Finding Unity in Computational Logic

Dale Miller
INRIA Saclay and LIX/École polytechnique
Route de Saclay, 91128 PALAISEAU Cedex, France
dale.miller@inria.fr

While logic was once developed to serve philosophers and mathematicians, it is increasingly serving the varied needs of computer scientists. In fact, recent decades have witnessed the creation of the new discipline of Computational Logic. While Computation Logic can claim involvement in many, diverse areas of computing, little has been done to systematize the foundations of this new discipline. Here, we envision a unity for Computational Logic organized around recent developments in the theory of sequent calculus proofs. We outline how new tools and methodologies can be developed around a boarder approach to computational logic.

Computational logic, unity of logic, proof theory

1. SOFTWARE AND HARDWARE CORRECTNESS IS CRITICALLY IMPORTANT

Computer systems are everywhere in our societies and their integration with all parts of our lives is constantly increasing. There are a host of computer systems—such as those in cars, airplanes, missiles, hospital equipment—where correctness of software is paramount. In the area of consumer electronics, big changes in the attitude towards correctness has taken place. A decade ago, bugs and errors in, say, desktop PCs, music players, and telephones, were mostly nuisances and were not "life-threatening": such flaws were fixed by rebooting the system or by living without a feature. Today, however, these same devices are tightly integrated into networks and bugs and errors open us to attacks from malicious software, breaches in security, loss of anonymity, *etc*.

Attempting to establish various kinds of correctness-related properties of software systems is no longer an academic curiosity. A typical platitude used to motivate the development of a strong basis for the correctness of computer systems was something like "You can't build a tall building on a sandy beach; one needs a solid foundation." The modern updating of that statement requires moving off the beach into the water: "If you are in a canoe, a small leak might be manageable: if you are in a submarine, a small leak is lethal." As it is painfully clear today, plugging your computer into the Internet is similar to descending into the depth of the sea: if there is a crack in your security, it will be exploited quickly. One cannot be relaxed anymore about leaks.

Our ability to provide at least some formal guarantees about software systems will be directly related to our ability to deploy new functionality and services. The demanding requirements in safety critical systems matched with our inability to formally establish their correct behavior means that developments and deployments of, say, avionic systems or high-speed train switches, moves at a glacial pace and clings to old technological solutions that have no history of failing. If we cannot distinguish applets from viruses, we cannot expect people to really use the rich set of flexible services organized around mobile code. Our future could resemble William Gibson's novel *Virtual Light*, in which network security was so bad that important data was transferred by bikers carrying hard-disks! If we cannot produce software with at least certain guarantees, the development and deployment of all the new features and services that we all hope to see in our hardware and software system will be greatly delayed.

2. LOGIC IS A KEY

It is to logic that researcher, designers, and practitioners turn to help address the problems of establishing formal properties. The importance of logic comes, in part, because of its universal character and the rich set of results surround it. Speaking most broadly, logic plays two main roles. In one such role, logical expressions themselves form the core programming languages: functional and logic programming paradigms start here. In this role, logic does not guarantee correctness *per se* but the many deep theoretical properties that have been established for logic can be directly applied to the problem of establishing formal properties. A second

and more popular role for logic involves using logical expressions to specify something about how a program actually works. In this setting, there are two languages, the *ad hoc* one of the programming language and the *principled* one of logic.

Not only can logic be used to formally establish correctness (*e.g.*, proving a program correct), its universal character leads it to be used on a language for communicating meaning between different entities. For example, humans designers who wish to communicate precisely a programming language's meaning to users or implementers of that language will use logic-based formalism to capture such meaning. Also, when machines need to exchange data and programs, logical expressions, via typing, memory layout specifications, correctness certificates (*e.g.*, in the proof carrying code setting), *etc*, are often written using logic.

Indeed, logic plays a role in computer science similar to that played by differential and integral calculus in the physical sciences and engineering disciplines [12, 21]. Twenty years ago, Martin Davis [12] observed that the mathematical logic had already an intimate relationship with computer science.

> When I was a student, even the topologists regarded mathematical logicians as living in outer space. Today the connections between logic and computers are a matter of engineering practice at every level of computer organization. . . . Issues and notions that first arose in technical investigations by logicians are deeply involved, today, in many aspects of computer science.

Since these words were written, the deep involvement of these two disciplines has grown so rich that it has given rise to the new field of *computational logic*.

3. ... BUT LOGIC HAS BEEN BADLY FRACTURED

While there is some recognition that logic is a unifying and universal discipline underlying computer science, it is far more accurate to say that its universal character has been badly fractured in the past few decades along a number of axes.

