent it is with a domain model (a process we call validation), the verifier returns feedback to the user that indicates the accuracy or correctness of the argument (such as by highlighting errors within the input text representing an argument). This submission/feedback loop may occur only once, or may repeat itself many times throughout the formal reasoning process.

Formal Arguments

A formal argument typically consists of one or more plain text documents. It may contain both the argument and descriptions of entities involved in the argument (e.g. programs). The argument may or may not be semi-formal, which means that either the domain model or the argument (or both) are less precise than, for example, automatically verifiable symbolic proofs in first-order logic.

As an example, we consider a plain text (interpreted by \LaTeX) formal argument that $\sqrt{2}$ is irrational, seen in Figure 1.3. The argument is made by assuming the negation of the hypothesis, and concluding with a contradiction.

Assert for any $n, m \in \mathbb{Z}$,
if $m \neq 0$,
  $n$ and $m$ are relatively prime, and
  $\sqrt{2} = n/m$ then
  
  $m \cdot \sqrt{2} = m \cdot (n/m)$,
  $m \cdot \sqrt{2} = n$,
  $(m \cdot \sqrt{2})^2 = n^2$,
  $m^2 \cdot \sqrt{2}^2 = n^2$,
  $m^2 \cdot 2 = n^2$,
  $n^2 = m^2 \cdot 2$,
  $n^2 = 2 \cdot m^2$,
  $n^2$ is even,
  $n$ is even, and
  $n^2 = (2 \cdot (n/2))^2$,
  $n^2 = 2^2 \cdot ((n/2)^2)$,
  $n^2 = 4 \cdot ((n/2)^2)$.
  $2 \cdot m^2 = 4 \cdot (n/2)^2$,
  $m^2 = 2 \cdot (n/2)^2$,
  $m^2$ is even,
  $m$ is even,
  $GCF(m, n) \geq 2$,
  $GCF(m, n) = 1$, and
  there is a contradiction.

Figure 1.3: An example of a formal argument.

One important observation at which we hinted earlier is that an argument traditionally considered to be formal by, for example, the community of practicing mathematicians typically does not include many of the definitions that give the argument meaning, and almost never contains a precise specification of the domain model that should be used to judge its correctness.
1.3 The Verifier Interface

Implicit in the scenario we described is the presence of a verifier *interface*. The interface is the abstraction layer that lies above the validation procedures the machine verifier implements for the domain model being considered. What would make an ideal interface? How should the quality of an interface be measured?

The interface determines the representation format into which the user must transform her argument. Thus, it must strive to facilitate this transformation process by meeting the user’s expectations. It must provide an abstraction of the system that corresponds to the user’s intuition and experience. Furthermore, an interface must inform the user of the system’s capabilities (such as libraries of results the user might need to employ), limitations (such as supported syntactic constructs and idioms), and actual results (such as error messages indicating mistakes in formal arguments). It must make clear what it expects of the user. Ideally, it should do all this in a way that is immediate, interactive, and context-aware.

The User’s Mental Domain Model

Implicit in our scenario is the presence of a mental domain model of the user. Such a model may necessarily rely on intuition and may be error-prone. However, it is also more broad and may include relationships (metaphorical, structural, and so on) between the domain under consideration and other domains. We refer by the term *user context* (or context) to a user’s mental domain model, experiences, and thought process when authoring a particular argument. The context is the sphere of awareness a working human user maintains of the relevant constructs, arguments, experiences, and background materials necessary to accomplish the formal reasoning task at hand. Our own observations, and those collected from related work [Wie03], suggest two relevant characteristics of a context.

The first is the context’s size and breadth: it encompasses a wide array of experiences, potentially inconsistent or not directly related to one another. It includes various kinds of syntactic idioms, logical definitions and propositions. Of particular interest is the vast repository of analogies and metaphors a user’s context usually contains (as enumerated at length and in great detail in texts such as that by Lakoff and Núñez [LNn01]). The context also includes many aspects of the argument at hand, such as the objects and concepts it involves and any logical errors that it might contain. The second characteristic is its flexible capacity to support easy exploration, querying, and short-term adjustments (using not just references or indexes but also structure). Relevant constructs found in the vast context are introduced based on the contents of an argument, and information about the argument at hand can also be explored and queried.

Inevitably, the quality of an interface layer will be influenced by its ability to comprehend the user’s argument by simulating the user’s mental domain model. As observed by others working in this area [ACP01], any development that improves machine comprehension of arguments by analogy would be beyond the current state of the art. It is also worth noting that in explicitly recognizing the possibility that a context can contain distinct, inconsistent collections of experiences (such as different logical systems), we are in part inspired by the notion of a cognitive context found in literature on formal ontologies [PB08].

Implementing a Separate Interface Layer

Recognition of the presence of a user context leads us to consider introducing a distinct interface layer into our scenario, and we call this the *interface* layer. We distinguish this layer from *validation layer*, which encompasses the machine verifier. This distinction can be justified by the fact that each layer solves a separate problem. It is the exclusive responsibility of the validation layer to measure an argument’s accuracy and correctness with respect to a precise domain model, whereas it is the exclusive responsibility of the interface layer to simulate the user’s context for that domain.

As we discuss in more detail and with references to previous work in Chapter 2, there are several common components of machine verifiers that could be considered essential interface layer components, rather than validation layer components. These include:

- the concrete syntax and parser for arguments;
- a library that contains:
1.3. THE VERIFIER INTERFACE

- existing definitions and results relevant to the domain,
- existing definitions and results from other domains familiar to human users;

- the interactive protocol, which can include:
  - the option to select the kind of validation to employ (allowing the user to make a trade-off between certainty and argument complexity/response time),
  - automatic, real-time hints for supported syntactic constructs, idioms, and propositions (as found in the library),
  - indexed, interactively searchable feedback explaining what parts of an argument are valid, what errors are present, and what the system knows about the argument;

- inference, search, and automation procedures that correspond to logical systems but actively incorporate the library in the validation process (such as by computing congruence closures of supported relations).

Even separated from the problem of verification with respect to a specific domain model, many of these features can be improved and extended indefinitely. There are a variety of advantages to developing these components within the framework of a distinct interface layer. In particular, these components can be used in scenarios with different numbers of underlying verifiers (and corresponding domain models). In the absence of a precise domain model, they can provide “lightweight” corroboration, and the capabilities of such a system constitute one potential characterization for “lightweight” systems. For the designer of an individual machine verifier, a separate interface layer makes easier the task of satisfying the de Bruijn criterion [Wie03], which states that a machine verifier should consist of a very small kernel, without compromising the quality of the user interface. If multiple verifiers and domain models are involved, the interface layer can target all of these or even choose among them based on the circumstances, while the user is still required to familiarize herself with only a single interface. This also provides a standard “intermediate representation” that designers of verifiers can target.

As we discuss in Section 2.4 of Chapter 2, a separate interface layer also provides a disciplined yet anthropocentric way for machine verifiers to satisfy the Poincaré principle, which states that a system should be able to prove the correctness of calculations automatically [BBB+97].
Chapter 2

An Abstraction Layer for Machine Verification Systems

We identified four major components that serve the purpose of an interface in existing machine verification systems. In this chapter, we describe these components in greater detail. We comment on how each component can contribute to the overall usability of the verifier, and propose design principles that can further extend this contribution. In many cases our proposals echo or further develop observations that motivated work directed towards improving the usability of formal reasoning and verification systems, such as the Tutch proof checker [ACP01], the Scunak mathematical assistant system [Bro06], and the ForTheL language and SAD proof assistant [VLPA08].

2.1 Concrete Syntax

In designing a successful interface and syntax for a machine verifier, one can consider the balance between the past and future, and the amount of flexibility available to accommodate both beginners and expert users. Both of these considerations are directed towards making more convenient the process of converting an argument into a form that the machine verifier can process.

One author, commenting on designing an interface for representing proofs, opines that “we seem to be stuck at the assembly language level” [Wen99]. A potential strategy in the design of such representation is the elimination as many aspects of the interface as possible that a user would consider unnecessary if she were communicating the argument to another human.

Improvements to both the flexibility and manageability of the syntax are of little use unless other components in the interface layer can provide the necessary support. Earlier efforts make claims that certain verification systems support “human-readable” representations of formal arguments [Wen02, Rud92, ACP01]. However, because of a lack of consistent conventions in some cases (e.g. in notation for representing graphs or vector concatenation) and the very large size of the libraries of those systems, the syntax of such systems can still be obscure even to expert users who may with to employ them in their own work. The issue of teaching users in real time about system capabilities (including libraries of results users might need to employ) is usually not addressed sufficiently well. A repository or library of syntactic idioms and semantic propositions, a means of informing the user of these in a context-sensitive manner, and an ability to ascertain the implicit reasoning of the user semantically are essential. These are discussed in Sections 2.3, 2.4 and 2.2.

2.1.1 Backward Compatibility: Simple and Familiar Syntax

Expert communities have established conventions for notation, and these should be exploited to the greatest extent possible. This facilitates wide adoption and makes it possible to use a system as a medium of
communication within such communities. Our observations are shared by the designers of Scunak [Bro06], who refer to the need for “naturality” in a system’s concrete representation.

One way to satisfy this requirement is to adopt subsets of concrete syntaxes from well-established authoring and content management tools such as LISP and MediaWiki. The designers of systems such as Scunak [Bro06] have chosen to do exactly this, and in recent work [DSW08] the ΩMEGA proof verifier [SBF02] has been integrated with the scientific text editor TexMACS. To better serve users in engineering, mathematics, and the applied sciences, the Fortress programming language [ACH08] incorporates common mathematical symbols and syntactic constructs into its syntax, and the designers are putting effort into assembling a flexible parser that simplifies user-directed expansion of the language syntax [Ryu09].

Another complementary approach is to adopt a simple syntax such as that of the Tutch system [ACP01], which has only four basic constructs, and including constructs that allow the user to easily specify which portions of an input should be verified and which should be ignored. While the designers of the ForTheL language and SAD proof assistant [VLPA08] chose a syntax resembling natural language text, their syntax uses a few unfamiliar conventions and deviates from the notation employed for mathematical formulas in LISP. Furthermore, the high-level constructs for managing proofs in ForTheL are reminiscent of proof scripts for interactive theorem provers and present similar obstacles to new users. Enforcing high-level constructs like proofs and theorems inhibits the user from employing a lightweight approach and presents obstacles to users who might follow different proof construction conventions when authoring their own formal arguments.

More widely, there exist other efforts to create interfaces and systems for practical formalization of mathematics. The MathLang project [KW08] is an extensive, long-term effort that aims to make natural language an input method for mathematical arguments and proofs. Natural language arguments are converted into a formal grammar (without committing to any particular semantics), and the participants are currently working on ways to convert this representation into a fully formalized logic. The MathLang project is focused primarily on mathematical texts, and representative texts from different areas are used as a guide in determining the direction of future research in the project.

2.1.2 Forward Compatibility: No Explicit Annotations

In many existing systems, forward compatibility is impaired because the syntax or interface explicitly support certain proof strategies or verification algorithms. For example, one popular approach to the design of interfaces for proof authoring and verification systems is embodied in interactive theorem provers, such as those designed for Coq [PV97] and Isabelle/Isar [WP06]. However, as has been observed before [ACP01], this is actually an inconveniently rigid framework. The user is required to learn how to direct the interactive system, cannot easily “jump around” while constructing a proof, and cannot resort to a lightweight approach under which only some parts of the proof are formal and correct. Furthermore, this discourages designers of verification and content management systems from adopting the interface and discourages users from employing the interface as a communication medium.

The logical and mathematical concrete syntax in a system should only allow the user to represent formal (or mostly formal) expressions, and no syntactic constructions should be provided for “helping” a particular verification strategy (e.g. by specifying which axiom or theorem is being applied at a particular point in a formal argument). Because special annotations that might not be necessary in the future are left out of the formal representation, future systems need not deal with parsing such special annotations. A gradually evolving variety of proof search techniques, tactics, and heuristics can then be added to verification tools that operate on this representation without affecting the representation. Systems like ΩMEGA [SBF02] arguably demonstrate this idea by attempting to do proof reconstruction on individual proof steps of a formal argument after a user has made changes to it.

This more natural representation for formal arguments also lends itself well to another crucial characteristic of our notion of a natural context: the ability to retrieve definitions and propositions by structure. Most of the facts or rules being applied in a typical proof found in the literature are not explicitly mentioned. This is a principle to which the designers of the Scunak system [Bro06] refer as “retrievability” ... by content rather than by name.” Likewise, the designers of the Tutch system posit that an “explicit reference to an inference
2.2 Libraries for Syntactic Idioms and Logical Propositions

Within the context of an evaluation and taxonomy of a collection of formal verification systems, it has been observed that "[in] practice for serious formalization of mathematics a good library is more important than a user friendly system" [Wie03]. It is certainly true that a library is essential if a machine verification is to formalize any significant fraction of mathematics, human users must spend considerably effort contributing to this library. This is especially important when one considers that in certain areas of research, new concepts, or merely new relationships between existing concepts, are introduced regularly. A friendly interface facilitates the essential process of library expansion.

But the relationship between usability and library size operates in both directions. Library size is an important factor in the usability of a system because it simulates the static context. In fact, a library can supersede and encompass the other components. A library can contain a database of syntactical constructs (including idioms), a database of collections of inference rules and the logical systems associated with them, a database of concept definitions and related propositions, and a database of algorithms and procedures for automating the verification process.

<table>
<thead>
<tr>
<th>Introduce $P, m$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assume $P$ is a finite set, $P$ is non-empty, and $P \subset \mathbb{N}$.</td>
</tr>
<tr>
<td>Assume for all $n \in \mathbb{N}$, if $n$ is prime then $n \in P$.</td>
</tr>
<tr>
<td>Assume $m = P_0 \cdot \ldots \cdot P_{</td>
</tr>
<tr>
<td>Assert $m \in \mathbb{N}$.</td>
</tr>
<tr>
<td>Assert for any $p \in \mathbb{N}$,</td>
</tr>
<tr>
<td>if $p$ is a prime factor of $m + 1$ then</td>
</tr>
<tr>
<td>$p$ is not a factor of $m$,</td>
</tr>
<tr>
<td>$p$ is prime,</td>
</tr>
<tr>
<td>$p \in P$,</td>
</tr>
<tr>
<td>$p$ is a factor of $m$,</td>
</tr>
<tr>
<td>there is a contradiction.</td>
</tr>
</tbody>
</table>

Figure 2.1: An example of a proof of the infinitude of primes.

If we suppose that the primary purpose of a library is to model the static context of a human user, it
CHAPTER 2. AN ABSTRACTION LAYER FOR MACHINE VERIFICATION SYSTEMS

is essential that the library be large. Even if one considers a small collection of mathematical concepts, a practicing mathematician is familiar with a large number and a great variety of propositions that describe relationships between the concepts in this collection. To illustrate this, Figure 2.1 presents a very short proof of the infinitude of primes. This short proof contains explicit references to finite sets, natural numbers, prime numbers, products, and factors. It also contains many implicit references to the properties of these concepts, and to the relationships between them. Any system that aims to support the kind of formal reasoning activity used in the construction of such a proof must contain an extensive library containing all these concepts. A large library, in turn, necessitates an interface for library management and expansion.

Organizing into a hierarchical library of definitions and propositions the multitude of relevant concepts and relationships found within a typical user context is difficult. What principles can be used to decide how these propositions should be grouped? Would such a hierarchy be of any value to a user? It is arguably more important to assemble a large collection of pertinent propositions and index them for easy retrieval based on the expectations of the user. If such a library is large and broad in scope, is indexed by structure and context (rather than only by name or hierarchical classification), and is accessible to users through a flexible interface, it may be more valuable than an organized library of a more limited scope. There exists a contemporary approach to knowledge assembly that satisfies these requirements.

2.2.1 A Rich Ontology of Concepts, Definitions, and Propositions

An approach to knowledge assembly worth considering can be found in work in the subdiscipline of artificial intelligence that deals with the assembly and application of ontologies. Some work in this area studies possible applications of semi-formal and formal knowledge databases in the development of software that can assist humans in everyday tasks. One example is the Cyc Project, an effort to assemble “[a] system that rests on a large, general-purpose knowledge base” that “can potentially manage tasks that require world knowledge, or ‘common sense’ - the knowledge that every person assumes his neighbors also possess” [PML+06]. This approach can be adopted in the design of an interface layer for machine verification systems. Instead of working with a database of commonsense propositions about real-world concepts, it is possible to assemble and provide a method of utilization for a database of propositions involving common mathematical concepts.

A large knowledge repository is of limited use without an effective interface for management and exploration that can be used at least by expert designers or administrators. Pertinent to this is work being done as part of the Open Mind Common Sense project, some of which [Chk05, CG05, SBL04] aims to collect a large database of commonsense propositions [LS04a, LS04b] about the real world from users. Users submit propositions using a simple online submission form, and this database is exposed through an interface that processes user queries. Even when it is restricted to experts and system designers, is an effective tool for accumulating over time and from many experts a large collection of propositions about any domain, including the one we are considering.

Work also exists in assembling indexed databases of computational methods (algorithms) that can be utilized through an interface. One example is the online search tool Hoogle, designed to allow users to explore the Haskell libraries [Mit08]. Another example with the same underlying motivation is the WolframAlpha “answer engine” (as opposed to search engine) [Wol], which allows users to interact through a natural language interface with a collection of databases that contain both facts and algorithms. However, natural language is not necessarily an effective interface for a database of propositions or computational methods if it does not provide a means for specifying the context of a query. In particular, even a human capable of communicating in a natural language cannot answer queries when the queries are posed out of context.

2.2.2 Accessing and Employing the Ontology in Real Time

An approach to library design that involves the construction of a large library of definitions and propositions can be viewed as a variation on the notion of a type system in which the usual priorities have been rearranged. Traditional type systems are designed to have terse descriptions and definitions, and this allows proofs about the properties of such systems to be concise. By contrast, we prioritize the assembly of a large collection of inference rules that can contribute to modelling a user context, and any assurances about the system’s
soundness and consistency must be provided by some secondary mechanism that can translate an object composed of these rules into a representation for one or more appropriate verifiers.

This relationship suggests one approach to exposing a large library to the user that involves developing a more general notion of type inference. In particular, this involves developing a type inference algorithm that can take into account the entire library while still running with relative efficiency and terminating. This algorithm might also be not only syntax-directed, but context-directed. In fact, the type inference algorithm for any language with variables is already context-directed.

A related and particularly promising technique that can be leveraged in providing a user easy access to a large library of facts about formal concepts is to use algorithms that compute congruence closures of relations [BT00]. A congruence closure can effectively implement a context-directed inference algorithm for finite collections of concepts or expressions introduced by the user, and work exists addressing how efficient algorithms for computing congruence closures can be constructed [NO80, NO07]. The integration of congruence closure algorithms with a large library of common concepts and definitions as described in Section 2.2.1 may lead to substantive improvements in the usability of formal reasoning systems.