- Logic has entered into a wide range of application areas, including, for example, computer architecture, databases, software engineering, programming languages, computational linguistics, and artificial intelligence. These different application areas have been pushing their own agendas on how logic should be exploited.
- Many logics have been invented in recent years. The number of adjectives that are now routinely added to the word "logic" is frightening: first-order / higher-order, classical, intuitionistic, linear, modal (S4, S4, S4.1, S4.2, . . .), temporal (LTL, CTL, . . .), deontic, dynamic, quantum, etc. With so many adjectives in common use today, one wonders if there is any sense to insisting that there is a core notion of "logic".
- There are a large number of computational logic tools: model checkers, interactive and automatic theorem provers, logic circuit simulators/testers, type inference systems, etc. Within each of these categories, there are a plethora of specific techniques and tools that often have little relationship to one another.

The use of the word "fractured" here is deliberate. Developing many different sub-disciplines is a typical development within maturing disciplines: for example, within mathematics, there are a great many adjectives applied to the term *algebra*. In this case, however, many of those sub-disciplines of algebras were developed to provide for *commonality* by making more abstract previously developed and disparate algebraic structures. But one sees little effort within the literature of computational logic to provide for commonality.

Specialization has made it possible for logic to contribute significantly in these many application areas and to attract the interest of many in industry. On the other hand, this fracturing of logic comes with a high cost to the discipline and it greatly diminishes its potential. In particular, theoretical and implementation results achieved in one slice of applied logic are seldom exported to other slices. Similarly, great efforts are applied to develop tools, libraries, and packages for one tool that are closely related to large efforts based in other tools. More serious still is that people working in one narrow domain will sometimes think of their sub-domain as being the universal formalism: since they are missing the bigger picture, they invent ways to make their domain more expressive even when much of what is needed is already accounted for in (other slices of) logic.

In this paper, we argue that there needs to be forces that are pushing against this fracturing and that attempts to see a core of computation logic as being based on few but deep concepts.

4. THE STATE-OF-THE-ART

4.1. The early dreams for logic

One of the first dreams that one has for logic is that it is *universal* in its scope. First-order classical logic, for example, can be formalized using both syntactic means (proofs) and semantic means (truth) and it can be proved that these characterize the same set of formulas: namely, the theorems. On such a secure foundations,

one can build, for example, set theory. It was natural, therefore, to consider universal (complete) methods for implementing logic as a way to automate proofs in mathematics and computer science. A slogan of early work in computational logic might have been: "universal logic and universal implementation implies universal solutions".

Early dreams in automated reasoning, model checking, and logic programming focused on simple and theoretically complete methods with the hope that they would be yield comprehensive solutions in practice. The hope was to have one framework and one implementation that provided universal applicability.

Automated and interactive provers In the 1960-1970's, there was a great deal of work implementing automated systems for first-order logic that was based on such complete paradigms as resolution [38] or on conditional rewriting [8]. Such early work produced a great deal of information about proof strategies and methods to implement formal logical systems effectively (unification, backtracking search, rewriting strategies, *etc*). Another lesson from these early systems was more disappointing: those systems came no where near achieving the ambitions of effective, universal deployment. Only "toy" theorems could be proved automatically. Furthermore, the nature of logic automation was such that the usually speed up in program execution that resulted from improvements in hardware and compilers would not make much of a dent in the "state-explosion" that occurred with such provers. Starting around the same time (and continuing to today), a number of interactive proof environments for mathematics were developed: Automath [13], Nqthm [8], Mizar, [39], NuPRL [9], Coq [10], PVS [35], and Matita [4]. (See, for example, Geuvers's recent survey article on proof assistants [17].) By in large, these system chose either first-order logic with induction or a higher-order constructive logic based on intuitionistic logic as a suitable framework for encoding mathematics. Also, while interaction was central to the functioning of such systems, they all incorporated the ability for integrating and extending automation to some extent, often using tactics and tacticals. These kinds of interactive and largely programmable theorem provers have turned them into proof editors and proof checkers but this move has greatly extended their effective deployment.

Model checkers and logic programming By shifting one's attentions to simpler theorems involving weaker properties (for example, shifting from full correctness to detecting deadlocks), one can employ logic and deduction using the ideas and techniques found in model checking [15, 37]. While great successes can be claimed for such systems, the hope of having a universal approach to deduction in the model checking setting again quickly run into the state explosion problem. If we are willing to deal with still weaker properties, logic programming techniques can be used to explore algorithmic aspects of logic. The Prolog language, for example, exploits the Horn clause fragment of first-order logic to provide a programming language that can, after a fashion, turn some declarative specifications into proper programs. But again, the universality of this dream of deductively describing a relationship and then getting an effective implementation of that relationship turned out to be largely illusory.

4.2. Making an impact by specialization

Much of this early work yielded important results and lessons. One of those lessons was, however, that universal methods were usually of little practical use and that the hope to deploy them in even rather weak settings was naive. This early work then lead to a new phase in the employment of logic for mathematics and computer science.

Pick domains and specialize One way to make deductive systems more practical involves having researchers focus on applications and sub-domains. Once an application domain is narrowed significantly, specific approaches to deduction in that setting could be developed. There have been any number of highly successful examples of this style of narrow-and-specialize, including, for example, SAT solvers, deductive databases, type systems, static analysis, logic circuits, etc. Such systems are making routine and important contributions in day-to-day practice.

Many different frameworks In [36], Paulson described his Isabelle theorem prover as a generic framework in which the "next 700 theorem provers" could be written. The argument (largely implicit in that paper) is that writing a theorem prover is a difficult task and future designers of theorem provers should work within an established framework. Such a framework can help to ensure correct implementations of core deduction features as well as provide for basic user-interfaces, and integration with various specialized inference engines (for example, Presburger arithmetic). Such frameworks are now popular choices and allow proof system developers to either explicitly designed new logics (as is the case in Isabelle) or extended the deductive powers of a core prover using various library and package mechanisms (as is the case in many other provers such as Coq, HOL, and NuPRL). Working entirely within a particular logical framework is certainly a conservative perspective that is the appropriate choice in many situations.

4.3. Verification is too big for formal methods

The universal applicability of logic and its associate proof methods on computer system verification has also been attached from another angle. De Millo, Lipton, and Perlis [31] have stressed that formal proofs (in the sense

often attributed to mathematics) is unlikely to work for the verification of computer systems give that the latter involves social processes, evolving specifications, and remarkably complex specifications and programs.

On many occasions in computer science, a negative result can be productive: witness, for example, who the undecidability of the halting problem leaded the extensive study of specialized domains where decidability can be established. Similarly here: if one accepts the negative premise that logic and formal proof cannot solve the problems of verifying computer systems, then one might expect to see an explosion of many logics and proofs used on many smaller aspects of building correct software. For example, the social processes involved in building computer systems must communicate precisely among various people involved in that process. There are many things that need to be communicated between the members of the society (tools, types, static analyzes, operational semantics, examples, counter-examples, etc). Logic, with its precision and its possibility of universality, can and has been applied. Not all things, of course, *are* logic but logic offers an extremely valuable aid in defining, formalizing, automating, and communicating many such aspects of software systems. Thus, the impossibility of using one logic and formal method leads to the need to have many specialized logics.

5. BENEFITS OF A UNIFYING FRAMEWORK

Some communities realize this fractured nature of logical systems and propose *ad hoc* solutions: standardized challenge problems, standardized frameworks, XML formats for formulas and proofs, etc. Other researchers are involved with finding protocols for plugging one tool into another tool. While well engineered and inter-operating systems can have important practical consequences, we shall insist that we must also develop a broader and more expressive foundation for computational logic, one where the many applications of logic can be explained with a common foundation.

There are a number of important consequences to providing such a common framework to logic and its uses in computer systems.

Transfer of implementation techniques A formal and rich foundation for a wide variety of computational logic systems will allow for researchers and developers to see that solutions they have developed in one area—*e.g.*, data structures, algorithms, or search heuristics—can be transfer to other areas.

Rich integration of different technologies In a similar way, it should be possible to see that different tools are, at least formally, performing deduction in the same kind of proof systems as another tool: as such, rich forms of integration of those tools should be made possible. For example, it is possible for model checking and inductive/coinductive theorem proving to be seen as building sequent calculus proofs in a logic with fixed points [7, 5]. Such a common deductive setting can be used to more tightly integrate these two rather different approaches to logic.

New breed of computational logic systems One reason to push for new foundations over engineered integration is that such new foundations should make it possible to provide for completely new approaches to the architecture and scope of logic-based systems. For example, linear logic [18] was presented as a revolution in computation logic and it has, indeed, made it possible to rethink a great deal of the conceptual nature of logic. Today, we have much richer ways of thinking about the structural rules (*e.g.*, contraction, weakening), about the role of games and interaction in logic, about concurrency in proofs, and about organizing inference rules into large-scale inference rules.

Teaching of logic An extremely important aspect of the foundation of *any* science is its ability to explain clearly the totality of the science. A new foundation should provide a meaningful way to organize and present most aspects of computational logic systems. In turn, such developments will lead to new ways to teach logic so that its unity can be stressed.

6. METHODOLOGY

What exactly is logic? Since we are exploring the frontiers of what logic can be for computer science, we do not try to completely define it here. On the other hand, we have the most ambitious plans for logic. In particular, we shall always use it as a term that can be ascribed "beauty" in the sense of the following quotation.