Related work in the construction of SMT solvers [NOT06], and especially general-purpose, multi-domain SMT solvers [BPT07], [FHT+07] is also of some relevance. Such systems integrate multiple algorithms and techniques within a single tool. This allows them to provide some verification and computation capabilities for formulas that involve predicates and operators from undecidable theories. These systems are often directed towards applications in modelling and engineering real-world systems, but it is likely that a similar multi-pronged approach is necessary for providing practical support for users engaged in formal reasoning within the context of novel research or classroom instruction.

### 2.3 Interaction and Real-time Assistance

One popular approach to the design of interfaces for proof authoring and verification systems is embodied in interactive theorem provers, such as those designed for Coq [PV97] and Isabelle/Isar [WP06]. However, as has been observed before [ACP01], this is actually an inconveniently rigid framework. The user is required to learn how to direct the interactive system, cannot easily “jump around” while constructing a proof, and cannot resort to a lightweight approach under which only some parts of the proof are verified as formal and correct.

Several existing systems designed with usability in mind have eschewed this approach [Bro06, ACP01], though it is still employed by systems such as the EPGY system [MRS01], in which valid algebraic manipulations are strictly enforced and can only be done by way of an interactive interface.

#### 2.3.1 Automatic Hints for Supported Syntactic Idioms

Syntax is a means of communication, and a simple and natural formal syntax is useful because it provides a means for encoding formal arguments that can be learned quickly. However, this simple syntax is used to represent a large library of operators, predicates, and even syntactic idioms. It is necessary to both store all these conventions in some sort of database, and to expose them to a user without requiring that they spend time reading documentation. While an indexed database of syntactic idioms (and, more generally, logical propositions [CG07]) is a natural starting point, real-time keyword-based lookup techniques for programming environments [HWM09, LM07] provide a means for further improving the usability of a system. These same techniques can also be used to provide indexed, searchable error (and validation) reports that a user can explore interactively. This approach addresses the difficulty of providing relevant and informative error messages to users by treating the error reporting process as an interactive exchange between the system and user, rather than as a one-time delivery of feedback.

#### 2.3.2 Flexible and Lightweight Validation Capabilities

It is possible that a researcher wishes to employ only certain benefits of a formal representation, such as syntax error detection, unbound variable detection, or type checking, and does not wish to invest any extra
effort to enjoy additional benefits. Sometimes a user does not have confidence in arguments based on the fundamental principles of a logic because she is not certain the logic is an adequate model for her purposes. This is a common concern in research on the safety of cryptographic protocols [AR00]. Recent work reveals that others are beginning to acknowledge this concern, offering a notion of correctness that deals with syntax, arity, and bound variables called "ontological correctness" [VLPA08], or demonstrating that a type system can provide some limited verification capabilities [Bro06]. This progress presents the opportunity to go further by providing a wider spectrum of lightweight verification methods (such as those that guarantee superficial consistency by verifying that symbolic manipulations were correct, regardless of their semantics).

Many established verification tools, such as in Coq, Isabelle/Isar, Mizar, and PVS, do not allow such selections to be made easily. It is certainly possible within these systems to introduce user-defined axioms, but the interfaces do not treat this as a primary and essential activity. In particular, they do not embody our notion of a vast, all-encompassing natural context: distinctions between systems are rigid and the interface faithfully reflects this. On the other end, systems like the Alloy modelling language [Jac02] do encourage a more organic approach, but once the commitment has been made to using only the benefits provided by the chosen system, there is no possibility of introducing a more rigorous approach in the future using a compatible representation.

2.4 Inference, Search, and Automation

A separate interface layer provides a disciplined yet anthropocentric way for machine verifiers to satisfy the Poincaré principle, which states that a system should be able to prove the correctness of calculations automatically [BBB+97]. In fact, it is precisely the task of an interface layer to satisfy this principle (while the verifier design must be more concerned with satisfying the aforementioned de Bruijn criterion). The drawback of automation strategies in many modern verifiers is that they are based on ad hoc algorithms and "tactics" that are implemented because it is possible to implement them, without extensive consideration of whether to correspond to the context (mental domain model) of the user. As a result, the strategy of automation in general is open to a variety of criticisms. Ad hoc strategies might be difficult to characterize for a user because it is difficult to determine the circumstances in which they will work (and they cannot work in all circumstances because the capability they automate is undecidable in general). They might become "out-of-date" when newer algorithms are developed, making older proofs incompatible.

2.5 Implementing an Interface Layer

Existing efforts found in related work, combined with our own observations, provide a blueprint for both implementing an interface layer and a unifying standard – the user context – by which to measure its effectiveness. We follow this blueprint in implementing each component according to the principles enumerated in this chapter. We combine these components into an integrated implementation of an interface layer for machine verifiers. This implementation serves as a prototype for evaluating the effectiveness of the outlined techniques and criteria. In the next chapter we describe this process and the product of its execution.
Chapter 3

An Automated Assistant for Formal Reasoning

We describe the definition and implementation of the AARTIFACT system, an interface layer for machine verification systems. We organize our description according to the common interface layer components enumerated and discussed at length in Chapter 2. In particular, we describe AARTIFACT in great detail by presenting the definitions and some implementation details of its concrete and abstract syntaxes, its user interface, its underlying logical inference procedures, and its simulated context.

Figure 3.1 illustrates the overall organization of the various components of the system, and how they behave in practice. An expert-managed database contains a large library of syntactic constructs and propositions. This database is compiled into a client-side JavaScript application for syntax lookup, and a server-side executable that can perform formal verification. This ensures that only the server must be trusted to perform verification correctly, while the computational burden of providing syntax lookup is carried by the client machine. The server sends the JavaScript application to the client when the web interface illustrated in Figure 3.6 first loads. Users then author arguments on their own browser with the help of the JavaScript application, and have the option at any time of submitting their arguments with the click of a button to the server for validation. When submitted, arguments are processed and feedback is immediately returned to the user for examination.

Figure 3.1: Overview of system components and operation.
3.1 Concrete Syntax and Abstract Representation

The aartifact system works with arguments that are represented as plain text ASCII files. The user can specify in a succinct and unobtrusive manner which portions of a file are to be considered for machine verification using a pair of delimiters \vbeg ... \vend. Portions of an argument to be considered for machine verification is represented using a familiar and simple concrete syntax that overlaps with English, MediaWiki markup, and \LaTeX: the user may use a selection of \LaTeX constructs in mathematical notation, and can use English phrases as predicates. This approach ensures backward compatibility, in that it is possible to leverage existing knowledge: a user familiar with \LaTeX and English only needs to learn three or four simple syntactic constructs and conventions to begin writing formal arguments immediately. Because our representation itself is no more complex than \LaTeX (and actually much simpler), it is not necessary to use any particular editor or environment, and the user has the flexibility of constructing the proof in any order. Naturally, designers can also construct applications and editors that target this representation, and aartifact can easily be integrated with any editing or content management tool. We have demonstrated this flexibility by integrating it with the MediaWiki content management system,\footnote{Available online at http://www.aartifact.org} allowing users to verify their formal arguments as they are writing or editing them simply by clicking a button on the editing page.

Any portion of an argument to be considered for machine verification consists of a sequence of statements. Each statement introduces variables or logical expressions. For example, the Assume statement signals that the user wants to indicate that an expression is true without subjecting it to verification (and, consequently, introduce it into the context), and an Assert statement signals that the user wishes to perform machine verification on the expression. The aartifact system intentionally does not provide a rich proof language for constructs such as definitions, lemmas, theorems, case distinctions, and so forth. This reduces the amount of new, specialized syntax and semantics a user must learn when employing our system. These structures can easily be introduced outside of the formal syntax (that is, outside the \vbeg ... \vend delimiters) based on the needs and preferences of the user or community of users.

The system’s parser supports several English phrases corresponding to common logical operations (such as implication, conjunction, and quantification). The user is also free to use her own English phrases as predicate constants, and these phrases can contain expression parameters. Our system tries to accommodate a small collection of punctuation symbols commonly used grammatically and in \LaTeX and MediaWiki formatting, allowing them to be placed between English phrases and mathematical expressions.

3.1.1 Concrete Syntax and Parsing

It is the task of the parser to convert into the abstract syntax the concrete syntax of the portion of an argument that is to be considered for verification. The parser for the concrete syntax was constructed in Haskell using the Parsec parser combinator library [LM01], which is expressive enough for constructing infinite lookahead parsers for general context-sensitive grammars. This library is simple to use and allows for a succinct parser implementation. The aartifact parser performs without noticeable delay on all inputs on which it has been tested (the infinite lookahead capability is utilized at only a few points in the parser definition).

3.1.2 Abstract Syntax

We present the abstract syntax for statements, expressions, and constants.

Figure 3.1.2 summarizes the abstract syntax for aartifact. Let \( X \) be the set of all variables. We denote by \( x \) a single variable, and by \( \mathbf{x} \) a vector of variables. We sometimes also denote by \( \mathbf{x} \) the set of variables found in \( \mathbf{x} \) (thus, \( \mathbf{x} \subset X \)). Note that there are only three kinds of statements, and two of them (Assume and Assert) are very similar from the user’s perspective. The expression syntax corresponds to the typical syntax of any higher-order logic and is used to represent a variety of constructions. Figure 3.3 presents a more detailed look at a portion of the abstract syntax corresponding to common constants used in arithmetic and set theory. Whenever there is a possibility of confusion or ambiguity, we use a tilde to distinguish a symbolic
3.1. CONCRETE SYNTAX AND ABSTRACT REPRESENTATION

expressions \( e \) ::=  
\[ \begin{align*} 
& c \\
& x \\
& e_1 e_2 \\
& (e_1, \ldots, e_n) \\
& e_1 \Rightarrow e_2 \\
& e_1 \land e_2 \\
& e_1 \lor e_2 \\
& \neg e \\
& \forall x.e \\
& \exists x.e 
\end{align*} \]

statements \( s \) ::=  
\[ \begin{align*} 
\text{Assume } e & \quad | \quad \text{Assert } e & \quad | \quad \text{Intro } \exists 
\end{align*} \]

Figure 3.2: Abstract syntax.

constant within our abstract syntax from the actual mathematical object (e.g. \( \dot{+} \) or \( \dot{\mathbb{Z}} \) for a constant in our abstract syntax, and \( + \) or \( \mathbb{Z} \) for the mathematical object).

We explain in more detail how some common mathematical constructions are represented within the abstract syntax.

**English phrase predicates.** English phrases acting as predicates can have 0 or more arguments. An English phrase predicate is represented using a list of words (string literals without punctuation or spaces) and placeholders (which we denote \([ \ ]\)). If the English phrase contains any mathematical arguments, the English phrase predicate is applied to a tuple of expressions representing the arguments. For example, the predicate found in the expression \( \backslash p\{\$p\$ is a path in \$G\$\} \) is represented using the list \([[], \text{is}, \text{a}, \text{path}, \text{in}, [], []\] , and the entire expression is represented as
\( \{[], \text{is}, \text{a}, \text{path}, \text{in}, [], []\} (p, G) \).

**Mathematical operators.** Common mathematical operators (such as those in arithmetic and basic set theory) are treated as constants. If an operator has an arity of 2 or above, it is by convention applied to a tuple of expressions. Thus, the input \( \$2 + 2\$ \) is represented as \( \dot{+} (2, 2) \).

**Explicit sets.** The operator \( \{ \} \) is applied to a tuple expression to represent an explicit set within the expression syntax. For example, the set \( \{1, 2, 3\} \) is represented as \( \{\} (1, 2, 3) \).

**Explicit ranges for quantified variables.** Any instance of concrete syntax of the form
\[ \forall x_1 \dot{\in} e_1, \ldots, x_n \dot{\in} e_n.e \]
is converted into abstract syntax of the form
\[ \forall x_1, \ldots, x_n.((x_1 \dot{\in} e_1) \land \ldots \land (x_n \dot{\in} e_n)) \Rightarrow e. \]
3.2 The Simulated Context (Library)

The AARTIFACT system processes a user’s formal argument by performing a rough simulation of the user’s context. The static context is a library that simulates a user’s general knowledge and experience across many
formal reasoning domains, while the dynamic context is a data structure that simulates a user’s knowledge about the components of the particular argument being authored and submitted for verification. As illustrated in Figure 3.4, the static context is compiled into an executable that processes arguments. The executable uses the static context within it to assemble a dynamic context whenever it processes an argument.

![Diagram](image)

Figure 3.4: Illustration of the two components of the AARTIFACT system.

3.2.1 Static Context

The purpose of the static context is to simulate the knowledge and experience of a typical user within a collection of formal domains. It is implemented as a relational database containing a collection of propositions involving common mathematical concepts such as numbers, sets, relations, maps, graphs, and so forth. This database also contains a large collection of common syntactic idioms used to represent mathematical concepts.

The Database of Propositions

The AARTIFACT library of supported propositions and definitions contains hundreds of entries. Each proposition deals with semantic concepts, properties they may have, and relationships that may hold between them. The following proposition represents a very simple example:

"for any \( x, y, z \),
\( x \in \mathbb{R}, \ y \in \mathbb{R}, \ z \in \mathbb{R}, \ x < y, \ y < z \)
implies that
\( x < z \)."

Many of these propositions simply state an equivalence between two forms of notation or syntax. They can be viewed as establishing a normal form for representing certain concepts or properties thereof. For example, the following proposition converts the typical notation for a set of integers in a finite range, \(\{x, \ldots, y\}\), into a predicate that is then used in other propositions about the properties of sets of integers in a finite range:

"for any \( x, y \),
\( x \in \mathbb{Z}, \ y \in \mathbb{Z}, \ x \leq y \)
implies that
\( \{x, \ldots, y\} \) is the set of integers ranging from \( x \) to \( y \)."
CHAPTER 3. AN AUTOMATED ASSISTANT FOR FORMAL REASONING

3.3 Interaction and Real-time Assistance

When employing a machine verifier, a user typically submits her argument to the interface layer for interpretation and verification and, after some processing, feedback is returned to the user in some form. This process might repeat itself as the user assembles or adjusts her argument based on the feedback or on other requirements. The AARTIFACT system is designed with this typical scenario in mind.

The web interface components of AARTIFACT described in this section are implemented using HTML, JavaScript, and PHP. The strategy is similar to the one employed in related work [Kal07], though that work focuses more on delivering (using only a web browser) the particular look, feel, and functionality of an existing proof assistant.
3.3. INTERACTION AND REAL-TIME ASSISTANCE

3.3.1 Argument Submission and Validation Feedback

The AARThIFACT web interface provides a means for selecting one of a collection of validation techniques. As illustrated in Figure 3.6, when validation is requested the raw ASCII text of an argument is submitted to the system.

The system then processes the argument according to the selected validation procedure and returns a marked version of the same raw input as feedback. In this feedback, colors are used to highlight both errors (e.g. unbound variables, unverifiable subexpressions in assertions) and verifiable assertions. Over the web interface, this is accomplished using HTML; if the application is used from the command line, these highlights can also be in the form of ANSI escape sequences for colored text. The executable accomplishes this by maintaining a data structure within the parser that couples the abstract syntax with the original concrete syntax.

In order to demonstrate the feasibility of making the AARThIFACT system available, it has been integrated with the MediaWiki content management system [Wiki]. Figure 3.6 provides a screenshot of a user interacting with this system: the left-hand side of the webpage displays the user’s working formal argument, while the right side displays the same argument after it has been processed. The interface allows users to verify their formal arguments as they are writing or editing them simply by clicking a button on the edit page. It is worth noting that while only the AARThIFACT verification executable is currently utilized with the web interface, any other verification tool with a command-line interface that can accept ASCII input and can produce text or HTML output could be invoked using this interface. The content management system as a whole allows a community of users to maintain a library of results and collaboratively expand it.

3.3.2 Real-time Syntactic Idiom Lookup and Hints

It is unreasonable to expect a user to learn all the possible syntactic constructs and idioms for common concepts. Thus, a real-time keyword lookup system is integrated into the user interface. Whenever a user is typing an argument, the text immediately surrounding the user’s cursor is broken up into keywords, and these are then used to look up and present suggestions and examples of relevant syntactic constructs. Figure 3.6 illustrates such a scenario. The user’s cursor is positioned over text that includes the words “relatively prime”. At the top of the interface, a suggestion appears that provides examples of relevant predicates, and a link to view propositions within the library that include this predicate.

In this way, the system is able to inform the user of what is expected in a context-sensitive manner, and in doing so to establish a mode of communication with a user who may already know about the concepts she wishes to employ, but may not yet be familiar with the system’s syntax or library. This is essential when the user wishes to employ concepts and notations that are not necessarily consistent within the community. For example, the supported forms of notation for closed real number intervals might be \( \{ x \mid 0 \leq x \leq 10 \} \) and \([0,10]\), the notation for the set difference operator might be \( \setminus \) or \(-\), and the notation for concatenation of vectors might be \( \cdot \) or \(\circ\). Informing the user of these conventions within a context in which they are thinking about them saves time and provides an opportunity to learn the system’s syntax within a relevant context. Even if the user is not familiar with any syntactic convention, she may temporarily type keywords related to the concept in question directly into the argument in order to receive information about supported notations for that concept.

This feature is implemented as a Javascript application that is compiled from the contents of the library. The JavaScript application is delivered to the user’s browser whenever the web interface page is loaded, and the user’s browser executes it. This approach makes it possible to provide instant feedback without burdening the server, which must process validation requests and generate feedback.

\(^2\)Available at http://www.aartifact.org.
CHAPTER 3. AN AUTOMATED ASSISTANT FOR FORMAL REASONING

Figure 3.6: A screenshot of the web interface. The user authors a formal argument represented using concrete syntax. As the user types, real-time hints for syntax are provided at the top of the interface based on the text surrounding the cursor. The user can select a logical system or validation procedure and click “Verify” to receive feedback that reproduces the raw input as HTML with color highlights indicating valid and invalid portions of the argument (with blue and red, respectively).

3.4 Inference, Search, and Automation

This section presents the definition of the inference rules for expressions, which are used to determine what kinds of expressions represent “true” assertions. The inference rules for statements, which are used to verify an argument, are also presented.
3.4. Inference, Search, and Automation

3.4.1 Inference Rule Template for Expressions

Denote by $\equiv$ simple syntactic equivalence of expressions. This notion makes no accommodations for renaming; for example, it is the case that

$$\forall x. x \equiv x \not\equiv \forall y. y \equiv y.$$  

Define $FV(e)$ to be the collection of free variables in an expression $e$. We also define substitutions on expressions. For any vector of variables $\mathbf{v}$ and any vector of expressions $\mathbf{e}$, denote by $[\mathbf{v} \mapsto \mathbf{e}]$ the operator that performs a capture-avoiding substitution in the usual manner on any expression (replacing each $x$ in $e$ with the corresponding $e'$ in $\mathbf{e}$). If $\theta = [\mathbf{v} \mapsto \mathbf{e}]$, then $\theta(e)$ and $e[\mathbf{v} \mapsto \mathbf{e}]$ both denote the application of this operator on an expression $e$. For a substitution $\theta$, $\text{dom}(\theta)$ denotes the set of variables bound within the substitution and $\text{ran}(\theta)$ denotes the set of expressions to which the variables in $\theta$ are mapped. Given two expressions $e, e'$ and some variables $\mathbf{v}$, define $\text{match}(\mathbf{v}, e, e')$ to be the substitution that syntactically unifies (makes syntactically equivalent) $e$ and $e'$ by appropriately substituting any free occurrences in $e$ of the variables from $\mathbf{v}$. Note that this is one-sided: the substitution $\theta = \text{match}(\mathbf{v}, e, e')$ is such that $\theta(e) \equiv e'$ but it may be that $e \not\equiv \theta(e')$. Obviously, $\text{match}(\mathbf{v}, e, e')$ can be undefined.