> We ascribe beauty to that which is simple; which has no superfluous parts; which exactly answers its end; which stands related to all things; which is the mean of many extremes.
>
> — Ralph Waldo Emerson, The Conduct of Life, "Beauty" (1860)

In particular: logic is *simple*, given by its natural and universal syntax and small sets of inference rules; logic has *no superfluous parts*, which is the promised of such formal results as the cut-elimination theorem; logic *exactly answers its ends* for describing static truth or computational dynamics, as witnessed by soundness and (relative) completeness results; logic is *related to all things* computational and its role in the foundations of computer science is often compared to the role of calculus in the foundations of physics [21]; and, finally, logic is the *mean of many extremes* given its intimate use in a range of "extremes" from databases, to

programming languages, to type systems, to certificates, to verification, to model checking, etc.

We shall also not try to find a single setting to discuss all things that have been referred to as logic. In particular, we shall mostly limit ourselves to classical, intuitionistic, and linear logic since they have a long and well established relations to computing. Many other logics, such as modal, spatial, tense, *etc*, will be explicitly left out of this discussion, even though many of them can be understood as embeddings into or modular extensions to one of these core logics. Even with a focus on three core logics, there are a many ways that these can be elaborated (*eg.*, propositional versus quantifications, first-order versus higher-order quantification, with or without equality) and there are even more proof systems for these logics (sequent, natural deduction, tableaux, Hilbert-style, matrix methods, *etc.*).

6.1. Harder problems, better solutions

One should not fear attacking problems of integration since they often lead to better and more elegant solutions. For example, several computational logic systems often limit themselves to the classical theory of first-order Horn clauses (systems such as logic programming, rule-based systems, and deductive databases). This particular logic is, however, so weak that many properties about it holds *almost by accident*: for example, on this set of formulas, classical, intuitionistic, and linear logics coincide. It is not until one considers extensions to Horn clauses that one finds deeper phenomena. For example, it was moving to higher-order versions of Horn clauses that the sequent calculus was first used to replace more traditional resolution proof systems for Horn clauses. Once sequent calculus was introduced, a wealth of proof theory results and innovations (such as linear logic, focused proof systems, etc) could to be applied to the proof-search (logic programming) paradigm. Ultimately, it was discovered that a certain kind of completeness theorem for logic programming [28] could be applied to all of (linear) logic [3]. Finally, logic programming and theorem proving for full logics were integrated into a common framework.

6.2. The unity of (computational) logic

We shall look to logic to be a universal language and proof theory as a universal framework to organize and infer structure about both logic and computation. Logic can be composed of a great many connectives, quantifiers, etc. Proof theory teaches us that if we can achieve cut-elimination for logics, then we can expect that most features of logic fit together orthogonally. That is, they do not interact or, if they interact, that interaction is made evident and controlled. As a result of this orthogonality, we have the opportunity to see logic as a rather large collection of possible connectives, quantifiers, and other operators (*e.g.*, exponentials, modals, fixed points [7], and subexponentials [33]) and that we can choose from these as we wish. In this sense, the propositional classical logic system used within SAT solvers is, in fact, just one of many subset of, say, higher-order linear logic. Modern proof theory also teaches us that *contexts* and their associated *structural rules* (in contrast to *introduction* and *elimination* rules) play an important role in describing logics. But again, the choice of what structural rules to use is largely orthogonal to other choices. For example, Gentzen's original version of the sequent calculus was developed to unify the treatment of classical and intuitionistic logic: the difference between these two logics was governed by structural rules. Girard later showed that linear logic could fit into this same scheme by further varying the structural rules. Richer integration is also possible, where, say, linear and classical connectives can exist together [19, 24].

It is also clear that there are limits to some integrations of logic. In particular, if logic is use to specify computations (*i.e.*, by having proofs be computation traces) then reasoning about such computations might well need to be based on a different logic. The standard division of *object-level* and *meta-level* reasoning works here. Even in this setting, some important integration is still possible: in particular, terms structures and their associated binding operators can be shared between the meta-logic and the object-logic.

6.3. Reconsidering the role of mathematics

It seems important to reconsider the relationship between mathematics and computation. In many ways, mathematics does not provide a good framework for understanding computation. While denotational semantics of, say, the λ-calculus might well be considered a proper mathematical topic, results in, say, denotational semantics, while deep and revealing, seem more crafted to an applications and less suited as a general framework for organizing a wide range of topics.

The roles for logic we consider here deal with not only specifying computation but also *reasoning about computation*: reasoning about mathematics is not, a priori, our concern. This choice of emphasis actually has significant consequences. In particular, the traditional approach to reasoning about computation in almost all ambitious theorem proving systems today follows the following two step approach.

Step 1: *Implement mathematics.* This step is achieved by picking a general, well understood formal system. Common choices are first-order logic, set theory, or some foundation for constructive mathematics, such as a higher-order intuitionistic logic. Such a framework is then usually provided with rich abstraction and modularity mechanisms that aid in the construction of large theories.

Step 2: *Reduce computation to mathematics.* Computation is generally encoded via some model theoretic semantics (such as denotational semantics) or as an inductive definition over an operational semantics.

A key methodological element for us here is that we shall drop mathematics as an intermediate and attempt to find a more direct and intimate connections between computation, reasoning, and logic. One main reason that reasoning about mathematics and about computation can differ is that many elements of computation have rather "intensional" treatments whereas the treatment of things mathematical often are directly treated extensionally. The notion of *algorithm* is an example, of course, of this kind of distinction: there are many algorithms that can compute the same function (say, the function that sorts lists). In a purely extensional treatment, it is functions that are represented directly and algorithm descriptions that are secondary. If an intentional default can be managed instead, then function values are secondary (usually easily captured via the specification of evaluators or interpreters.

For a more explicit example, consider whether or not the formula

$$\forall w_i.\ \lambda x.x \neq \lambda x.w$$

is a theorem. In a setting where λ-abstractions denote functions (the usual extensional treatment), we have not provided enough information to answer this question: in particular, this formula is true if and only if the domain type i is not a singleton. If, however, we are in a setting where λ-abstractions denote syntactic expressions, then it is sensible for this formula to be provable since no (capture avoiding) substitution of an expression of type i into $\lambda x.w$ can result in $\lambda x.x$.

Computation is full of intensional features besides bindings within syntax, including, for example, the usage of resources such as time and space. Mathematical techniques can, of course, treat intensionality, but experience with such treatments demonstrate that they do not reach an acceptable level of "canonicity". Logic and, particularly, proof theory, can provide rather direct treatments of many of these intensional aspects of computation.

Of course, mathematics, as something foreign and different from computation, might provide interesting and fresh challenges to our thinking of computation. One may have mathematical structures based on, say, partial orders or topological constructions, that pose interesting insights into how computation might be achieved. In a similar fashion, the physical world might well provide interesting ways to rethink computation: the example of quantum computing immediately come to mind.

6.4. Proof theory as an alternative approach to meaning

"Proof theory semantics" is a term that has been used for a number of years, largely in the narrow and philosophical context of determining the proper meaning of the logical connectives [22]. Such a style semantics uses the inference rules (the "uses" of the connectives) as the origin of meaning and then uses the meta-theory of, for example, sequent calculus as the formal setting for organizing that meaning. Girard's famous slogan "From the rules of logic to the logic of rules" [20] is another illustration of the switch from looking for the semantics of inferences to looking that the semantics offered by inference rules. Here, we shall develop a plan proposed in [26] to exploit proof theory as a vehicle for describing semantics within the broader setting of computation.

Example: To illustrate the immediateness of using proof theory to provide meaning, we present a simple example. Alan Turing encoded computation using strings, machines, and computation traces. He used these then to reason about the power of computing via standard enough mathematical techniques (inductive definitions, set-theoretical constructions, encodings as functions, etc). While this mathematical framework was, of course, highly appropriate for his particular goals of proving the first theorems about limitations of computation, that framework has not served us well when we wish to reason about the meaning of specific computations. In the proof theory approach to relational programming, computation can be described using terms, formulas, and cut-free proofs. On one hand, such cut-free proofs encode computation traces in much the same way as Turing's computation traces: however, there is a great deal of structure and many formal results surrounding sequent calculus proofs. These proof theory results make it possible to reason richly about computation using devices involving abstractions, substitutions, and cut-elimination [27].

6.5. Exploiting what proof theory does not resolve canonically

While proof theory provides a remarkably robust and deep analysis of abstraction, substitution, and duality, there are several computational phenomena that it alone does not provide information. For examples, proof theory does not offer canonical treatments of first-order quantification, the structure of worlds in modal logics, focusing polarity of atomic formulas, and the exponentials. Such non-canonical aspects of proof theory can often be exploited by the computer scientist. For example, first-order quantification is richly applied in a wide range of applications with greatly varying domains of quantification. Changes in the assignment of focusing bias for atomic formulas in proof search allows one to mix forward-chaining and backward-chaining style proof search to suit applications [23]. Linear logic introduced

the exponentials in order to account for unbounded behaviors in logic but those exponentials are not given a canonical treatment by any standard proof theory techniques [11]. Thus non-canonicity allows one to introduce the notion of *subexponentials* [33] which, in turn, provides a notion of "locality" to logic.

7. FOCUSED PROOF SYSTEMS

One of the most exciting developments in recent years within proof theory involves *focused proof system*. While "focusing-like" phenomena have been observed in proof systems back in the mid-1980s [28], it was Andreoli's focused proof system [3] for linear logic [18] that really transformed the topic. The earlier results were rather limited, both in their "focusing behavior" and in the subsets of logic to which it could be applied. Andreoli's result, however, applied to a full and rich logic.