The variable $\Delta$ always denotes a context of bound variables that maps variables to a fixed set of sorts, and $\Phi$ always denotes a set of “true” expressions in a given context. The judgment $\Delta, \Phi \vdash e$ indicates that the expression $e$ has no free variables other than those bound in $\Delta$, that all expressions in $\Phi$ also have no free variables other than those bound in $\Delta$, and that $e$ represents a “true” statement in some sense.

Figure 3.7 presents the logical inference rules that define how such judgments on expressions can be constructed. We call these rules a template because they are parameterized by a syntactically-defined subset of admissible expressions, and can thus be used to represent different logics. However, they can only be used to represent logics in which the sort of a variable within an expression can always be determined using syntax alone. While this may seem like a serious limitation, it is important to note that we view this capability merely as a bookkeeping convenience that allows us to keep track of which logical system a user is trying to employ, and not as a restriction on the system. When verifying an expression, a verification algorithm attempts to infer it under many possible instantiations of the template. As a last resort, it tries the trivial instantiation under which all variables are of the same sort and all expressions are admissible. This last resort corresponds to a kind of “lightweight” verification strategy.

A straightforward way to represent a particular instantiation of the inference rules is to define four sets $s^e, t^e, a^e, e^e$ of the space of syntactically well-formed expressions (corresponding to sorts, terms, atoms, and sentences as defined in popular presentations of logic [Kle67]). Examples can be found in a relevant report [Lap09a], in which it is also explained how these instantiations ensure the consistency of the inference rules with respect to particular logical systems. Notice that occurrences of variables within the syntax definitions (represented by $x_{\text{sort}}$) are labelled by the syntactically-determined sort of all variables found in that position.

The inference rules interact with the parameters $s^e, t^e, a^e, e^e$ in two ways. First, an additional premise is implied for every inference rule in Figure 3.7:

$$\vdash \Delta; \Phi \vdash e$$

We omitted this rule for the sake of clarity. It ensures that only sentences as defined by the parameter $e^e$ can be inferred. Second, two functions $\sigma$ and $\xi$ are used to determine the sorts of variables and expressions. The function $\sigma$ takes an expression and a variable, and returns the syntactically-determined sort of that variable within the expression. If it cannot determine a sort or if there is an inconsistency in the way the variable is used, the function is undefined. The function $\xi$ takes a context $\Delta$ and an expression $e$, and determines its sort within that context by checking which syntactic definition $t^e$ the expression respects and using $\Delta$ to determine the sort of variables within the expression. If it cannot determine a sort or there is a contradiction, the function is undefined. The symbols $\sigma$ and $\xi$ are overloaded to also represent the vector versions of these functions (corresponding to element-wise mappings of $\sigma(e)$ on a vector of variables and $\xi(\Delta)$ on a vector of expressions).

Some of the inference rules are worth addressing. The $[\forall\text{-}\text{Intro}]$ rule discards from the context any formulas that contain (as free) the variables being introduced by the universal quantifier. The reader may
wonder why α-conversion could not address this problem. While this may be a reasonable solution when the expressions in question are only viewed by a machine in their final form, it makes less sense when we consider the perspective of a user employing step-by-step forward reasoning. If the user wishes to employ an assumption involving a variable that is currently within the context, she is unlikely to re-introduce this variable by quantifying over it universally; in fact, if she does do so, it is likely to be an error. Nevertheless, it is worth noting that our definition of this rule is based on our preferences, and alternative approaches are not ruled out for the future.

Note that the first premise of ∨-Intro not only requires that the syntactically-determined sorts of the newly-introduced variables be recorded within Δ, it requires that the variable have the same sort in every subexpression of e in which it appears (otherwise, σ is undefined). Both the ∨-Elim and 3-Intro rules require that the sorts of the expressions σ match the sorts of the variables σ that they will replace. This ensures that any attempt to substitute variables for expressions respects the sorts defined within a logical system.

### 3.4.2 Inference Rules for Statements

An argument consists of a sequence of statements $s_1; \ldots; s_n; \text{end}$, where end is implicit. Let $s; S$ denote that a statement $s$ is followed by a list of statements $S$. The logical inference rules for statements are found in Figure 3.8. Using an Assume statement, a user can introduce a new expression into the assumption context, so long as it has no free variables with respect to the variable environment Δ. An assertion statement is treated the same way as an assumption, with the added restriction that $e$ must be derivable from the given assumption context $\Phi$ according to the inference rules for expressions in Figure 3.7. In an introduction statement, the list of named variables is added to the variable context and any formulas dealing with those variables are removed from the assumption context. The function $\sigma$ is extended to determine the sort of the variable in the same manner, but by examining all expressions found in the list of statements $S$ that represents the rest of the document.
3.4.3 Dynamic Context

The dynamic context is a data structure that simulates the user’s awareness and understanding of all the relevant expressions within an individual argument. This data structure is coupled with the assumption context in the inference engine. It data structure is defined to be a hypergraph in which the nodes represent equivalence classes of argument expressions, and in which the hyperedges represent common mathematical relations, both low-level and high-level.

Definitions

For any set \( S \), define \( P(S) \) to be the power set of \( S \) and \( S^* \) to be \( \bigcup_{i=0}^{\infty} S^i \). Let \( \mathcal{E} \) be the space of all syntactically well-formed (not necessarily closed) logical expressions. For the purposes of this discussion, we assume expressions can contain predicates, operators, and logical operators (including implication, conjunction, and universal quantification, which we denote by \( \Rightarrow \), \( \land \), and \( \forall \), respectively). For a full and extensive presentation of the syntax of expressions, we refer the reader to previous work [Lap09a]. Let \( \mathcal{R} \) be the set of all predicate and relation symbols. In particular, it is the case that

\[
\{+, -, :, /, \cup, \cap, \times, \rightarrow\} \subseteq \mathcal{R},
\]

but \( \mathcal{R} \) also contains all possible user-introduced symbols for operators, relations, and predicates. Let \( \text{arity} : \mathcal{R} \rightarrow \mathbb{N} \) be a function that returns the arity of the relation to which each symbol corresponds. As a convention, we will refer using \( E \subseteq \mathcal{E} \) to a finite subset of \( \mathcal{E} \), and we will refer using \( R \subseteq \mathcal{R} \) to a finite subset of \( \mathcal{R} \).

Definition 3.4.1. For a finite \( E \subseteq \mathcal{E} \), let \( Q_E \) be the set of equivalence contexts over \( E \), where for all \( Q \in Q_E \) it is always the case that \( Q \subseteq P(E) \).

Definition 3.4.2. For a finite \( R \subseteq \mathcal{R} \), a finite \( E \subseteq \mathcal{E} \), and \( Q \in Q_E \) let \( \mathcal{C}_{Q,R} = P(R \times Q^*) \) be the collection of relation contexts over \( Q \) and \( R \). Each \( C \in \mathcal{C}_{Q,R} \) is a set of vectors. For each vector \( \bar{r} \in C \) there exists \( r \in R \) such that \( \bar{r} \in \{r\} \times Q^{\text{arity}(r)} \). Note that for each \( r \in R \), there exists \( C_r \subseteq C \) such that \( C_r \subseteq \{r\} \times Q^{\text{arity}(r)} \).

Note that for any finite \( R \subseteq \mathcal{R} \), finite \( E \subseteq \mathcal{E} \), and \( Q \in Q_E \) it is possible to represent any individual \( C \in \mathcal{C}_{Q,R} \) as a hypergraph in which \( Q \) is the set of nodes and each entry in \( C_r \) for every \( r \in R \) is an edge, labelled \( r \), connecting \( \text{arity}(r) \) nodes. Because the range of the function \( \text{arity} \) is \( \mathbb{N} \), such a graph could contain some number of distinct edges that link no nodes. However, this number of edges is bounded because \( R \) is finite.

Lemma 3.4.3. For any finite \( R \subseteq \mathcal{R} \), finite \( E \subseteq \mathcal{E} \), and \( Q \in Q_E \), it is the case that \( \mathcal{C}_{Q,R} \) is of finite size, and for every \( C \in \mathcal{C}_{Q,R} \), \( C \) is of finite size.

Proof. Because \( E \) is finite, \( P(E) \) is finite and so every \( Q \subseteq P(E) \) is finite. This means that for any \( r \in R \), \( Q^{\text{arity}(r)} \) is finite. Because \( R \) is finite, there exists a finite number of finite sets \( Q^{\text{arity}(r)} \) where \( r \in R \). Thus,

\[
\bigcup_{r \in R} \left( \{r\} \times Q^{\text{arity}(r)} \right) \supseteq C
\]
is a finite union of finite sets, so every \( C \) is finite. Furthermore, this implies that

\[
\mathcal{C}_{Q,R} = P\left( \bigcup_{r \in R} \{ r \} \times Q^{\text{arity}(r)} \right)
\]

is finite.

**Amenable Formulas**

For any set \( S \), let \( \bar{x} \) be any vector of elements drawn from \( S^* \). We sometimes abuse notation by using \( \bar{x} \) to refer to the set of elements in the same vector; this is always clear from the context. If \( \bar{x} \) and \( \bar{y} \) are vectors, then we say that \( \bar{x} \subseteq \bar{y} \) if and only if every variable contained in \( \bar{x} \) is also found in \( \bar{y} \). Let \( A' \) represent the space of all variables.

**Definition 3.4.4.** For any finite \( R, E, Q \), we call a formula \( c \in E \) amenable if it is of the form

\[
\forall \bar{x}, r_1(\bar{x}_1) \land \ldots \land r_n(\bar{x}_n) \Rightarrow r_{n+1}(\bar{x}_{n+1})
\]

where for all \( i, r_i \in R \) and \( \bar{x}_i \in (\bar{x} \cup Q)^* \). That is, if \( u \in \bar{x}_i \), then \( u \) may be an equivalence class in \( Q \) or a variable in \( \bar{x} \).

Note that an amenable formula is a specific kind of Horn formula. Let \( A_{Q,R} \subseteq E \) be the set of all amenable formulas under a given \( Q \) and \( R \).

For any \( \bar{x} \in (A' \cup Q)^* \), let \( \bar{x}[\bar{y}/\bar{x}] \) denote the result of a substitution that replaces any instances \( u_i \in \bar{x} \) that are in \( \bar{y} \) with a corresponding \( q \in \bar{y} \), and leaves any instances of \( u_i \in \bar{x} \) that are in \( Q \) untouched. Let \( \equiv \) represent equality over \( C \) in any relation context \( C \in \mathcal{C}_{Q,R} \). We relate entries in some \( C \in \mathcal{C}_{Q,R} \) to formulas according to the following definition.

**Definition 3.4.5.** For any \( C \in \mathcal{C}_{Q,R} \) and \( \bar{x} \in A' \), for any \( \bar{y} \in C \), we say that \( \bar{x} = (c_1, \ldots, c_n) \) matches with respect to \( \bar{x} \) the formula

\[
r_1(\bar{x}_1) \land \ldots \land r_n(\bar{x}_n)
\]

if for all \( i, \bar{x}_i \subseteq \bar{x} \cup Q \) and there exists \( \bar{y} \in (C_{r_1} \times \ldots \times C_{r_n}) \) and \( \bar{y} \in Q^* \) such that for all \( i, c_i \equiv (r_i, \bar{x}_i[\bar{y}/\bar{x}]) \).

We can now define the closure of a relation context.

**Definition 3.4.6.** For any \( C \in \mathcal{C}_{Q,R} \) and \( A \subseteq A_{Q,R} \) define transit\((C, A)\) to be a relation context \( C' \in \mathcal{C}_{Q,R} \) where \( C \subseteq C' \) and in which it is the case that for every \( a \in A \) of the form

\[
\forall \bar{x}, r_1(\bar{x}_1) \land \ldots \land r_n(\bar{x}_n) \Rightarrow r_{n+1}(\bar{x}_{n+1}),
\]

if a collection of edges \( \bar{x} \subseteq C \) matches with respect to \( \bar{x} \) the formula \( r_1(\bar{x}_1) \land \ldots \land r_n(\bar{x}_n) \) then \((r_{n+1}, \bar{x}_{n+1}[\bar{y}/\bar{x}]) \in C'\).

**Definition 3.4.7.** For any \( C \in \mathcal{C}_{Q,R} \) and \( A \subseteq A_{Q,R} \) define,

\[
C^{(0)} = C \\
C^{(i+1)} = \text{transit}(C^{(i)}, A),
\]

and let

\[
\text{closure}(C, A) = \bigcup_{i \in \mathbb{N}} C^{(i)}.
\]

We are interested in implementing an efficient algorithm that computes \( \text{closure}(C, A) \), and we discuss the implementation in Section 3.4.3. However, we first argue that there necessarily exists an algorithm for computing \( \text{closure}(C, A) \) that terminates.
3.4. INFERENCE, SEARCH, AND AUTOMATION

Theorem 3.4.8. For any $C \in \mathcal{C}_{Q,R}$ and $A \subset \mathcal{A}_{Q,R}$, closure$(C, A)$ is of finite size.

Proof. We know by Lemma 3.4.3 that any $C \in \mathcal{C}_{Q,R}$ is finite, and there are finitely many such sets. This means that there is an upper bound $b \in \mathbb{N}$ on the size of any set $C \in \mathcal{C}_{Q,R}$.

We know that for any $C \in \mathcal{C}_{Q,R}$, it is the case that $C^{(0)} \in \mathcal{C}_{Q,R}$, so it is finite. For any $C^{(i)} \in \mathcal{C}_{Q,R}$, for any $i$, if any $\bar{e} \subset C^{(i)}$ matches the left-hand side of any formula $a \in A$, then the substitution $[\bar{e}/\bar{T}]$ cannot introduce any new equivalence classes, so all equivalence classes within $\bar{e}_{n+1}[\bar{e}/\bar{T}]$ must already be drawn from $Q$. Thus, $C^{(i+1)} \in \mathcal{C}_{Q,R}$, so $C^{(i+1)}$ is also finite.

Furthermore, for any $i$, one of two cases must hold. If $C^{(i+1)} = C^{(i)}$, then $C^{(j)} = C^{(i)}$ for all $j \geq i$, so closure$(C, A) = C^{(i)}$, so closure$(C, A)$ is finite. If $C^{(i+1)} \neq C^{(i)}$, then $C^{(i+1)} \supset C^{(i)}$, which means $b - |C^{(i+1)}| < b - |C^{(i)}|$.

Thus, a single application of $\text{transit}$ must either have no effect, or increase the size of the set. The number of such possible increases is bounded by the finite value $b \in \mathbb{N}$, so closure$(C, A)$ is finite.

Implementation and Performance Evaluation

Our implementation of a data structure for relation contexts drawn from $\mathcal{C}_{Q,R}$ utilizes two auxiliary data structures: one for the set $Q$ and another for the set $C$. Both data structures are represented as finite association maps. The set $Q$ is represented as a finite map $M_e : E \rightarrow Q$ and the set $C \in \mathcal{C}_{Q,R}$ is represented as a finite map $M_r : R \rightarrow \{C_r \mid r \in R\}$.

In our implementation, we represent association maps using purely functional red-black trees. Our implementation is a variant of a classic example of a pure functional implementation of this data structure [Oka99]. The data structure is polymorphic in both the type of the indices and the type of the range values to which indices are mapped (this is essential for our application, as described further below). Like the classic example, our structure supports only insertion of new associations between indices and values, and an update of the value associated with an existing index. There is no support for removal, and such a capability is not required for our application because of the pure evaluation semantics of the Haskell language.

Throughout the process of verification of a formal argument, the formal reasoning system maintains a simulated natural context that includes some $E$, $R$, $Q$, and $C \in \mathcal{C}_{Q,R}$. At each point in this process, the system may encounter an additional assumption in the form of an expression introduced by the user. If this expression can be converted to some $(r, \bar{e}) \in R \times E^*$, it is then necessary to determine the equivalence classes corresponding to $\bar{e}$. By using the association map $M_e$ that is implemented using a data structure that supports insertion and retrieval operations of a logarithmic complexity, it is possible to both adjust $M_r$ to some $M'_r$ (if new expressions were introduced) and obtain the list of equivalence classes $\bar{e}$ in time $O(|\bar{e}| \log |Q|)$. Next, it is necessary to compute $C' = \text{closure}(C \cup (r, \bar{e}), A)$. Our algorithm accomplishes this by repeatedly applying the transit operation until it detects convergence.

Worst-case Upper Bound on Size of the Dynamic Context

Suppose that $A$ is finite, and that the arity of relations is bounded by $k = \max\{\text{arity}(r) \mid r \in R\}$. The size of the dynamic context is then bounded by

$$O(|R| \cdot |Q|^k).$$

This implies that for any $C \in \mathcal{C}_{Q,R}$, $|C| \leq |Q|^{k+1}$.

Worst-case Upper Bound on Duration of the Dynamic Context Closure Computation

As before, suppose that $A$ is finite. Suppose also that conjunction lists on the left-hand sides of the formulas in $A$ are at most of length $n$.

In a single computation of $\text{transit}$, for a given amenable formula $a$ containing $n$ subexpressions in its premise, at most $n$ subsets of $C$ must be obtained (corresponding to various subsets $C_r$), and this takes time
$O(n \log |R|)$, which is negligible. It is then necessary to check if any $\tau \in C^n$ matches the premises of $a$, and this takes time $O(|C|^n)$, or $O(|Q|^{k+1})$. Thus must be repeated for each $a \in A$. In the worst case, each transit computation contributes only one of a possible $|Q|^{k+1}$ new elements, and all the distinct elements are generated in the closure computation before it converges. This results in a worst-case complexity of

$$O\left(|A| \cdot |Q|^{(k+1)(n+1)}\right).$$

Practical Upper Bounds and Performance Profiling

While the worst-case complexity of the closure computation is characterized by a polynomial of a potentially high degree, the computation’s complexity under practical conditions is substantially lower. We characterize practical conditions in two ways: by noting constant upper bounds on parameters within the complexity formula, and by considering a topological restrictions on the hypergraph representing a relation context.