For the sake of concreteness, we present an example of a focused proof system. In particular, the LJF focused proof system for intuitionistic logic is given in Figure 1. To understand this proof system, we need to define what is meant by *polarity* in intuitionistic logic. Atoms in LJF are arbitrarily divided between those that are positive and those that are negative. *Positive formulas* are of the following forms: positive atoms, *true*, *false*, $A \wedge^{+} B$, $A \vee B$ and $\exists x A$. *Negative formulas* are among negative atoms, $A \wedge^{-} B$, $A \supset B$ and $\forall x A$. Sequents in *LJF* can be interpreted as follows:

1. The sequent $[\Gamma], \Theta \longrightarrow \mathcal{R}$ is an *unfocused sequent*. Here, Γ and Θ are both multisets and Γ contains only negative formulas and positive atoms. The symbol $\mathcal{R}$ denotes either the formula R or the "bracketed" formula $[R]$. End sequents of LJF proofs usually have the form $[], \Theta \longrightarrow R$.
2. The sequent $[\Gamma] \longrightarrow [R]$ is a special case of the previous sequent in which Θ is empty (and, hence, not written) and $\mathcal{R}$ is of the form $[R]$. Such sequents denote the end of the asynchronous phase: proof search continues with the selection of a focus.
3. The sequent $[\Gamma] \xrightarrow{A} [R]$ represents *left-focusing* on the formula A. Provability of this sequent is related to the provability of $\Gamma, A \vdash_I R$.
4. The sequent $[\Gamma] -_A\!\!\rightarrow$ represents *right-focusing* on the formula A. Provability of this sequent is related to provability of the sequent $\Gamma \vdash_I A$.

There are several things to observe about this proof system.

1. Proof search (reading the inference rules from conclusion to premise) groups all invertible inference rules into one "phase", that is, a grouping of inference rules all without a focus (either left or right).
2. When no invertible rules remain then a formula from either the left or right-hand-side is picked as the *focus*: a focused phase is the result of performing introduction rules on the focused formula.
3. The initial rules are only available with focused sequents.
4. Asynchronous formulas on the left are treated linearly (they are deleted once used) whereas the only formulas that are contracted are negative formulas on the left.
5. Completeness for *LJF* is stated as follows: If $\vdash_I B$ then for every possible assignment of polarity to atoms, the sequent $[\cdot] \longrightarrow B$ has a proof in *LJF*.

Polarity assignment to atoms does not affect provability but it can have an important consequence on the size and shape of proofs. For example, consider the Horn clause specification of the Fibonacci series:

$$\{\textit{fib}(0,0), \textit{fib}(1,1),$$

$$\forall n \forall f \forall f' [\textit{fib}(n,f) \supset \textit{fib}(n+1,f') \supset \textit{fib}(n+2, f+f')\}.$$

If all atomic formulas are given a negative bias, then there exists only one focused proof of $\textit{fib}(n, f_n)$: this one can be classified as a "backward chaining" proof and its size is exponential in n. On the other hand, if all atomic formulas are given a positive bias, then there is an infinite number of focused proofs all of which are classified as "forward chaining" proofs: the smallest such proof is of size linear in n.

This one focused proof system is rather general: it subsumes all other known focused proofs systems for intuitionistic logic in the literature. To illustrate how this proof system can account for other proof systems, consider only the implicational fragment of intuitionistic logic. Figure 2 derives the rules for this fragment under the assumption that all atomic formulas are negative while Figure 3 derives the rules for this fragment under the assumption that all atomic formulas are positive. The proof system with negative atoms not only describes "goal-directed", "top-down" proofs, it can also be used to describe simply typed λ-terms that are in head-normal form. On the other hand, the proof system with positive atoms not only describes "program-directed", "bottom-up" proofs, it can also be used to describe simply typed λ-terms that are in administrative normal form [16].

When important new insights into the foundations of a topic (here, computational logic) are discovered, one expects to see many new results ripple out from that insight. A few of these new developments based on focused proofs are outlined below. In general, we work on focused proofs system as a cornerstone in our approach to describing a unity to computational logic. We outline our major rationale for this perspective.