For the collection of several hundred amenable formulas $A$ within our current implementation of the formal reasoning system, the number of premises in a formula’s conjunction list is bounded above by 3. Similarly, the arity of relations in $R$ is also bounded above by 3. The $|Q|^{k+1}$ factor in our worst-case complexity formula represents a scenario in which each computation of transit adds exactly one new element to the relation context. However, a much more common scenario is one in which this operation adds a collection of elements to the relation context, and only a few more transit computations are necessary to reach convergence. One possible hypothesis is to assume that about $\log(|Q|^{k+1}) = (k+1) \log |Q|$ computations of transit are necessary to reach convergence. Adopting this assumptions and substituting $n$ and $k$ for our upper bounds on their averages, we obtain a complexity of

$$O(|A| \cdot |Q|^4 \cdot 4 \log |Q|).$$

During the verification of a typical example proof (such as our proof that $\sqrt{2}$ is irrational, presented in an earlier report [Lap09a]), $|Q|$ does not exceed 100, which means our complexity formula’s most significant term, $|Q|^4$, does not exceed $10^8$. This number of operations can take place in under 4 seconds on a modern 3.0GHz processor with 1GB of RAM.

Figure 3.9: $\sqrt{2}$ is irrational

Figures 3.9, 3.10, and 3.11 present the growth rate of the dynamic context components $R$, $E$, and $Q$ for a collection of example arguments and a particular static context $A$. The data in Figure 3.9 is produced by processing the formal argument in Chapter 1, Figure 1.3. These data illustrate that $|R|$ tends to exhibit linear growth in terms of $|E|$ on the examples presented.

Figure 3.12 illustrates one worst-case scenario, corresponding to a sequence of assumptions of the form

$$a_1 < a_2, \ a_2 < a_3, \ldots, \ a_{39} < a_{40}.$$
Such an argument leads to quadratic growth in the size of the dynamic context hypergraph.

There are two important observations to make about such worst-case scenarios. Most relations involving the basic, common concepts used in mathematical practice have arity between 0 and 4, with the overwhelming majority of relations having arity 1 or 2. Table 3.4.3 presents the distribution of arities for a given static context.

Furthermore, this worst-case behavior becomes increasingly unlikely for a relation as the arity of the relation grows. This is because the arguments of a relation with arity 3 or greater are often drawn from different classes. For example, if we consider the relation $r(S,i,j)$ if $S$ is the set of integers bounded on both sides by $i$ and $j$,

we see that $S$ must necessarily come from a different class of expressions (sets) than do $i$ and $j$ (integers). While the worst case for the number of hyperedges labelled with $r$ is still $O(n^3)$ even under these restrictions (if there are $\frac{n}{2}$ distinct integers and $\frac{n}{2}$ distinct sets within the context), a user can be expected within a typical argument not to introduce a large collection of distinct items from the same class within a given scope.

Another way that a formal argument can lead to rapid growth of the dynamic context hypergraph is if
large algebraic expressions are introduced:

Assert for any $a \in \mathbb{R}$, $14a = a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a$.

However, all of the formal arguments that have been assembled in the course of evaluating the system have conformed to a constraint on the complexity of individual expressions. Figures 3.13, 3.14, and 3.15 show the size of each expression considered over the course of an evaluation of formal arguments.

It is the objective of future work to better characterize “typical” formal arguments, and perhaps even to detect unusual arguments that can cause the dynamic context to grow to an unmanageable size. This would allow the system to process such arguments in some more limited manner.

Table 3.1: Distribution of arity relations in a static context of about 400 propositions.

<table>
<thead>
<tr>
<th>arity</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td># relations</td>
<td>1</td>
<td>57</td>
<td>48</td>
<td>10</td>
<td>3</td>
<td>119</td>
</tr>
</tbody>
</table>
3.4. INFERENCE, SEARCH, AND AUTOMATION

Figure 3.13: $\sqrt{2}$ is irrational

Figure 3.14: $\mathbb{R}$ is a vector space $\Rightarrow \mathbb{R}^2$ is a vector space

Figure 3.15: soundness of the NetSketch formalism
Part II

Use Cases, Applications, and Integrations
Chapter 4

Evaluation of Machine Verifiers

Determining the effectiveness of a formal reasoning and verification system can be a highly subjective activity. However, it is possible to introduce some objective metrics and criteria, as has been done in side-by-side comparisons of existing systems [Wie03]. If we consider this particular comparison, the design of AARTIFACT is deliberately intended to facilitate a large library and a large amount of automation, though these are addressed within the context of user context simulation.

Many existing approaches to evaluation of machine verification systems focus on very low-level proofs. For example, the designers of Scunak [Bro06] propose that proofs of fundamental concepts, such as algebraic distribution laws, are a good touchstone for measuring the success of proof verification systems. On the contrary, focusing on such proofs distracts from efforts directed towards recognizing the larger proof steps humans prefer to use in practice, ones in which distributivity is one of many natural, implicit manipulations.

A more effective evaluation of the usability and flexibility of a machine verification system is its application both in instructional settings and in the pursuit of ongoing and novel research efforts. The AARTIFACT system has been utilized within the classroom by instructors when presenting examples, by students when authoring formal solutions, and by course staff when grading student solutions. The AARTIFACT system has also been applied within the context of novel research; it has been used in the formal development of the NetSketch formalism [LK09, BKLO10].
Chapter 5

Classroom Instruction of Formal Reasoning

The AARTIFACT system has been deployed within two undergraduate courses: an advanced undergraduate course on functional programming [Lap09b], and an introductory undergraduate course in linear algebra. These deployments served as a means both for evaluating the usability of the AARTIFACT system, identifying necessary improvements to the system’s capabilities, and, more generally, observing any pedagogical and administrative advantages (and disadvantages) there are to using a machine verification system in classroom instruction of the formal reasoning topics involved in courses of this kind.

5.1 Classroom Instruction of Reasoning about Pure Functional Programs

The curriculum course on functional programming typically covers are pure functional programming and algebraic reasoning about functional programs, including the assembly of inductive proofs of propositions about programs. The AARTIFACT system was used to present examples of proofs by induction, and was distributed to students within the framework of a homework assignment. Students used the system to assemble their proofs. Each student’s solutions were graded according to a superficial inspection of the student’s formal assumptions and an automated verification of the student’s formal assertions. Shortly after the assignment, the students completed a classroom examination in which they had to assemble a formal proof using pencil and paper (without access to the formal reasoning system).

5.1.1 Related Work in Instruction of Reasoning about Functional Programming

There have been attempts to teach courses on the topic of program verification since the advent of Hoare logic [Hoa69]. Currently, there exist graduate-level, and some undergraduate-level, courses that cover the specific topic of rigorous machine-assisted formal reasoning and proof authoring using systems such as Coq [PV97]. There also exists a great deal of material demonstrating how systems such as Coq can be used to reason formally about both imperative programs [CMM+09, BC04], and functional programs [Cha09, Mol97]. Other formal reasoning tools specifically designed for reasoning about functional code [Col96] exist, as well. Some have reported on their experience using similar systems in the classroom [WS04]. Our report is distinguished from this large body of work by our emphasis on a lightweight formal verification approach using a formal reasoning system whose design is oriented around simulating a natural context. There is also work in separate communities on using formal methods in the instruction of the more general task of software development.
development [OP98], and systems such as Alloy [Jac02] have also been deployed in the classroom.² Our work is distinguished from this by our application of the lightweight approach for very local and specific results about individual functions within functional code usually addressed by heavyweight systems used in the first body of work.

The EPGY Theorem-Proving Environment [MRS01] is designed for teaching algebra within a classroom setting at the levels of middle and high school. Its design provides extensive support for algebraic manipulations and computations. However, its design does not directly support a lightweight approach to verification: manipulations are strictly enforced, and authoring arguments requires the use of an interactive interface.

5.1.2 Examples of Verifiable Formal Proofs

We list all of the examples presented to students, as well as all of the assigned problems and their solutions. Each example or problem involves a proof by induction of a formal proposition about a collection of one or more functions. Most examples are drawn directly from an introductory text on functional programming [Tho99]. In this article, we delimit the mathematical notation with pound signs (# ... #) to avoid invoking the \texttt{\LaTeX} parser. We note that the contents of this section can be supplied directly to and verified by the formal reasoning system provided to students.

Examples with recursively-defined lists

The option to perform lightweight verification makes it possible to focus on certain aspects of the formal reasoning and proof authoring process without being bogged down in other aspects. For instance, the following examples make no explicit mention of the induction principle on lists, or of an explicit equality function on lists. In fact, there is no mention of types. Nevertheless, some verification is possible: one can gain more confidence that a proof is correct by confirming that the proof can be verified automatically, and then briefly inspecting the assumptions and superficial structure of the proof. This brief manual inspection is significantly more simple than a manual proof verification.

The following introduces the two symbols corresponding to an empty list and a list head, as well as assumptions corresponding to several functions typically defined for within functional languages for recursively-defined lists.

\beg
\begin{verbatim}
Introduce #cons, nil#.
Introduce #length, append, map, foldr#.
Assume # length nil = 0#.
Assume #\forall x,xs. length (cons x xs) = 1 + length xs#.
Assume #\forall ys. append nil ys = ys#.
Assume #\forall x,xs,ys. append (cons x xs) ys = cons x (append xs ys)#.
Assume #\forall f. map f nil = nil#.
Assume #\forall f,x,xs. map f (cons x xs) = cons (f x) (map f xs)#.
Assume #\forall f,b. foldr f b nil = b#.
Assume #\forall f,b,x,xs. foldr f b (cons x xs) = f x (foldr f b xs)#.
\end{verbatim}
\end{verbatim}
\vend

These functions (sometimes under a different name) can be found in the Haskell [Je99] standard library.

²A list of courses is found at \url{http://alloy.mit.edu/community/courses}.
Mapping an identity

Suppose that the identity function is introduced.

\begin{verbatim}
\beg
Introduce #idf#.
Assume #\forall x. idf x = x#.
\end{verbatim}

We want to show that for all lists, applying \texttt{map idf} to the list will return an equivalent list. The base case is a direct application of the base case definition for \texttt{map}.

\begin{verbatim}
\beg
Assert #map idf nil = nil#.
\end{verbatim}

To argue that the inductive step is correct, we show that if we assume that our desired result holds for some list \texttt{xs}, it also holds for a list that has an additional element \texttt{x} added to the beginning of the list.

\begin{verbatim}
\beg
\assert
\forall x,\forall x\texttt{ }\texttt{xs}.
\begin{align*}
\text{map idf xs} &= \texttt{xs} \\
\text{cons x (map idf xs)} &= \texttt{cons x xs} \\
\text{map idf (cons x xs)} &= \texttt{cons (idf x) (map idf xs)} \\
\text{idf x} &= \texttt{x} \\
\text{map idf (cons x xs)} &= \texttt{cons x (map idf xs)} \\
\text{cons x (map idf xs)} &= \texttt{cons x xs} \\
\text{cons x (map idf xs)} &= \texttt{cons x xs}
\end{align*}
\end{verbatim}

Notice that in both the base case assertion, and in the sequence of equations in the proof of the inductive step, it is never necessary to explicitly reference which property of equality is being applied, or which assumptions is being instantiated. The user is free to supply comments for steps that she feels might require explanation, but can safely omit this information in cases where this would only serve to clutter the argument.

Folding over an associative operator

Suppose we have an associative binary operator and corresponding identity.

\begin{verbatim}
\beg
\Introduce #op, id#.
Assume #\forall x. op id x = x#.
Assume #\forall x. op x id = x#.
Assume #\forall x, y, z. op x (op y z) = op (op x y) z#.
\end{verbatim}

We want to show that for any two lists, concatenating then folding with respect to the operator is the
same as folding each of the two lists, and then concatenating the results. We first present the base case.

\vbegin
\begin{assert}
\forall ys.
\begin{align*}
\text{append } \text{nil} \text{ } \text{ys} & = \text{ys} \\
\text{foldr } \text{id } \text{ys} & = \text{foldr } \text{id} \text{ (append } \text{nil } \text{ys)} \\
\text{op } \text{id } \text{foldr } \text{id } \text{ys} & = \text{id} \\
\text{op } \text{id } \text{foldr } \text{id } \text{(append } \text{nil } \text{ys)} & = \text{foldr } \text{id } \text{ys}
\end{align*}
\end{assert}
\vend

Next, we present the inductive step.

\vbegin
\begin{assert}
\forall x, xs, ys.
\begin{align*}
\text{op } \text{foldr } \text{id } \text{xs} \text{ (foldr } \text{id } \text{ys)} & = \text{foldr } \text{id} \text{ (append } \text{xs } \text{ys)} \\
\Rightarrow
\begin{align*}
\text{op } x \text{ (op } \text{foldr } \text{id } \text{xs} \text{ (foldr } \text{id } \text{ys)}) & = \text{op } x \text{ (foldr } \text{id} \text{ (append } \text{xs } \text{ys)}) \\
\text{append } \text{(cons } x \text{ (append } \text{xs } \text{ys)} & = \text{cons } x \text{ (append } \text{xs } \text{ys)} \\
\text{foldr } \text{id} \text{ (cons } x \text{ (append } \text{xs } \text{ys)}) & = \text{foldr } \text{id} \text{ (append } \text{cons } x \text{ xs) ys)
\end{align*}
\end{align*}
\end{assert}
\vend

Identity of append

Suppose we want to show that the empty list is also an identity for the function that appends lists. The definition of the function already shows that it is a left identity, so all that remains is to show by induction that it is a right identity. The base case is trivial.

\vbegin
\begin{assert}
\text{append nil nil = nil}
\end{assert}
\vend

The inductive case is only slightly more complex.
5.1. CLASSROOM INSTRUCTION OF REASONING ABOUT PURE FUNCTIONAL PROGRAMS

Associativity of `append`

Suppose we want to show that the function for appending lists is associative. We only need to perform induction over the first argument to the function because the second argument is never decomposed within the function’s definition. We begin with the base case.

```
vbeg
  Assert
  $\forall ys, zs.
    \ append ys zs = append ys zs
    \ append nil ys = ys
    \ append nil (append ys zs) = append ys zs
    \ append (append nil ys) zs = append nil (append ys zs)
  $.  
vend
```

We present the inductive step.

```
vbeg
  Assert
  $\forall x, xs, ys, zs.
    \ append (append xs ys) zs = append xs (append ys zs)
  \Rightarrow
    $%-- put "x" on both sides
    \ cons x (append (append xs ys) zs) = cons x (append xs (append ys zs))$
    $%-- right side
    \ append (cons x xs) (append ys zs) = cons x (append xs (append ys zs))$
    $%-- left side
    \ append (append (cons x xs) ys) = cons x (append (cons x xs) (append ys zs))$
    \ append (append (cons x (append xs ys)) zs = append (cons x (append (append xs ys)) zs)
    \ append (append (cons x (append xs ys)) zs = append (cons x (append (append xs ys)) zs)
    \ append (append (cons x (append xs ys)) zs = append (cons x (append (append xs ys)) zs)
  $.  
vend
```
User-defined functions

We write a proof drawn directly from an example in an introductory text on functional programming [Tho99]. If our formal reasoning system were restricted to a straightforward search algorithm that verifies logical expressions (without native support for algebraic manipulation of equations), the following assumptions might be needed. However, they are not necessary for our formal reasoning system. We reproduce them here for the sake of completeness.

\begin{verbatim}
Assume \forall x. 0 = 2 \cdot 0
Assume \forall x,y,z. (z\times x) + (z\times y) = z\times (x+y)
\end{verbatim}

We now present the example. Suppose we have defined the following two functions.

\begin{verbatim}
\beg
Introduce #sum, doubleAll#.
Assume # sum nil = 0#.
Assume #\forall x,xs. sum (cons x xs) = x + (sum xs)#.
Assume # doubleAll nil = nil#.
Assume #\forall z,zs. doubleAll (cons z zs) = cons (2 \cdot z) (doubleAll zs)#.
\end{verbatim}

We now want to prove that for any list of integers, it is the case that doubling the sum of its elements is equivalent to computing the sum of the doubled elements of the list. We start with the base case.

\begin{verbatim}
\beg
Assert #
0 = 2 \cdot 0
\wedge sum nil = 2 \cdot (sum nil)
\wedge sum (doubleAll nil) = 2 \cdot (sum nil)
#.
\end{verbatim}

Next, we present the inductive step.
5.1. CLASSROOM INSTRUCTION OF REASONING ABOUT PURE FUNCTIONAL PROGRAMS

\[ \forall x,xs. \sum (\text{doubleAll } xs) = 2 \cdot (\sum xs) \]

\[ \Rightarrow \]

\[ (2 \cdot x) + \sum (\text{doubleAll } xs) = (2 \cdot x) + 2 \cdot (\sum xs) \]

\%-- We first rewrite the left side.
\[ \wedge (2 \cdot x) + (2 \cdot (\sum xs)) = 2 \cdot (x + \sum xs) \]
\[ \wedge \sum (\text{cons } x xs) = x + \sum xs \]
\[ \wedge (2 \cdot x) + (2 \cdot (\sum xs)) = 2 \cdot (\sum (\text{cons } x xs)) \]

\%-- Next, we rewrite the right side.
\[ \wedge \sum (\text{cons } (2 \cdot x) (\text{doubleAll } xs)) = (2 \cdot x) + \sum (\text{doubleAll } xs) \]
\[ \wedge \text{doubleAll } (\text{cons } x xs) = \text{cons } (2 \cdot x) (\text{doubleAll } xs) \]
\[ \wedge \sum (\text{doubleAll } (\text{cons } x xs)) = (2 \cdot x) + \sum (\text{doubleAll } xs) \]

\%-- Finally, we put the two sides back together.
\[ \wedge \sum (\text{doubleAll } (\text{cons } x xs)) = 2 \cdot (\sum (\text{cons } x xs)) \]
\#. 
\[ \end{verbatim} \]

The append and length functions

For this next example, we again present a few assumptions that are not required by our formal reasoning system.

Assume \[ \forall x. 0 + x = x \]
Assume \[ \forall x,y,z. x+(y+z) = (x+y)+z \]

Suppose we want to prove that taking the lengths of two lists and adding them is equivalent to appending those same two lists and then computing the length of the result. We present the base case.

\[ \begin{verbatim}
\beg
_assert
#
forall ys. 
\append nil ys = ys
\wedge \length \append nil ys = \length ys
\wedge 0 + \length ys = \length ys
\wedge \length \append nil ys = 0 + \length ys
\wedge \length \append nil ys = \length nil + \length ys
#.
\end{verbatim} \]

Next, we present the inductive step.
Examples with other datatypes

Naturals

Suppose we have the following data type definition (e.g. in Haskell), representing natural numbers. The \( Z \) constructor corresponds to 0, and the \( S \) constructor corresponds to (+1).

\[
\begin{align*}
data Nat = & Z | S Nat \\
\end{align*}
\]

Furthermore, suppose we defined addition on objects of type "Nat" using the following function:

\[
\begin{align*}
\text{plus } Z & n = Z \\
\text{plus } (S m) & n = S (\text{plus } m n) \\
\end{align*}
\]

If we want to model this situation mathematically, we would introduce all the defined entities (the constructors and function), and then write universally quantified equations to represent the function definition.

\[
\begin{align*}
\text{Introduce } & Z, S, \text{ plus}. \\
\text{Assume } & \forall n. \text{ plus } Z n = Z. \\
\text{Assume } & \forall m, n. \text{ plus } (S m) n = S (\text{plus } m n). \\
\end{align*}
\]

Now, suppose we want to show that for any \( x \), we also have that

\[
\text{plus } x Z = x
\]
The equations alone don’t tell us this directly, so we need to prove it by induction. In other words, we have to show that the above statement holds for all possible \( x \) of type \( \text{Nat} \) by first showing that the \( x = \text{Z} \) case holds, and then showing that if it holds for some \( x \), then it holds for \( \text{S} \; x \).

First, we verify that the base case holds. The below assertion is verified automatically because it is merely an application of the first assumption above in which the \( n \) in \( \forall n. \ldots \) is instantiated to \( \text{Z} \).

\[
\text{Assert} \ #\text{plus Z Z = Z#. %-- by definition of "plus"}
\]

Now, we review two operators from logic: implication (\( \rightarrow \)) and conjunction (\( \wedge \)). These are understood by the verifier in the usual way. For example, we could assert that equality is transitive.

\[
\text{Assert} \ #\forall x,y,z. \quad x = y \wedge y = z \rightarrow x = z#.
\]

The above should be read as “for any \( x,y,z \), if \( x \) is equal to \( y \) and \( y \) is equal to \( z \) then \( x \) is equal to \( z \)”. The \( \wedge \) operator has precedence over the \( \rightarrow \) operator, so no parentheses are necessary. We can now use these operators to construct the inductive case of our proof.

\[
\text{Assert} \ #\forall x. \text{plus x Z = x %-- inductive hypothesis} \rightarrow \text{S (plus x Z) = S x %-- "implies that"}
\]

\[
\quad \rightarrow S (\text{plus x Z}) = S x \quad \%-- apply "S" to both sides
\]

\[
\quad \rightarrow \text{plus (S x) Z = S (plus x Z)} \quad \%-- by definition of "plus"
\]

\[
\quad \rightarrow \text{plus (S x) Z = S x} \quad \%-- by transitivity of equality
\]

\[
\#.
\]

The above can now be summarized for concision in another assertion.

\[
\text{Assert} \ #\forall x. \text{plus x Z = x } \rightarrow \text{plus (S x) Z = S x#.}
\]

We can also restate both cases if we want to do so.

\[
\text{Assert} \ #
\]

\[
\quad \text{plus Z Z = Z}
\]

\[
\quad \wedge \forall x. \text{plus x Z = x } \rightarrow \text{plus (S x) Z = S x}
\]

\[
\#.
\]

Thus, we’ve proven our result by induction over \( x \). Note that the verifier supports “sequenced” equalities:
Assert \(\forall x,y,z. x = y = z \Rightarrow x = z\).

This might sometimes be a more concise way to do a proof. For example, the above proof of the inductive case could look a little shorter if it is written in the following way:

\[
\begin{align*}
\textbf{Assert}
\forall x.
\begin{align*}
\text{plus } x \ Z &= x \\
S \ (\text{plus } x \ Z) &= S \ x
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{plus} \ (S \ x) \ Z &= S \ (\text{plus } x \ Z) = S \ x
\end{align*}
\]

5.2 Classroom Instruction of Linear Algebra

The curriculum course on linear algebra and geometric algorithms introduces undergraduates to vector spaces, linear transformations, and many applications of these concepts. The AARThFACT system was used primarily for verification of algebraic proofs. As an example, we present the following portion of a proof that \(\mathbb{R}^2\) is a vector space given the fact that \(\mathbb{R}\) is a vector space.

First, the definition of addition for vectors in \(\mathbb{R}^2\) is assumed.

Assume for all \(x, y, x', y' \in \mathbb{R}\), \((x, y) + (x', y') = (x + x', y + y')\).

It is then possible to prove that this addition operation is commutative.

\[
\begin{align*}
\text{Assert for any } x, y, x', y' \in \mathbb{R}, \\
(x, y) + (x', y') &= (x + x', y + y'), \\
x + x' &= x' + x, \\
y + y' &= y' + y, \\
(x + x', y + y') &= (x' + x, y' + y), \\
(x' + x, y' + y) &= (x', y') + (x, y), \\
(x, y) + (x', y') &= (x', y') + (x, y).
\end{align*}
\]

A similar proof for associativity of vector addition is also possible.
5.3 Evaluation of the Capabilities of the System

The formal manipulations and proof techniques required for the examples presented in both courses can be split into three categories: application of properties of basic logical operators (conjunction, implication, and universal quantification), application of properties of equality, and algebraic manipulations involving arithmetic operators. The AARTIFACT design provided relatively robust support (at least within the context of classroom instruction) for each kind of formal manipulation.

5.3.1 Logical Operators

The inference rules by which the formal reasoning system operates (and which define its behavior with respect to logical operators) are presented in full in earlier work on the formal reasoning system [Lap09a]. We briefly review one inference rule in particular whose form is essential for the examples we present in this work.

Sequential Conjunction

As described in more detail in earlier work [Lap09a], our formal reasoning system supports a sequential interpretation of the standard conjunction operator. This interpretation is consistent with straightforward formulations of the sequent calculus [Gen69]. The interpretation of conjunction is roughly as follows: if the left-hand argument is determined to be true, it is added to the context under which the right-hand argument is checked. For readers who are interested in a more precise reminder, we reproduce below the introduction rules for the two variants of conjunction. Let \( \Phi \) by the context (containing information about existing assumptions and bound variables).

\[
\frac{\Phi \vdash e_1 \quad \Phi \vdash e_2}{\Phi \vdash e_1 \land e_2}
\]

\[
\frac{\Phi \vdash e_1 \quad \Phi \cup \{e_1\} \vdash e_2}{\Phi \vdash e_1 \land e_2}
\]

The second form of the rule is a consequence of the first when it combined with rules governing implication (the exact derivation can be found in earlier work [Lap09a]). This sequential form of conjunction is essential
to the usability of the interface provided by our formal reasoning system. It allows the user to author arguments consisting of a sequence of equations in which each equation is verified to be a consequence of one or more of the equations found earlier in the sequence. All of the formal arguments presented in this work are assembled in this manner.

5.3.2 Properties of Equality

The formal reasoning system we deployed is designed to simulate a natural context, and one essential characteristic of natural contexts is the presence of a reflexive, symmetric, and transitive equality relation. This relation is simulated using the component of the dynamic context that maintains equivalence classes of expressions.

This feature makes it possible to verify several very common algebraic manipulations involving equations. Any direct manipulation that takes advantage of these properties can be verified.

\[
\text{Introduce } \#x,y,z\#. \\
\text{Assert } \# x + y = x + y \#. \\
\text{Assert } \# ( x = y \land y = z ) \Rightarrow x = z \#. \\
\text{Assert } \# x = y \Rightarrow y = x \#.
\]

Furthermore, some manipulations implied by these properties can also be verified. For example, it is possible to substitute a portion of an equation with an equivalent expression. In the example below, the asserted expression corresponds to the first assumption except that \((y + z)\) is substituted with \((v + w)\) and \(c\) is substituted with \(u\).

\[
\text{Introduce } \#x,y,z,a,b,c,u,v,w\#. \\
\text{Assume } \# x + (y + z) = (a + b) + c\#. \\
\text{Assume } \# y + z = v + w \#. \\
\text{Assume } \# c = u \#. \\
\text{Assert } \# x + (v + w) = (a + b) + u \#.
\]

It is also possible to perform the same operation to both sides of an equation. In the example below, we add \(z\) to both sides of an equation.

\[
\text{Introduce } \#x,y,z,a,b\#. \\
\text{Assume } \# x + y = a + b \#. \\
\text{Assert } \# x + (x + y) = z + (a + b) \#.
\]

However, some intuitive manipulations are not yet possible with the current feature set of the formal reasoning system, and we mention these below in Section 5.3.4.

5.3.3 Algebraic Manipulations with Arithmetic Operators

The formal reasoning system we utilized is also capable of modelling algebraic manipulations involving the arithmetic operators corresponding to addition, substraction, multiplication, and division. In particular, the system can recognize applications of the associativity, commutativity, and identity laws of addition and multiplication, and the distributive law between addition and multiplication when the expression being distributed is a constant. This capability is described briefly in work describing the formal reasoning system [Lap09a]. The examples in Sections 5.1.2 and 5.1.2 demonstrate how this capability allows users of the formal reasoning system to avoid explicitly stating (or retrieving from a library) a few of these common algebraic laws.
5.3. **EVALUATION OF THE CAPABILITIES OF THE SYSTEM**

5.3.4 **Improvements**

There were two noteworthy instances in which the intuitive expectations of users were not met by the formal reasoning system we deployed. This suggests that the user context that most students possess has these capabilities, and thus, any system oriented towards simulating a user context should also have these capabilities.

**Composition of Two Kinds of Manipulation**

The first is application of the symmetry of equality under a universal quantifier. Students expected (and rightly so) that proof scripts of the following form would be automatically verifiable:

\[
\begin{align*}
\text{Assume } & \forall \bar{x}, e_1(\bar{x}) = e_2(\bar{x}) \\
\text{Assert } & \forall \bar{y}, e_2(\bar{y}) = e_1(\bar{y})
\end{align*}
\]

This demonstrates that when querying their own context, students are able to compose elimination of the universal quantifier with application of the symmetry of equality. It is necessary that a system that simulates a user context be capable of inferring that this step has been performed. This issue has been addressed by introducing additional propositions into the natural context.

**Implicit Application of Induction Principles**

The second instance was more substantial: the lightweight proof that the students were asked to construct did not allow the students to use the statement being proven in subsequent proofs. This was not necessary for the assignment, but it does illustrate a difference between the use of lightweight verification system by domain experts, and the use of such systems by students. An expert could decide whether he is satisfied with a lightweight proof, and introduce the theorem proven as an assumption. On the other hand, students must be given strict guidelines in this situation because they may not have the experience to make these decisions.

While we avoided this issue by assembling an assignment in which reuse of the proven statements was not needed, we still failed to meet the intuitive expectations of the students. One way to resolve this is to ask the students to write a more complete proof within our system (involving the explicit introduction and application of propositions corresponding to the induction principles over each of the data types involved) so that the proven statement can be used in subsequent arguments. However, we believe the approach that is more consistent with our general principles is to eventually introduce automated support for this capability. The students should indeed be able to use the statements proven, but should not need to do more work than was required of them in our version of the assignment.

**On Lightweight Formal Verification and Limiting Formal Reasoning Systems**

Some features of the formal reasoning system we deployed were never used in any examples or assigned problems, including those supporting manipulations involving disjunction and existential quantification. All of the examples presented in this report can be verified by a formal reasoning system that does not support manipulations involving disjunction and existential quantifiers. We accomplished this by requiring only lightweight verification of arguments, and by selecting only those examples that involve the proof of a simple universal proposition. We did not explicitly state or require the use of any induction principles within the exercises.

This approach had several advantages. Students were required to comprehend fewer formal concepts, and could thus use their time and effort to study the basic properties of implication, conjunction, and universal quantification (as well as the properties of equality). It also put less pressure on the verification system: providing support for a variety of manipulations involving the introduction and elimination of existential quantifiers (as well as disjunction) is a difficult design task. Users did not need to deal with any inadequacies the system might have in this regard. This demonstrates that while a limited verification system may not seem interesting from the perspective of an expert logician, it can seem quite reasonable to a domain expert.
unfamiliar with more general forms of logic. Our experience suggests that it is possible to have a limited but successful formal reasoning system that supports only a subset of the inference rules governing common logical operators.

5.4 Pedagogical and Administrative Consequences

We briefly comment on how the design of the formal reasoning system affected administrative and practical issues from various perspectives.

5.4.1 Instruction

We observed a few pedagogical advantages to using our formal reasoning system within classroom instruction. The presence of an application that provides direct feedback about formal arguments makes it possible to easily present and reinforce a precise list of valid formal manipulations. The use of a parser combinator library in the implementation of the formal reasoning system’s parser (as discussed briefly in previous work [Lap09a]) makes it possible to support multiple parsing regimes, including one that corresponds almost exactly to the syntax of the particular programming language in which the formally analyzed code is written. Admittedly, the parser combinator library that we used [LM01] was designed with Haskell in mind, but we believe its flexibility would make it similarly easy to adjust the system for formal reasoning exercises involving other functional programming languages.

5.4.2 Usability from the Perspective of Students

One frequent concern about any sort of automated search or inference done by a formal reasoning system is that the user will become frustrated because she cannot predict what the system might be inferring, or what the system’s limitations are. However, within this deployment we have found that no such frustration occurs when the system’s capabilities are intuitive and can be described in a succinct and straightforward manner. The syntax of the AARTIFACT system did not present much difficulty to the students in either course. For the functional programming course, the mathematical syntax of AARTIFACT was augmented with typical Haskell operators and looked very similar to Haskell syntax. For the course on linear algebra, the syntax used was a subset of \( \LaTeX \) and was natural enough that many students were able to utilize by consulting nothing more than an example of an argument. It is also worth noting that some students used the AARTIFACT syntax in writing their pencil and paper exam solutions (without access to the verifier). It is debatable what this might indicate about their understanding of the formal reasoning techniques they employ, but it does demonstrate that the syntax is not too cumbersome to be used manually (most likely because it corresponds so closely to the kind of syntax humans already developed).

The formal manipulations enabled by the simulated contexts and the sequential variant of the conjunction inference rule were understood by many students without any explicit guidance. This indicates that the semantics of the system corresponded well to the existing contexts of the students who had already been introduced to the mathematical conventions governing the concepts involved.

5.4.3 Grading

It is important to observe that despite the use of a lightweight approach, the task of grading was simplified substantially. When reviewing a student’s solutions to the assigned problems, the grader only needed to verify manually that the student’s formal argument had the appropriate superficial form. In our particular examples, this amounted to confirming that the student’s solutions did not introduce any assumptions beyond those which we provided, that the asserted arguments used only sequential conjunction (rather than, for example, implication), and that assertions corresponding to both the base case and the inductive case of an argument were present. The grader was not required to spend any effort on the more daunting and intellectually taxing task of actually verifying that each formal argument (which could vary from one student to the next) consisted of a sequence of valid manipulations.
Acknowledgements

We would like to thank for their cooperation and feedback Professor Assaf Kfoury (the instructor of both courses), Michael Goldberg (the grader for both courses), and the students enrolled in both courses.
Chapter 6

Safe Compositional Network Sketches

Many large-scale, safety-critical systems can be viewed at least in part as constrained-flow networks (CFNs). That is, they can be viewed as interconnections of subsystems, or modules, each of which is a producer, consumer, or regulator of flows. These flows are characterized by a set of variables and a set of constraints thereof, reflecting inherent or assumed properties or rules governing how the modules operate (and what constitutes safe operation). Our notion of flow encompasses streams of physical entities (e.g., vehicles on a road, fluid in a pipe), data objects (e.g., sensor network packets or video frames), or consumable resources (e.g., electric energy or compute cycles).

Current practices in the assembly of such systems involve the integration of various subsystems into a whole by “system integrators” who may not possess expertise or knowledge of the internals of the subsystems on which they rely. NetSketch [BKLO09b] is a tool for the specification of constrained-flow networks and the certification of desirable safety properties imposed on such networks. NetSketch assists system integrators in two types of activities: modelling and design. As a modelling tool, NetSketch enables the abstraction of an existing (flow network) system while retaining sufficient information about it to carry out future analysis of safety properties. As a design tool, NetSketch enables the exploration of alternative safe designs as well as the identification of minimal requirements for missing subsystems in partial designs. Formal analysis is at the heart of both of the above modelling and design activities. In earlier work we overview NetSketch, highlight its salient features, illustrate how a prototype tool could be used in actual applications [BKLO09b] and provide a formal definition of its underlying formalism [BKLO10].

6.1 Related Work

The work described in this chapter represents an effort to apply to novel research a formal reasoning system that is designed according to principles that emphasize usability and lightweight verification approaches. However, the DSL and underlying formalism, modelled using a formal reasoning system in this report, is itself a language designed according to such principles. It differs most significantly in that it is intended for use in specific domains that can be modelled as constrained-flow networks. As a result, there are two distinct bodies of related work.

6.1.1 Usability and Application of Formal Reasoning Systems

In this chapter we illustrates the usefulness of several characteristics of a formal reasoning system that is designed to simulate a natural context. The interface of the system is familiar, straightforward, and requires no explicit reference of facts as they are applied within a formal argument. These features are recognized as important within several efforts and projects that have similar goals [ACP01, Bro06, VLP08, MRS01, SBF’02, SC05]. Furthermore, the system takes advantage of an extensive library of definitions and propositions dealing with common mathematical concepts and provides native support for some of these
concepts. This is inspired by work in a subdiscipline of artificial intelligence that deals with the assembly and application of ontologies. Particular examples include the Cyc Project [PML+06] and Open Mind Common Sense [Chk05, CG05, SBL04, LS04a, LS04b].

There exist few examples of applications of lightweight formal reasoning systems within novel research. Some examples include applications within cryptography [BJP02, BPW03], security in computation [BHM08, BMU08, BS03, BJ03], and economic mechanism design [TG07].

6.1.2 Formalisms for Modelling Constrained-flow Networks

Our formalism for reasoning about constrained-flow networks was inspired by and based upon formalisms for reasoning about programs developed over the decades within the programming languages community. While our work focuses in particular on networks and constraints on flows, there is much relevant work in the community addressing the general problem of reasoning about distributed programs. However, most previously proposed systems for reasoning in general about the behavior of distributed programs (Process algebra [BW90], Petri nets [Pet66], II-calculus [MPW92], finite-state models [LT89, LV95, LV96], and model checking [Hol97, HS99]) rely upon the retention of details about the internals of a system’s components in assessing their interactions with one another. While this affords these systems great expressive power, that expressiveness necessarily carries with it a burden of complexity. Such an approach is inherently not modular in its analysis: the details maintained in a representation or model of a component are not easily introduced or removed. Furthermore, these specifications are often wedded to particular methodologies and thus do not have the generality necessary to allow multiple kinds of analysis, making it difficult to reason about how systems specified using different methodologies interact.

Discovering and enforcing bounds on execution of program fragments is a well-established problem in computing [WEE+08], and our notion of types (i.e. linear constraints) for networks can be viewed as a generalization of type systems expressing upper bounds on program execution times. Existing work on this problem includes the aiT tool (described in [TFW00], and elsewhere), which uses control-flow analysis and abstract interpretation to provide static analysis capabilities for determining worst and best case execution time bounds. Other works, belonging to what have been called Dependent Type Systems, provide capabilities for estimating an upper bound on execution time and memory requirements via a formal type system that has been annotated with size bounds on data types. These include (but are not limited to) Static Dependent Costs [RG94], Sized Type Systems [HPS96], and Sized Time Systems [LH96]. Many other Dependent Type Systems directly target resource bounding for the real-time embedded community (e.g. the current incarnation of the Sized Time System [IFH06], Mobile Resource Guarantees for Smart Devices [AGH+05]).