Decision and Reaction Rules

$$\frac{[N,\Gamma] \xrightarrow{N} [R]}{[N,\Gamma] \longrightarrow [R]}\ Lf \qquad \frac{[\Gamma] -_P\!\rightarrow}{[\Gamma] \longrightarrow [P]}\ Rf \qquad \frac{[\Gamma], P \longrightarrow [R]}{[\Gamma] \xrightarrow{P} [R]}\ R_l \qquad \frac{[\Gamma] \longrightarrow N}{[\Gamma] -_N\!\rightarrow}\ R_r$$

$$\frac{[C,\Gamma],\Theta \longrightarrow \mathcal{R}}{[\Gamma],\Theta, C \longrightarrow \mathcal{R}}\ []_l \qquad \frac{[\Gamma],\Theta \longrightarrow [D]}{[\Gamma],\Theta \longrightarrow D}\ []_r$$

Initial Rules

$$\frac{}{[P,\Gamma] -_P\!\rightarrow}\ I_r,\ \text{atomic } P \qquad \frac{}{[\Gamma] \xrightarrow{N} [N]}\ I_l,\ \text{atomic } N$$

Introduction Rules

$$\frac{}{[\Gamma],\Theta, false \longrightarrow \mathcal{R}}\ falseL \qquad \frac{[\Gamma],\Theta \longrightarrow \mathcal{R}}{[\Gamma],\Theta, true \longrightarrow \mathcal{R}}\ trueL \qquad \frac{}{[\Gamma] -_{true}\!\rightarrow}\ trueR$$

$$\frac{[\Gamma] \xrightarrow{A_i} [R]}{[\Gamma] \xrightarrow{A_1 \wedge^- A_2} [R]}\ \wedge^- L \qquad \frac{[\Gamma],\Theta \longrightarrow A \quad [\Gamma],\Theta \longrightarrow B}{[\Gamma],\Theta \longrightarrow A\ \wedge^-\ B}\ \wedge^- R$$

$$\frac{[\Gamma],\Theta, A, B \longrightarrow \mathcal{R}}{[\Gamma],\Theta, A\ \wedge^+\ B \longrightarrow \mathcal{R}}\ \wedge^+ L \qquad \frac{[\Gamma] -_A\!\rightarrow \quad [\Gamma] -_B\!\rightarrow}{[\Gamma] -_{A\ \wedge^+ B}\!\rightarrow}\ \wedge^+ R$$

$$\frac{[\Gamma],\Theta, A \longrightarrow \mathcal{R} \quad [\Gamma],\Theta, B \longrightarrow \mathcal{R}}{[\Gamma],\Theta, A \vee B \longrightarrow \mathcal{R}}\ \vee L \qquad \frac{[\Gamma] -_{A_i}\!\rightarrow}{[\Gamma] -_{A_1 \vee A_2}\!\rightarrow}\ \vee R$$

$$\frac{[\Gamma] -_A\!\rightarrow \qquad [\Gamma] \xrightarrow{B} [R]}{[\Gamma] \xrightarrow{A \supset B} [R]}\ \supset L \qquad \frac{[\Gamma],\Theta, A \longrightarrow B}{[\Gamma],\Theta \longrightarrow A \supset B}\ \supset R$$

$$\frac{[\Gamma],\Theta, A \longrightarrow \mathcal{R}}{[\Gamma],\Theta, \exists y A \longrightarrow \mathcal{R}}\ \exists L \qquad \frac{[\Gamma] -_{A[t/x]}\!\rightarrow}{[\Gamma] -_{\exists x A}\!\rightarrow}\ \exists R \qquad \frac{[\Gamma] \xrightarrow{A[t/x]} [R]}{[\Gamma] \xrightarrow{\forall x A} [R]}\ \forall L \qquad \frac{[\Gamma],\Theta \longrightarrow A}{[\Gamma],\Theta \longrightarrow \forall y A}\ \forall R$$

Figure 1: *The Intuitionistic Sequent Calculus LJF. Here, P is positive, N is negative, C is a negative formula or positive atom, and D a positive formula or negative atom. Other formulas are arbitrary. Also, y is not free in Γ, Θ, or R.*

$$\frac{[\Gamma] \longrightarrow [A]}{[\Gamma] \longrightarrow A}\ []_r \qquad \frac{[B,\Gamma] \xrightarrow{B} [A]}{[B,\Gamma] \longrightarrow [A]}\ Lf \qquad \frac{}{[\Gamma] \xrightarrow{A} [A]}\ I_l,\ \text{atomic } A$$

$$\frac{[\Gamma] \longrightarrow B \quad [\Gamma] \xrightarrow{C} [R]}{[\Gamma] \xrightarrow{B \supset C} [R]}\ \supset L, R_r \qquad \frac{[\Gamma, B] \longrightarrow C}{[\Gamma] \longrightarrow B \supset C}\ \supset R, R_l$$

Figure 2: *The implicational fragment of LJF with negative atoms. Here, A is atomic.*

$$\frac{[\Gamma] \longrightarrow [A]}{[\Gamma] \longrightarrow A}\ []_r \qquad \frac{[\Gamma, B \supset C] \xrightarrow{B \supset C} [A]}{[\Gamma, B \supset C] \longrightarrow [A]}\ Lf \qquad \frac{}{[\Gamma, A] \longrightarrow [A]}\ Rf, I_r$$

$$\frac{[\Gamma] \longrightarrow B \qquad [\Gamma] \xrightarrow{C} [A]}{[\Gamma] \xrightarrow{B \supset C} [A]}\ \supset L, R_l \qquad \frac{[\Gamma, A] \longrightarrow B}{[\Gamma] \longrightarrow A \supset B}\ \supset R, []_l$$