More generally, there has been a large interest in applying custom type systems to domain specific languages (which peaked in the late nineties, e.g. the USENIX Conference on Domain-Specific Languages (DSL) in 1997 and 1999). Later type systems have been used to bound other resources such as expected heap space usage (e.g. [HJ03], [AGH+05]). The support for constructing, modelling, inferring, and visualizing networks and properties of network constraints provided by our work is similar to the capabilities provided by modelling and checking tools such as Alloy [Jac02].

One of the essential activities our formalism aims to support is reasoning about and finding solution ranges for sets of constraints that describe properties of a network. In its most general form, this is known as the constraint satisfaction problem [Tsa93] and is widely studied [Tsa99]. One variant of the constraint satisfaction problem relevant to our work involves only linear constraints. Finding solutions respecting collections of linear constraints is a classic problem that has been considered in a large variety of work over the decades. There exists a great deal of established material [Sch98], including many documented algorithms [CLR90, Ch. 29], and many analyses of practical considerations [Fle87]. However, most approaches consider a homogenous set of constraints of a particular class.

The work in this paper extends and generalizes our earlier work in Traffic (Typed Representation and Analysis of Flows For Interoperability Checks [BBKM05]), and complements our earlier work in Chain (Canonical Homomorphic Abstraction of Infinite Network protocol compositions [BBK03]).

While our formalism supports the specification and verification of desirable global properties and has a rigorous foundation, it remains ultimately lightweight. By “lightweight” we mean to contrast our work to the
heavy-going formal approaches – accessible to a narrow community of experts – which are permeating much of current research on formal methods and the foundations of programming languages (such as the work on automated proof assistants [Pau94, Her94, Cia03, CS03], or the work on polymorphic and higher-order type systems [TLC07], or the work on calculi for distributing computing [Bou97]). In doing so, our goal is to ensure that the constructions presented to users are the minimum that they might need to accomplish their task.

6.2 Soundness of the NetSketch Formalism

6.2.1 Modules: Untyped and Typed

We introduce several preliminary notions. For more detailed definitions, we refer readers to the full technical report on the formalism [BKLO09a].

Definition 6.2.1. Syntax of Constraints

Let \( \mathcal{X} = \{x_0, x_1, x_2, \ldots \} \) be an infinite set of parameters. The set of constraints over \( \mathbb{N} \) and \( \mathcal{X} \) can be defined in extended BNF style, where we use metavariables \( n \) and \( x \) to range over \( \mathbb{N} \) and \( \mathcal{X} \), respectively:

\[
\begin{align*}
  e \in \text{Exp} & \quad ::= \quad n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid \ldots \\
  c \in \text{Const} & \quad ::= \quad e_1 = e_2 \mid e_1 < e_2 \mid e_1 \leq e_2 \mid \ldots 
\end{align*}
\]

We include in \( \text{Const} \) at least equalities and orderings of expressions. Possible extensions of \( \text{Const} \) include conditional constraints, negated constraints, time-dependent constraints, and others. Within our formalism, constraints in \( \text{Const} \) are part of a given flow network abstraction and may be arbitrarily complex; constraints to be inferred or checked against the given constraints are from some restriction of \( \text{Const} \), such as linear constraints:

\[
\begin{align*}
  e \in \text{LinExp} & \quad ::= \quad n \mid x \mid n \times x \mid e_1 + e_2 \\
  c \in \text{LinConst} & \quad ::= \quad e_1 = e_2 \mid e_1 < e_2 \mid e_1 \leq e_2 
\end{align*}
\]

We are adopting a lightweight approach in our machine-assisted verification of the formalism, so we do not explicitly model the structure of the set of constraints. Instead, we introduce abstract constants and predicates related to constraint sets.

<table>
<thead>
<tr>
<th>Introduce the constant ( \text{Const} ). Assume for any ( C ), ( C ) is a constraint set iff ( C \subseteq \text{Const} ). Assume for any ( C ), if ( C ) is a constraint set then ( C ) is a set.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduce the constant ( \mathcal{X} ). Assume for any ( P ), ( P ) is a parameter set iff ( P \subseteq \mathcal{X} ). Assume for any ( P ), if ( P ) is a parameter set then ( P ) is a set.</td>
</tr>
<tr>
<td>Introduce the constant ( \text{parameters} ). Assume for any ( C ), if ( C \subseteq \text{Const} ) then ( \text{parameters}(C) \subseteq \mathcal{X} ).</td>
</tr>
</tbody>
</table>

Definition 6.2.2. Untyped Modules

A module is a network component that is amenable to exhaustive analysis. We specify an untyped module \( A \) by a four-tuple \((A, I, O, C)\).
Assume for any \( A, I, O, C \),
\[(A, I, O, C) \text{ is an untyped module} \]
iff
\[
I \subseteq \text{parameters}(C),
O \subseteq \text{parameters}(C),
O \cap I = \emptyset, \text{ and}
C \subseteq \text{CONST}.
\]

**Definition 6.2.3.** Typed Modules

For a module \((A, I, O, C)\) as specified in Definition 6.2.2, a typed specification or typing for \((A, I, O, C)\) is the untyped module specification paired with a constraint set \(C'\) from some restricted space of constraints (as described in Definition 6.2.1).

Assume for any \( A, I, O, C, C' \),
\[(A, I, O, C) : C' \text{ is a typed module} \]
iff
\[
(A, I, O, C) \text{ is an untyped module,}
C' \subseteq \text{CONST}, \text{ and}
\text{parameters}(C') \subseteq I \cup O.
\]

A typing judgment \((A, I, O, C) : C'\) may or may not be valid. The validity of judgments presumes a formal definition of the semantics of modules, which we introduce in Section 6.2.2.

### 6.2.2 Semantics of Network Typings

A network typing, as defined precisely in Section 6.2.3 further below, is specified by an expression of the form \((M, I, O, C) : C\) where \((M, I, O, C)\) is an untyped network and \(C\) is a finite set of constraints such that \(\text{parameters}(C) \subseteq I \cup O\). In this section, we introduce the definition and semantics of constraints.

**Definition 6.2.4.** Satisfaction of Constraints Let \(Y \subseteq \mathcal{X}\) be a subset of the parameter space. Let \(V\) be a valuation for \(Y\), i.e., \(V\) is a map from \(Y\) to \(\mathbb{N}\). Suppose all expressions and constraints are written over parameters in \(Y\). We use \(\models\) to denote the satisfaction relation.

The interpretation of an expression relative to \(V\) is defined by induction over \(e \in \text{Exp}\):

\[
V(e) = \begin{cases} 
  n & \text{if } e = n, \\
  V(x) & \text{if } e = x \in Y, \\
  p & \text{if } e = e_1 \ast e_2 \& p = V(e_1) \ast V(e_2), \\
  q & \text{if } e = e_1 + e_2 \& q = V(e_1) + V(e_2), \\
  r & \text{if } e = e_1 - e_2 \& r = V(e_1) - V(e_2),
\end{cases}
\]
6.2. SOUNDNESS OF THE NETSKETCH FORMALISM

Satisfaction of a constraint by \( V \) is defined by cases over \( c \in \text{Const} \):

\[
\begin{align*}
V \models e_1 = e_2 & \quad \text{iff} \quad V(e_1) = V(e_2) \\
V \models e_1 < e_2 & \quad \text{iff} \quad V(e_1) < V(e_2) \\
V \models e_1 \leq e_2 & \quad \text{iff} \quad V(e_1) \leq V(e_2)
\end{align*}
\]

Satisfaction of a set of constraints relative to \( V \) is defined in the natural way:

\[
V \models \{ c_1, \ldots, c_p \} \quad \text{iff} \quad V \models c_1 \quad \text{and} \quad \ldots \quad \text{and} \quad V \models c_p
\]

We introduce several useful operators involving parameters, constraints, and network typings.

**Definition 6.2.5. Comparison of Constraint Sets**

We define a precise way of relating constraint sets to one another in terms of constraint semantics.

Introduce the set operators \( \Rightarrow, \Leftrightarrow \).

Assume for any \( C, C' \),
if \( C \subseteq \text{Const}, C' \subseteq \text{Const} \) then
\[
C \Rightarrow C'
\]
iff
for all \( V \in X \rightarrow \mathbb{N} \), \( V \models C \) implies \( V \models C' \).

Assume for any \( C, C', C \Leftrightarrow C' \) if \( C \Rightarrow C' \) and \( C' \Rightarrow C \).

Naturally, any set implies all subsets of itself.

Assume for any \( C, C', C' \subseteq C \) implies \( C \Rightarrow C' \).

**Definition 6.2.6. Closures of Constraint Sets**

For a finite constraint set \( C \), its closure is the set of all constraints implied by \( C \).

Introduce the constant closure. Assume for any \( C \), closure\((C) = \{ c | c \in \text{Const}, C \Rightarrow \{ c \} \} \).
Assume for any \( C \), if \( C \subseteq \text{Const} \) then closure\((C) \subseteq \text{Const} \).

Naturally, a set is a subset of its own closure, and the closure of a set contains the closure of each of its subsets.

Assume for any \( C, C \subseteq \text{closure}(C) \).
Assume for any \( C, C', C \subseteq C' \) implies closure\((C) \subseteq \text{closure}(C') \).

If the parameters of two constraint sets are disjoint, then the closure operation preserves the union operation on these two constraint sets.

Assume for all \( C, C' \),
if parameters\((C) \cap \text{parameters}(C') = \emptyset \) then closure\((C \cup C') = \text{closure}(C) \cup \text{closure}(C') \).
Furthermore, the parameters of the closures of any such two constraint sets are also disjoint.

\[
\text{Assume for any } C, C', \text{ parameters}(C) \cap \text{parameters}(C') = \emptyset
\]
implies \[
\text{parameters(closure}(C)) \cap \text{parameters(closure}(C')) = \emptyset.
\]

**Definition 6.2.7. Restrictions on Constraint Sets**

In subsequent definitions, we require the ability to restrict a constraint set using particular parameter sets.

Introduce set operators \(\upharpoonright, \downharpoonright\).

Assume for any \(C, P\), \(C \subseteq \text{Const}\) and \(P \subseteq \mathcal{X}\)
implies that
\[
C \upharpoonright P = \{c | c \in C, \text{parameters}(c) \subseteq P\},
\]
\[
C \downharpoonright P = \{c | c \in C, \text{parameters}(c) \cap P \neq \emptyset\}.
\]

Thus, \(C \upharpoonright P\) is the subset of \(C\) in which only parameters from \(P\) occur, and \(C \downharpoonright P\) is the subset of \(C\) in which every constraint has an occurrence of a parameter from \(P\). We present some facts about these operators.

Assume for any \(C, P, C \upharpoonright P \subseteq C\).
Assume for any \(C, P, C \downharpoonright P \subseteq C\).
Assume for any \(C, C', S, S', \) if \(C \subseteq C'\) and \(S \subseteq S'\) then \(C' \upharpoonright S' \Rightarrow C \upharpoonright S\).
Assume for any \(C, C', S, S', \) if \(C \subseteq C'\) and \(S \subseteq S'\) then \(C' \downharpoonright S' \Rightarrow C \downharpoonright S\).
Assume for any \(C, C', P, P',\)
\[
\text{parameters}(C) \cap \text{parameters}(C') = \emptyset
\]
implies that
\[
(C \upharpoonright P) \cup (C' \upharpoonright P') = (C \cup C') \upharpoonright (P \cup P').
\]

**Definition 6.2.8. Pre- and Post-conditions for Network Sketches**

Let \((M, I, O, C) : C\) be a typed network sketch. Recall that \(\text{parameters}(C) \subseteq I \cup O\). We partition closure\((C)\) into two subsets.

Introduce the constants \(\text{pre, post}\). Assume for any \(M, I, O, C, C,\)
\[
\text{pre}((M, I, O, C), C) = \text{closure}(C) \upharpoonright I,
\]
\[
\text{post}((M, I, O, C), C) = \text{closure}(C) \downharpoonright O = \text{closure}(C) - \text{pre}((M, I, O, C), C).
\]

Note that while the parameters of \(\text{pre}(C)\) are all in \(I\), the parameters of \(\text{post}(C)\) are not necessarily all in \(O\), because some constraints in \(C\) may contain both input and output parameters. For both operators, the constraint set \(C\) for any network sketch implies the constraints in the \(\text{pre}\) and \(\text{post}\) constraint sets.

Assume for all \(M, I, O, C, C \Rightarrow \text{pre}((M, I, O, C), C)\).
Assume for all \(M, I, O, C, C \Rightarrow \text{post}((M, I, O, C), C)\).
6.2. SOUNDNESS OF THE NETSKETCH FORMALISM

In the full report describing our formalism [BKLO09a], we introduce two different semantics, corresponding to what we call “weak satisfaction” and “strong satisfaction” of typing judgements. Both semantics are meaningful, corresponding to whether or not network nodes act as “autonomous systems”, i.e., whether or not each node coordinates its action with its neighbors or according to instructions from a network administrator. The definitions of “weak satisfaction” and “strong satisfaction” are very similar except that the first involves an existential quantification and the second a universal quantification. In this report, we introduce only the strong variant of satisfaction.

**Definition 6.2.9. Satisfaction**

Let \( V : I \rightarrow \mathbb{N} \) be a fixed valuation of the input parameters of a network \((M, I, O, C)\).

\[
\begin{align*}
\text{Assume for any } M, I, O, C, C', \text{ for any } V,
V \models ((M, I, O, C) : C) \\
\text{iff}
V \models \text{pre}((M, I, O, C), C) \text{ implies that}
\text{for all } V',
V \subseteq V' \text{ and for all } K \in C, V' \models K
\text{ implies that}
V' \models \text{post}((M, I, O, C), C).
\end{align*}
\]

There are several useful facts about the \( \models \) relation.

\[
\begin{align*}
\text{Assume for any } V, V', C, \text{ if } V \subseteq V' \text{ and } V \models C \text{ then } V' \models C.
\text{Assume for any } V, C, C', \text{ if } C \Rightarrow C' \text{ and } V \models C \text{ then } V \models C'.
\text{Assume for any } V, C, C', \text{ if } V \models C \text{ and } V \models C' \text{ iff } V \models C \cup C'.
\end{align*}
\]

**Definition 6.2.10. Validity of Typings**

Informally, a typing is strongly valid iff, for every network flow satisfying \( \text{pre}((M, I, O, C), C) \), and for every way of channelling the flow through \((M, I, O, C)\) that is consistent with its internal constraints \( C \), \( \text{post}((M, I, O, C), C) \) is satisfied.

\[
\begin{align*}
\text{Assume for any } M, I, O, C, C,
((M, I, O, C) : C) \text{ is strongly valid}
\text{iff}
\text{for all } V \in I \rightarrow \mathbb{N},
V \models ((M, I, O, C) : C).
\end{align*}
\]

### 6.2.3 Typed Network Sketches

We define a specification language to assemble modules together, also allowing for the presence of network holes. This is a strongly-typed domain-specific language (DSL). For the sake of concision, we present only the typed version of the DSL and refer readers interested in seeing separate presentations of both the untyped and typed DSL to the full technical report presenting our formalism [BKLO09a].

A network sketch is written as \((M, I, O, C)\), where \( I \) and \( O \) are the sets of input and output parameters, and \( C \) is a finite set of finite constraint sets. \( M \) is not a name but an expression built up from: (1) module names and (2) the constructors \texttt{conn}, \texttt{loop}, and \texttt{hole}. A “network hole” can be viewed as a place-holders...
with some associated attributes. In this presentation, each network hole \( \text{hole}(X, \{M_1, \ldots, M_n\}) \) explicitly contains a label \( X \) and a set of network sketches \( \{M_1, \ldots, M_n\} \). Any of the network sketches in \( \{M_1, \ldots, M_n\} \) can be interchangeably “placed” into this hole, depending on changing conditions of operation in the network as a whole. This treatment of network holes differs from the treatment in the technical report describing the formalism [BKLO09a], but only superficially.\(^1\)

**Definition 6.2.11. Syntax of Raw Network Sketches**

We introduce the raw syntax in extended BNF style. The formal expressions written according to the following BNF are said to be “raw” because they do not specify how the internal constraints of a network sketch are assembled together from those of its subcomponents. This is what the rules in Section 6.2.3 do precisely.

---

Introduce the constants \( \text{conn}, \text{loop}, \text{hole} \).

\[
\begin{align*}
A, B & \in \text{ModuleNames} \\
X, Y, Z & \in \text{HoleNames} \\
\theta & \in \mathcal{X} \to \mathcal{X} \\
M, N & \in \text{RawSketches} := \\
& A \mid \text{conn}(\theta, M, N) \\
& \mid \text{loop}(\theta, M) \\
& \mid \text{hole}(X, \{M_1, \ldots, M_n\})
\end{align*}
\]

As a convention, we use upper-case letters to refer to modules and networks – from the early alphabet (\( A, B \)) for modules and from the middle alphabet (\( M, N \)) for networks. Also note carefully that \( M \) and \( N \) are metavariables, ranging over expressions in \( \text{RawSketches} \); they do not appear as formal symbols in such expressions written in full. By contrast, \( A \) and \( B \) are names of modules and can occur as formal symbols in expressions of \( \text{RawSketches} \). We assume that each occurrence of the same module or the same hole in a raw sketch has its own private set of names. This invariant can be ensured using isomorphic renaming. For a slightly more detailed discussion of how this can be accomplished, we refer readers to the full technical report presenting the formalism [BKLO09a].

**Type Inference Rules**

When networks are connected (i.e. composed) with themselves or one another, maps are introduced that relate the parameters that represent the output links of a network to parameters that represent the input links of a network. By convention, these maps are represented using lowercase Greek letters (\( \theta, \phi, \psi \)).

We also introduce a function \( \text{constraints} \) that transforms a map \( \theta \) into a corresponding set of constraints \( \{x = \theta(x) | x \in \text{dom}(x)\} \). Because we are adopting a lightweight verification approach in which we do not explicitly define the space \( \text{Const} \), we do not explicitly define \( \text{constraints} \).

---

\(^1\)In the full report’s treatment [BKLO09a], a \texttt{let}-bound, labelled hole can only appear *once* within its scope. Our presentation “collapses” the \texttt{let} syntax within network sketches in the other report by placing directly within a hole’s syntax the set of network sketches bound to that hole. It also becomes necessary to assume that all hole labels are unique, but this can easily achieved through renaming. This change in the formalism presentation is motivated by a desire to simplify the formal definitions and arguments by removing the need to maintain an explicit environment.
6.2. SOUNDNESS OF THE NETSKETCH FORMALISM

Each network \((M, I, O, C)\) has a collection of constraint sets \(C\) as a component, and assembling multiple modules using some of these rules requires that a complex operation on these collections be performed. We summarize these operations in the definitions presented below. We refer readers to the full report describing our formalism [BKLO09a] for a more detailed discussion of the reasoning behind these operations.

Introduce the constants \(cn, lp, hl\).

Assume for any \(C, C', \theta\),
\[
\text{cn}(\theta, C, C') = \{C \cup C' \cup \text{constraints}(\theta) | C \in C, C' \in C'\}.
\]

Assume for any \(C, \theta\),
\[
\text{lp}(\theta, C) = \{C \cup \text{constraints}(\theta) | C \in C\}.
\]

Assume for any \(n, I, O, I', O', C\),
\[
\text{hl}(n, I, O, I', O', C) = \{\text{constraints}(\phi) \cup \text{constraints}(\psi) | C_i \cup \text{constraints}(\phi) \cup \text{constraints}(\psi) \cup I_i \rightarrow I_i, \psi \in O_i \rightarrow O', C \in C\}.
\]

We introduce a few properties of these operations with respect to satisfaction.

Assume for any \(V, C, \theta\),
\[
\text{for all } K \in \text{lp}(\theta, C), V \models K
\]
implies that \(\text{for all } K' \in C, V \models K'\).

Assume for any \(V, C, C', \theta\),
\[
\text{for all } K \in \text{cn}(\theta, C, C'), V \models K
\]
implies that \(\text{for all } K' \in C, V \models K'\) and \(\text{for all } K'' \in C', V \models K''\).

Our inference rules are a means by which to inductively construct valid typing judgments. We indicate that a typing judgment is valid by prefixing it with the \(\vdash\) (turnstile) symbol.

Introduce the constant \(\vdash\).

There are 5 inference rules: [Module], [Connect], [Loop], [Hole], and [Weaken]. We begin with the base case, the inference rule [Module].

Assume for any \(A, I, O, C, C', C''\),
\[
(A, I, O, C) \vdash C' \text{ is a typed module, and } C \Rightarrow C'
\]
implies that \(\vdash (A, I, O, \{C\}) : C'\).

The rule [Connect] takes two network sketches, \(M\) and \(N\), and returns a network sketch \(\text{conn}(\theta, M, N)\).
in which some of the output parameters in $M$ are identified with some of the input parameters in $N$ according to what $\theta$ prescribes. We present the inference rule $\text{[Connect]}$.

Assume for any $M, I, O, C, C, N, I', O', C', C', \theta$, for all $I'', O'', C''$,
\[
\vdash (M, I, O, C) : C,
\vdash (N, I', O', C') : C',
\theta \in O \rightarrow I', \theta \text{ is injective,}
I'' = I \cup (I' - \text{ran}(\theta)),
O'' = O' \cup (O - \text{dom}(\theta)),
C'' = C' \downarrow I'' \cup O'', \text{ and}
\text{post}((M, I, O, C), C) \Rightarrow \text{pre}((N, I', O', C'), C') \downarrow \text{ran}(\theta)
\]
implies that
\[
\vdash (\text{conn}(\theta, M, N), I'', O'', \text{cn}(\theta, C, C')) : C''.
\]

Rule $\text{[Loop]}$ takes one network sketch, $M$, and returns a new network sketch $\text{loop}(\theta, M)$ in which some of the output parameters in $M$ are identified with some of the input parameters in $M$ according to $\theta$. We present the inference rule $\text{[Loop]}$.

Assume for any $M, I, O, C, C, \theta$, for all $I', O', C'$,
\[
\vdash (M, I, O, C) : C,
\theta \in O \rightarrow I, \theta \text{ is injective, and}
I' = I - \text{ran}(\theta),
O' = O - \text{dom}(\theta),
C' = C \downarrow I' \cup O', \text{ and}
\text{pre}((\text{loop}(\theta, M), I', O', \text{lp}(\theta, C)), C') \Rightarrow \text{pre}((M, I, O, C), C)
\]
implies that
\[
\vdash (\text{loop}(\theta, M), I', O', \text{lp}(\theta, C)) : C'.
\]

The rule $\text{[Hole]}$ is a little more complicated than the preceding rules. Each of the networks $M_i$ that can be placed into the hole must be typed, and the collection of constraint sets governing the hole (as defined by $\text{hl}$ further above) must take into consideration, for every network $M_i$, every possible permutation of the domains of the maps $\phi$ and $\psi$ that connect $M_i$ to the parameters of the hole. Furthermore, the hole’s type must be equivalent to the type of each of the networks $M_i$ under each possible permutation of the connecting maps. Finally, the overall type $C'$ of the hole itself must make it possible to satisfy the definition of validity, so the side condition $\text{pre}((M', I', O', C'), C') \Rightarrow \text{post}((M', I', O', C'), C')$ is stipulated. We present the inference rule $\text{[Hole]}$.

Assume for any $n, X, M, I, O, C, C$, for all $M', I', O', C', C'$,
for all $i \in \{0, \ldots, n\}$,
\[
\vdash (M_i, I_i, O_i, C_i) : C_i,
\text{for all } \phi \in I' \rightarrow I_i, \psi \in O_i \rightarrow O',
C' \cup \text{constraints}(\phi) \cup \text{constraints}(\psi) \iff C_i,
M' = \text{hole}(X, \{M_i \mid i \in \{0, \ldots, n\}\}),
C' = \text{hl}(n, I, O, I', O', C),
\text{pre}((M', I', O', C'), C') \Rightarrow \text{post}((M', I', O', C'), C')
\]
implies that
\[
\vdash (M', I', O', C') : C'.
\]
6.2. SOUNDNESS OF THE NETSKETCH FORMALISM

In the premises of each of the rules, we introduced crucial side conditions expressing a relationship that must be satisfied by the derived types. These include any premises which are not typing judgments. Some of these may appear restrictive at first, but the rule \([\text{Weaken}]\) allows for the adjustment of derived types and constraints (in particular, their weakening) in order to satisfy the side conditions. We present the inference rule \([\text{Weaken}]\).

\[
\begin{align*}
\text{Assume for any } M, I, O, C, C', \\
\vdash (M, I, O, C) : C, \\
\text{pre}((M, I, O, C), C') \Rightarrow \text{pre}((M, I, O, C), C), \text{ and} \\
\text{post}((M, I, O, C), C) \Rightarrow \text{post}((M, I, O, C), C') \\
\text{implies that} \\
\vdash (M, I, O, C) : C'.
\end{align*}
\]

6.2.4 Soundness

The inference rules for typed network sketches presented in Section 6.2.3 are sound with respect to both strong and weak versions of validity. We present a machine-verifiable argument that the inference rules are sound with respect to the strong version of validity, but the proof for the weak version is almost identical. In the full technical report describing this formalism [BKLO09a], we note within the proof any differences that arise between the proofs for the two kinds of validity.

Supporting Lemmas

The proof requires a few simple lemmas involving the operators introduced and defined in previous sections. Because we are only interested in performing a lightweight automated verification of our proof, and because we are fairly confident that these simple lemmas are valid, we do not provide proofs for these lemmas. Our confidence in the correctness of our proof of soundness can be improved further by providing machine-verifiable proofs for these lemmas.

\[
\begin{align*}
\text{Assume for any } C, P, P', \text{ if } P \subseteq P' \text{ then } & C \upharpoonright P = C \upharpoonright P. \\
\text{Assume for any } C, P, P', \text{ if } P \subseteq P' \text{ then } & \text{closure}(C \upharpoonright P') \upharpoonright P = \text{closure}(C) \upharpoonright P. \\
\text{Assume for any } V, C, P, P', \text{ if } & V \models C \upharpoonright P \text{ and } V \models C \upharpoonright P' \text{ then } & V \models C \upharpoonright P' \cup P.
\end{align*}
\]

Proof of Soundness

The claim is stated formally in Theorem 6.2.12. The theorem is proven by an inductive argument for which there exists one base case.

**Theorem 6.2.12 (Soundness).** If \(\vdash (N, I, O, C) : C\) can be derived by the inference rules then for any \(V, V \models (N, I, O, C) : C\).

**Proof.** The theorem holds by induction over the structure of the derivation \(\vdash (N, I, O, C) : C\). Proposition 6.2.13 is the base case, and Propositions 6.2.14, 6.2.15, 6.2.16, and 6.2.17 cover the four possible inductive cases.

**Base Case**

Modules are the basis of our inductive proof. While it is possible to construct a module \(A\) with constraints \(C\) for which no \(V\) exists that can satisfy \(C\), our definitions of satisfaction and validity handle this by requiring
post constraints to be satisfied only when both \( C \) and the pre constraints are satisfied. Thus, any module with unreasonably restrictive constraints is trivially valid. Under both these and more routine circumstances, the premises for the inference rule [Module] ensure that all typed modules trivially satisfy our theorem.

**Proposition 6.2.13 (Module).** If we have by the inference rule [Module] that \( \vdash (A, I, O, C) : C \) then it is the case that \( V \models (A, I, O, C) : C \).

**Proof.** The argument relies on our condition for constraints within the inference rule [Module].

Assert for any \( A, I, O, C, C' \),
\( (A, I, O, C), C' \) is a typed module and
\( C \Rightarrow C' \)
implies that
for any \( V \), if \( V \models \text{pre}((A, I, O, \{C\}), C') \) then
for all \( V' \) if \( V \subseteq V' \) and for all \( K \in \{C\}, V' \models K \) then
\( V' \models C \),
\( C' \Rightarrow \text{post}((A, I, O, \{C\}), C') \), and
\( V' \models \text{post}((A, I, O, \{C\}), C') \).

\[ \square \]

**Inductive Cases**

**Proposition 6.2.14 (Connect).** If \( V \models (M, I, O, C) : C \), \( V \models (N, I', O', C') : C' \), and we have by the inference rule [Connect] that
\( \vdash (\text{conn}(\theta, M, N), I'', O'', \text{cn}(\theta, C, C')) : C'' \)
then it is the case that \( V \models (\text{conn}(\theta, M, N), I'', O'', \text{cn}(\theta, C, C')) : C'' \).

**Proof.** The argument is lengthy and relies in part on the fact that the parameters of the constraint sets for the two networks \( M \) and \( N \) are disjoint.

Assert for any \( M, I, O, C, C, N, I', O', C', C', \theta \), for all \( I'', O'', C'' \),
\( I, O \) are sets, \( I', O' \) are sets, \( C \) is a constraint set, \( C' \) is a constraint set,
parameters(\( C \)) \cap \text{parameters}(\( C' \)) = \emptyset,
\( \vdash (M, I, O, C) : C \),
\( \vdash (N, I', O', C') : C' \),
\( \theta \in O \rightarrow I', \theta \) is injective,
\( I'' = I \cup (I' - \text{ran}(\theta)) \),
\( O'' = O' \cup (O - \text{dom}(\theta)) \),
\( C'' = C \cup C' \restriction I'' \cup O'' \), and
\( \text{post}((M, I, O, C), C) \Rightarrow \text{pre}((N, I', O', C'), C') \restriction \text{ran}(\theta) \)
implies that
6.2. SOUNDNESS OF THE NETSKETCH FORMALISM

for any $V$, if $V \models ((M, I, O, C) : C)$ and $V \models ((N, I', O', C') : C')$ then

if $V \models \text{pre}((\text{conn}(\theta, M, N), I'', O'', \text{cn}(\theta, C, C'))), C'')$ then

$$\text{pre}((\text{conn}(\theta, M, N), I'', O'', \text{cn}(\theta, C, C'))), C'') = \text{closure}(C'') \upharpoonright I''$$

$$V \models \text{closure}(C'') \upharpoonright I''$$

$$V \models \text{closure}(C' \cup C'') \upharpoonright I''$$

$$\text{closure}(C \cup C') \downarrow I'' \Rightarrow \text{closure}(C \cup C') \downarrow I''$$

$$V \models \text{closure}(C) \downarrow I$$

$$V \models \text{closure}(C) \downarrow I$$

$$V \models \text{pre}((M, I, O, C), C) = \text{closure}(C) \downarrow I$$

$$V \models \text{pre}((M, I, O, C), C)$$

for all $V'$, if $V \subseteq V'$ and for all $K' \in \text{cn}(\theta, C, C')$, $V' \models K'$ then

for all $K' \in C$, $V' \models K'$,

$V' \models \text{post}((M, I, O, C), C)$,

$V' \models \text{pre}((N, I', O', C'), C') \downarrow \text{ran}(\theta)$,

$$\text{pre}((N, I', O', C'), C') = \text{closure}(C') \downarrow I'$$

$$V' \models \text{closure}(C') \downarrow I' \downarrow \text{ran}(\theta),$$

$$\text{closure}(C') \downarrow I'' \Rightarrow \text{closure}(C') \downarrow (I' - \text{ran}(\theta)),$$

$$\text{closure}(C' \cup C') \downarrow I'' \Rightarrow \text{closure}(C') \downarrow I''$$

$$V' \models \text{closure}(C') \downarrow (I' - \text{ran}(\theta)),$$

$I' - \text{ran}(\theta) \subseteq I'$,

$$\text{closure}(C') \downarrow I' \downarrow (I' - \text{ran}(\theta)) = \text{closure}(C') \downarrow (I' - \text{ran}(\theta)),$$

$$V' \models \text{closure}(C') \downarrow I' \downarrow (I' - \text{ran}(\theta)),$$

$$(I' - \text{ran}(\theta)) \cup \text{ran}(\theta) = I'$$

$$V' \models \text{closure}(C') \downarrow I' \downarrow (I' - \text{ran}(\theta)) \cup \text{ran}(\theta),$$

$$V' \models \text{closure}(C') \downarrow I'$$

$$V' \models \text{closure}(C') \downarrow I'$$

$$V' \models \text{closure}(C') \downarrow I'$$

$$V' \models \text{pre}((N, I', O', C'), C')$$

$$V' \models (N, I', O', C') : C'$$
for all $V''$, if $V' \subseteq V''$ then
- for all $K'' \in \mathcal{C}'$, $V'' \models K''$ and $V'' \models K'$, for all $K' \in \mathcal{C}'$, $V'' \models K'$,
- $V'' \models \text{post}((M, I, O, C), C)$,
- $V'' \models \text{post}((N, I', O', C'), C')$,

$\text{post}((M, I, O, C), C) = \text{closure}(C) \upharpoonright O$,
$\text{post}((N, I', O', C'), C') = \text{closure}(C') \upharpoonright O'$,

$V'' \models \text{closure}(C) \upharpoonright O$,
$V'' \models \text{closure}(C') \upharpoonright O'$,
$V'' \models \text{closure}(C) \upharpoonright O \cup \text{closure}(C') \upharpoonright O' = \text{closure}(C) \cup \text{closure}(C') \upharpoonright O \cup O'$,

$V'' \models \text{closure}(C) \cup \text{closure}(C') \upharpoonright O \cup O'$,
$\text{closure}(C \cup C') = \text{closure}(C) \cup \text{closure}(C')$,

$V'' \models \text{closure}(C \cup C') \upharpoonright O \cup O'$,

$C \cup C' \subseteq \text{closure}(C \cup C')$,
$C \cup C' \upharpoonright I'' \cup O'' \subseteq C \cup C'$,
$O'' \subseteq O \cup O'$, $C'' \subseteq C \cup C'$,
$\text{closure}(C'') \subseteq \text{closure}(C \cup C')$,
$\text{closure}(C \cup C') \upharpoonright O \cup O' \Rightarrow \text{closure}(C'') \upharpoonright O''$,
$\text{post}((\text{conn}(\theta, M, N), I'', O'', \text{cn}(\theta, C, C'))), C'') = \text{closure}(C'') \upharpoonright O''$,

$V'' \models \text{post}((\text{conn}(\theta, M, N), I'', O'', \text{cn}(\theta, C, C'))), C'')$.

\[ \square \]

**Proposition 6.2.15 (Loop).** If $V \models (M, I, O, C) : C$ and we have by the inference rule [LOOP] that

$\vdash (\text{loop}(\theta, M), I', O', \text{lp}(\theta, C)) : C'$

then it is the case that $V \models (\text{loop}(\theta, M), I', O', \text{lp}(\theta, C)) : C'$.

**Proof.** The proof takes advantage of the side conditions in the inference rule.

---

**Assert for any $M, I, O, C, \theta$, for all $I', O', C'$, $M, I, O, C, I', O', C'$ are sets, $\vdash (M, I, O, C) : C$, $\theta \in O \rightarrow I$, $\theta$ is injective, and $I' = I - \text{ran}(\theta)$, $O' = O - \text{dom}(\theta)$, $C' = C \upharpoonright I' \cup O'$, and $\text{pre}((\text{loop}(\theta, M), I', O', \text{lp}(\theta, C)), C') \Rightarrow \text{pre}((M, I, O, C), C)$ implies that**
for any $V$, 
if $V \models ((M, I, O, C) : C)$ then
  if $V \models \text{pre}((\text{loop}(\theta, M), I', O', \text{lp}(\theta, C)), C')$ then
    $V \models \text{pre}((M, I, O, C), C)$,
    for all $V'$,
    if $V \subseteq V'$ and for all $K' \in \text{lp}(\theta, C)$, $V' \models K'$ then
      for all $K \in C$, $V' \models K$,
      $V' \models \text{post}((M, I, O, C), C)$,
      $\text{post}((M, I, O, C), C) = \text{close}(C) \upharpoonright O$,
      $V' \models \text{close}(C) \upharpoonright O$,
    $O = \text{dom}(\theta) \subseteq O$,
    $C \upharpoonright I' \cup O' \subseteq C$,
    $\text{close}(C) \upharpoonright O \models \text{close}(C') \upharpoonright O'$,
    $\text{post}((\text{loop}(\theta, M), I', O', \text{lp}(\theta, C)), C') = \text{close}(C') \upharpoonright O'$,
    $V' \models \text{close}(C') \upharpoonright O'$,
    $V' \models \text{post}((\text{loop}(\theta, M), I', O', \text{lp}(\theta, C)), C')$.

\[ \blacksquare \]

**Proposition 6.2.16 (Hole).** If $V \models (M_i, I_i, O_i, C_i) : C_i$ for $i \in \{1, \ldots, n\}$ where $M, I, O,$ and $C$ are vectors, and we have by the inference rule [HOLE] that
\[ \vdash (\text{hole}(X, \{M_1, \ldots, M_n\}), I', O', C') : C' \]
then it is the case that $V \models (\text{hole}(X, \{M_1, \ldots, M_n\}), I', O', C') : C'$.

**Proof.** The proof is a straightforward application of the side conditions in the rule [HOLE].

\[
\begin{align*}
\text{Assert for any } n, X, M, I, O, C, C', \text{ for all } M', I', O', C', C', \text{ where } \\
\text{for all } i \in \{0, \ldots, n\}, \\
\vdash (M_i, I_i, O_i, C_i) : C_i, \\
\text{for all } \phi \in I' \rightarrow I, \psi \in O_i \rightarrow O', \\
C' \cup \text{constraints}(\phi) \cup \text{constraints}(\psi) \equiv C_i, \\
M' = \text{hole}(X, \{M_i | i \in \{0, \ldots, n\}\}), \\
C' = \text{hl}(n, I, O, I', O', C), \\
\text{pre}((M', I', O', C'), C') \Rightarrow \text{post}((M', I', O', C'), C') \\
\text{implies that} \\
\text{for any } V, \text{ if } V \models (M', I', O', C') : C' \text{ then} \\
v \models \text{pre}((M', I', O', C'), C') \\
\text{implies that} \\
v \models \text{post}((M', I', O', C'), C') \\
\text{for all } V', \\
\text{if } V \subseteq V' \text{ and for all } K \in \text{hl}(n, I, O, I', O', C), V' \models K \text{ then} \\
v' \models \text{post}((M', I', O', C'), C').
\end{align*}
\[ \blacksquare \]
Proposition 6.2.17 (Weaken). If \( V \models (M,I,O,C) : C \) and we have by the inference rule [Weaken] that \( \vdash (M,I,O,C) : C' \) then it is the case that \( V \models (M,I,O,C) : C' \).

Proof. The argument is straightforward.

Assert for any \( M,I,O,C,C', \)

\[(M,I,O,C) : C, \]

\[ \text{pre}((M,I,O,C),C') \Rightarrow \text{pre}((M,I,O,C),C) \text{, and} \]

\[ \text{post}((M,I,O,C),C) \Rightarrow \text{post}((M,I,O,C),C') \]

implies that

for any \( V \),

if \( V \models (M,I,O,C) : C \) then

\[ V \models \text{pre}((M,I,O,C),C') \]

implies that

\[ V \models \text{pre}((M,I,O,C),C) \text{ and} \]

for all \( V' \),

if \( V \subseteq V' \) and for all \( K \in C \), \( V' \models K \) then

\[ V' \models \text{post}((M,I,O,C),C), \]

\[ V' \models \text{post}((M,I,O,C),C'). \]

\( \Box \)

6.3 NetSketch and the Integrated Automated Assistant

6.3.1 Module Design and Analysis with an Automated Assistant

The NetSketch DSL and formalism provide a means by which to perform compositional analysis of entire networks of modules (the NetSketch “Sketch Mode” in Fig. 6.1), and in Section 6.2.3 we dealt with some of the inference rules that enable this capability. However, the formalism’s compositional analysis capabilities do not address the issue of individual module construction (the NetSketch “Base Mode” in Fig. 6.1). The latter involves manipulation of common mathematical concepts such as dimensions, sets, graphs, constraints, and (in)equations. Furthermore, the constraints commonly found in a particular domain might be of certain classes (such as linear constraints, quadratic constraints, and so on), and thus could effectively be analyzed and solved by particular tools and techniques from a broader collection of tools for solving systems of constraints of various classes. There are many reference textbooks and survey papers to draw on given the task of assembling and using such tools [Lec09, HP95, Neu04, Kar08]). Consequently, the tool must leverage a varied collection of existing systems and formalisms.

An automated assistant can help a user manage these formal and mathematical concepts when modeling or assembling individual modules. The automated assistant provides an accessible and lightweight interface that allows the user to (1) represent and manipulate common concepts in the domain of application, (2) access and leverage an extensive ontology of definitions and facts about these concepts, (3) utilize a variety of tools and algorithms that operate on these concepts.

We consider integration of the NetSketch DSL with a aartifact. It is the automated assistant’s responsibility to recognize whether a particular manipulation is valid based on the system’s ability to utilize the algorithms, underlying tools, and ontologies integrated within it.

An automated assistant for formal reasoning can provide several useful capabilities (see Fig. 6.1). It can provide an interface for the variety of tools a domain expert may want to employ while modeling or assembling CFNs. Specifically, it can leverage domain-specific ontologies of relevant propositions and concepts in helping a domain expert manipulate and reason about constraint collections. This includes (1) checking the sorts (i.e. dimensions) of subexpressions within complex constraints, (2) identifying the classes to which a
6.3. NETSKETCH AND THE INTEGRATED AUTOMATED ASSISTANT

constraint belongs (and invoking appropriate solvers), (3) reasoning algebraically about relationships between constraints (with support for a symbolic manipulation and evaluation).

6.3.2 A NetSketch Use Case and the Automated Assistant

We consider an example of a CFN scenario suitable for analysis using the NetSketch DSL: a vehicular traffic problem, inspired by problems in the study of traffic flow [Mae06]. We focus on how an automated assistant can be useful for an engineer who is modeling modules in a domain such as this. A more extensive examination of this example, including how the engineer in this scenario would assemble networks out of many typed module instances, is examined more in a pertinent report [BKLO09a]. The examples in this discussions are simple and only illustrate the use of an automated assistant; more extensive and sophisticated examples can be found in the full technical report describing this work [LKB10].

Format of Presentation. Formal definitions and arguments parsed by the aartifact system are presented within a framed box (example provided below).

Assert for any \( x, y, z \in \mathbb{R} \), if \( x > y \) and \( y > z \) then \( x > z \).

Traffic Flow Example. An engineer working for a metropolitan traffic authority has the following problem. Her city lies on a river bank across from the suburbs, and every morning hundreds of thousands of motorists drive across a few bridges to work in the city center. Each bridge has a fixed number of lanes, but they are all reversible (an operator can decide how many lanes are available to inbound and outbound traffic at different times of the day). The engineer must decide how many inbound lanes should be open in the morning with the goal of ensuring that no backups occur within the city center, with the secondary goal of maximizing the amount of traffic that can enter the city. The city street grid is a network of a large number of only a few distinct kinds of traffic junctions, such as the merge junction described below. The junction and the problem the engineer must solve locally within it is modelled as a module.

Derivation of Sorts for Constraint Subexpressions. The capabilities of an automated assistant are a strict superset of the capabilities of a type inference algorithm, so it is natural to task the automated
Assistant with deriving and verifying the sorts of any expressions within the constraints introduced by the user. In order to model the notion of “no backups”, it is necessary to utilize two sorts of parameters: density measured in tons per kilometer (T/km), and velocity measured in kilometers per hour (km/hr).

We consider an example of a module. A merge junction has two incoming links (call them 1 and 2) and one outgoing link (call it 3). This traffic junction can be modeled using a module (call it $A_M$) consisting of a single node (call it $M$) for which the formal definition is:

$$
N = \{M\}, \text{In} = \{1, 2\}, Q = \emptyset, \text{Out} = \{3\}
$$

$$
\text{Par} = \{1 \mapsto (v_1, d_1), 2 \mapsto (v_2, d_2), 3 \mapsto (v_3, d_3)\}
$$

$$
\text{Con} = \{ \text{regulating traffic through } M \} \cup \{ \text{lower/upper bounds on } v_i, d_i \}
$$

This definition specifies the structure of the junction, with the constraint set $\text{Con}$ only containing predefined constraints that are required for all merge junctions:

$$
(1) \quad d_1 + d_2 = d_3 ; \quad (2) \quad d_1 \ast v_1 + d_2 \ast v_2 \leq d_3 \ast v_3
$$

Constraint (1) enforces conservation of density when $M$ is neither a “sink” nor a “source”, whereas constraint (2) encodes the non-decreasing flow invariant, namely that traffic along the exit link may accelerate or become more dense.

When the two constraints are introduced, the automated assistant automatically derives sorts for each subexpression, as well as the correctness (and thus meaningfulness) of the overall constraint. For example, because the dimension associated with both $d_1$ and $d_2$ is T/km, the dimension associated with $d_1 + d_2$ is also T/km. Because T/km is the dimension associated with the expressions on both sides of the constraint, (1) is validated as a well-sorted constraint. Furthermore, in constraint (2), because $d_2$ is T/km and $v_2$ is km/hr, $v_2 \ast d_2$ is T/hr (called mass flow).

**Symbolic Verification of Relationships between Constraints.** Suppose that the user introduces two more constraints that represent an upper bound $k$ on the increase in traffic density on the outgoing link:

$$
(3) \quad d_3 \leq d_1 + k ; \quad (4) \quad d_3 \leq d_2 + k.
$$

Together with constraints (1) and (2), these constraints have some noteworthy consequences for the module $M$ that can be derived using a few algebraic manipulations:

| Assert | $d_3 \leq d_1 + k,$ | $d_1 + d_2 \leq d_1 + k,$ and $d_2 \leq k.$ |
| Assert | $d_3 \leq d_2 + k,$ | $d_1 + d_2 \leq d_2 + k,$ and $d_1 \leq k.$ |

Thus, these constraints in fact put an upper bound on the traffic density over incoming links. Now, suppose the user introduces two more constraints,

$$
(5) \quad d_3 \geq d_1 + d_2 \quad (6) \quad v_3 \geq v_1 + v_2,
$$

and wishes to know whether these guarantee that kinetic energy ($\text{km} \cdot \text{T/hr}^2$) is preserved:

$$
(7) \quad v_3^2 \ast d_3 \geq v_1^2 \ast d_1 + v_2^2 \ast d_2.
$$

This, too, can be verified using a sequence of algebraic manipulations.

| Assert | $v_3^2 \ast d_3 \geq (v_1 + v_2)^2 \ast d_3,$ |
| Assert | $(v_1 + v_2)^2 \ast d_3 \geq v_1^2 \ast d_3 + v_2^2 \ast d_3 + 2 \ast v_1 \ast v_2 \ast d_3,$ |
| Assert | $v_1^2 \ast d_3 + v_2^2 \ast d_3 + 2 \ast v_1 \ast v_2 \ast d_3 \geq v_1^2 \ast d_3 + v_2^2 \ast d_3,$ |
| Assert | $d_3 \geq d_1$ and $d_3 \geq d_2,$ |
| Assert | $v_1^2 \ast d_3 + v_2^2 \ast d_3 \geq v_1^2 \ast d_1 + v_2^2 \ast d_2,$ |
| Assert | $v_3^2 \ast d_3 \geq v_1^2 \ast d_1 + v_2^2 \ast d_2.$ |
Derivation of Constraint Class. A capability closely related to sort derivation is the derivation of constraint class. For example, simple syntactic propositions can be used to conclude that (1), (3), (4), (5), and (6) are linear constraints, (2) is a quadratic constraint, and (7) is a cubic constraint. Automated derivation of a constraint’s class is useful in determining what algorithms can be applied to characterize the solution space for a set of constraints.

In its most simple incarnation, this capability can be accomplished through a trivial, superficial analysis of the syntactic structure of an expression.

**Assert**

$d_1 = d_2 + d_3$ is a linear constraint.

Assert $d_1 \cdot v_1 \leq d_2 \cdot v_2 + d_3 \cdot v_3$ is a quadratic constraint.

**Evaluation-based Verification of Relationships between Constraints.** This method would involve evaluating expressions on specified ranges. The automated assistant can call appropriate constraint solvers as subroutines in order to accomplish this, or it can simply act as a calculator and perform an exhaustive evaluation of all possibilities within fixed ranges for the parameters involved.

Consider a crossing junction that has two incoming links (call them 1 and 2) and two outgoing links (call them 3 and 4). The corresponding gadget (call it $\mathcal{A}_X$) again has a single node, call it $X$, defined as follows:

$$
N = \{X\}, \text{In} = \langle 1, 2 \rangle, \text{Out} = \langle 3, 4 \rangle
$$

$$
\text{Par} = \{1 \mapsto v_1 \cdot d_1, 2 \mapsto v_2 \cdot d_2, 3 \mapsto v_3 \cdot d_3, 4 \mapsto v_4 \cdot d_4 \}
$$

$$
\text{Con} = \text{Con}_{\text{nodes}} \sqcup \text{Con}_{\text{links}} \quad \text{where}
$$

$$
\text{Con}_{\text{nodes}} = \{ \text{regulating traffic through } X \}
$$

$$
\text{Con}_{\text{links}} = \{ \text{lower/upper bounds on } v_1, d_1 \ldots \}
$$

Suppose all traffic entering through link 1 must exit through link 3 and at the same velocity, and all traffic entering through link 2 must exit through link 4 and at the same velocity. This is expressed by four constraints:

$$
v_1 = v_3; \quad v_2 = v_4; \quad d_1 = d_3; \quad d_2 = d_4
$$

If the total density of entering traffic, namely $d_1 + d_2$, exceeds a “jam density” that makes the two entering traffics block each other, there will be backups. We therefore presume there is an upper bound, say 10, on $d_1 + d_2$ below which the two traffics do not impede each other and there are no backups as a result:

$$
d_1 + d_2 \leq 10
$$

Below a total density of 10, we can imagine that the two incoming traffics are sparse enough so that they smoothly alternate taking turns to pass through the crossing junction.

We can perform a simple verification that $\{0, \ldots , 5\}$ is a valid range for the components $d_i$ simply by asserting the claim within the automated assistant.

**Assert for all $d \in \{0, \ldots , 5\}^4$

$d_1 = d_3, \; d_2 = d_4$

implies that

$d_1 + d_2 \leq 10, \; d_3 \in \{0, \ldots , 5\} \text{ and } d_4 \in \{0, \ldots , 5\}$. 
Chapter 7

A Formal Framework for Service-Level Agreements

Desirable application performance is typically guaranteed through the use of Service-Level Agreements (SLAs) that specify fixed fractions of resource capacities that must be allocated for unencumbered use by the application. Having the flexibility to transform SLAs from one form to another in a manner that is provably safe would enable hosting solutions to achieve significant efficiencies. We employ the AARTIFACT lightweight formal reasoning assistant to reason about the consistency of a proposed type-theoretic framework for the representation and safe transformation of SLAs. Relevant constructs within the framework are modelled using basic mathematical concepts such as sets, vectors, ranges, and related properties. Proofs about the safety of transformations in the framework consist of mechanically verifiable arguments that are manageable in length and complexity thanks to the AARTIFACT system’s user-friendly design. This application of the system illustrates the benefits of adopting a conventional and simple concrete syntax for formal arguments, of exposing to the user a database of supported syntactic idioms, and of allowing users to implicitly employ a database of propositions that deal with common mathematical concepts.

7.1 Background

We evaluate the effectiveness of the system by utilizing it to reason about a particular type-theoretic framework for modelling service-level agreements that are commonly used within the context of virtualization. Virtualization has been widely adopted due to its various benefits such as cost efficiency, deployment flexibility, and simplified management. It delivers these benefits by allowing users to acquire appropriate fractions of shared fixed-capacity resources for use subject to well-defined, binding Service-Level Agreements (SLAs) that ensure the satisfaction of minimal Quality of Service (QoS) requirements.

Such SLAs are guaranteed by the scheduling mechanisms of the hosting environment, which provide the appropriate resource fractions normalized over a relatively long time scales. While this might be appropriate for most applications, such coarse SLAs will not cater well to the needs of real-time applications, whose QoS constraints require resource allocations at a more granular level i.e. being provided with the appropriate resource fractions over short time scales.

Hierarchical scheduling, a subject of research for the past decade [SL03, RS01, ELSS07, HM06], has been suggested as an effective mechanism for providing scheduling of a finer granularity than is supported by virtualization. In this approach, the granularity of the reservations is refined using virtualization layers. Using hierarchical scheduling, resources are allocated by a parent scheduler at one level of the hierarchy to a child scheduler (or a leaf application) at the next level of the hierarchy. Conceptually, at any given layer of this hierarchy, the parent scheduler can be seen as allocating a virtual slice of the host at some granularity which is further refined by lower-layer schedulers, until eventually appropriated and consumed by a leaf application.
The problem addressed by the framework in question is the inference of a feasible and efficient hierarchical scheduler. Given a set of applications (tasks) each of which specified by minimal resource utilization requirements (SLAs), how should these tasks be mapped to the leaves of a forest (whose internal nodes represent virtual hosts and whose roots represent physical hosts)? This assumes that the SLAs requested by tasks (or offered by hosts) are mutable: it could be the case that there are multiple, yet functionally equivalent ways to express the resource requirements of a periodic real-time task. It is possible to rewrite SLAs as long as such rewriting is safe. The availability of such SLA transformations enables the consideration of different colocation possibilities (and associated hierarchical structures), thus allowing us to explore alternative mappings.

### 7.2 Example

SLAs can be seen as encapsulators of the resources supplied by hosts (producers) and demanded by tasks (consumers). While this concept is generic enough for a wide variety of resources, we provide a specific model for SLAs – namely, one that supports periodic, real-time resource supply and demand.\(^1\) A Service-Level Agreement (SLA) type is defined as a tuple of natural numbers \((C,T)\), \(C \leq T\), where \(C\) denotes the resource capacity supplied or demanded in each allocation interval \(T\). Figure 7.1 illustrates an example of a simple formal argument about the safety of a particular kind of SLA transformation: if a schedule satisfies \((C,T)\), it also satisfies \((C,T')\) for any \(T' \geq T\).

---

**Assume** for any \(C,T\), \((C,T)\) is an SLA iff \(C \in \mathbb{N}, T \in \mathbb{N}, C \geq 1, \text{ and } T \geq 1\).

Assume for any \(a,C,T\),

\[
a \text{ satisfies } (C,T) \iff (C,T) \text{ is a task, } a \in \{0,1\}^\infty, \text{ and for all } m \in \mathbb{N}, a_m + \ldots + a_{m+(T-1)} \geq C.
\]

Assert for any \(a,C,T\),

\[
\text{if } (C,T) \text{ is an SLA and } a \text{ satisfies } (C,T) \text{ then for all } T' \in \mathbb{N},
\]

\[
\text{if } T' \geq T \text{ then } (C,T') \text{ is an SLA, and for all } m \in \mathbb{N},
\]

\[
a_m + \ldots + a_{m+(T'-1)} \geq a_m + \ldots + a_{m+(T-1)},
\]

\[
a_m + \ldots + a_{m+(T'-1)} \geq C;
\]

\[
\text{for all } m \in \mathbb{N}, a_m + \ldots + a_{m+(T-1)} \geq C,
\]

\[
a \text{ satisfies } (C,T').
\]

---

Figure 7.1: A verifiable formal argument about the safety of a simple SLA transformation. The argument makes references to integers, sets of integers in specified ranges, summations thereof, and related properties.

This example illustrates the capability of the aartifact system to support reasoning about concepts within this application. In particular, the argument contains:

- user-defined predicates (e.g. “(\(C,T\)) is an SLA”);

\(^1\)Readers familiar with real-time scheduling work should note that our periodic, real-time SLA model mirrors existing periodic task models in the literature (e.g., [HR95, BBL99, BLA98, WSP04, ZWQ04]).
7.2. EXAMPLE

- use of syntactic idioms (e.g. \(S^{\infty}\) to refer to the set of vectors of unbounded length with components from a set \(S\), and \(v_1 + \ldots + v_n\) to refer to the sum of a range of components in a vector \(v\));
- implicit utilization of a variety of properties about sets of integers in a range (e.g. if \(a' \leq a\) and \(b \leq b'\) then \(\{a, \ldots, b\} \subseteq \{a', \ldots, b'\}\).

This example illustrates how we have utilized the AARTIFACT system for several results about SLA types as defined by the framework. The definitions and properties of the framework are defined in a machine-readable representation that is also uncluttered and accessible to humans. These arguments take full advantage of the system’s support for reasoning about common mathematical concepts (provided by a large collection of propositions in the database that deal with sets and related concepts) in employing, without explicit references, laws that govern the relationships between the concepts involved.
Bibliography