Figure 3: *The implicational fragment of LJF with positive atoms. Here, A is atomic.*

Engineering synthetic connectives and macro-rules Focused proof systems provide a way to "polarize" logical connectives into *invertible* and *non-invertible* phases. The key observation here is that these phases can be seen not as a sequence of "micro" rules but as a single, indivisible new "macro" inference rule. These macro rules then introduce "synthetic connectives". As a result, focused proof systems allow one to redesign logic into new sets of connectives and inference rules. The striking thing about this possibility is that, if one follows the rather flexible rules for polarization, the resulting set of macro rules remains sound and complete with respect to the original set of micro rules: at the same time, important proof theoretic properties such as cut-elimination are also maintained [24]. Thus, one can "re-engineer" a logic in a number of different ways in order to satisfy a range of different needs. For example, the differences between bottom-up reasoning (as in deductive databases) and top-down reasoning (as in logic programming) can be seen as simply differences of polarization. In the richest setting of focusing for, say, intuitionistic or classical logic [23], it is possible to also mix these two extremes in search in rich ways.

Algorithm specifications Many high-level specifications of algorithms rely on non-deterministic operations: *e.g.*, pick any member of a multiset, select any reachable state, etc. If one wants to encode such specifications into logic, one finds that there is much too much non-determinism within conventional proof systems for this to be possible. If, instead, one uses a focused proof system, one can often pick inference rules that exactly match the non-determinism in the algorithmic specification. In this way, it should be possible to bring a much more tight connection between logic and algorithm specification than has been possible before.

Game semantics The duality in focused proof system between invertible (negative) and non-invertible (positive) phases, is similar to the duality in two-player, dialog games [25] used in proof theory in order to provide a semantics for proofs. When considering the moves of your opponent, you have no choice (corresponds to invertibility) since you must consider all possible moves: when considering your own move, you have a choice and you might get it wrong (corresponds to non-invertibility). Connecting focusing in multiplicative and additive linear logic with dialog games has also made it possible to show that proof and refutation can be seen as perfectly dual [14, 30]. Girard has taken this basic duality of focused proof construction and used it to provide an entirely new formulation of inference rules and of logic, giving rise recently to his program on *Ludics* [20]. It should be possible to extend these notions of games and the duality of proof and refutation (or proof and counterexample) to the many different situations where focusing proof systems are employed.

8. THE UNITY OF PROOF SYSTEMS

A great deal of recent work in applying proof theory to the foundations of computational logic systems reveals that there are significant and serious chances for providing unity to a range of proof system for, at least, classical, intuitionistic, and linear logics.

Alternative approaches to unbounded behavior While Girard's original proposal to extend the core of linear logic (MALL) with the exponentials (?, !) in order to achieve unbounded behaviors is elegant, especially in its simplicity, this approach has some deficiencies. In particular, the use of exponentials breaks up focused proofs into smaller and smaller phases, thus negating the applicability of focusing in the first place. There have been at least a couple of recent alternative proposals considered to the use of exponentials. For example, Baelde and Miller [7, 5] proposed adding least and greatest fixed points to MALL directly. The resulting logic is surprisingly elegant and expressive. MALL plus fixed points has been used to describe the logical foundations of a model checker and is being used to design a new theorem proving architecture for the search for proofs involving induction. Liang and Miller [24] have blended together classical logic (which has natural notions of unbounded behavior) and MALL. The resulting logic is an exciting new approach to the Unity of Logic [19], one where we can retain focusing behavior for classical, intuitionistic, and linear logics.

Logic programming vs model checking vs theorem proving The differences between these three activities can be characterized by their different uses of fixed points. Logic programming involves *may* behavior only: is it possible to unfold a fixed point and non-deterministically pick a path to a success. On the other hand, both model checking and theorem proving deal with *must* as well as *may* behavior. These two differ in that model checkers generally assume finite fixed points while (inductive) theorem provers allow fixed points to have infinite unfoldings. Given these rough descriptions, it is possible to see rich ways that these activities can fit together into one system (and one logic!) and enhance each other: for example, a theorem prover might prove certain symmetry lemmas (via induction) and these could be used in the model checker to greatly reduce search space. Similarly, tabling within model checkers can be seen as lemma generation in theorem provers [29].

Many proof systems just differ in polarization Miller and Nigam [32] have recently shown that a range of commonly used proof systems (sequent calculus, natural deduction, tableaux, Hilbert-style, etc) are, in fact, just different polarization of a common specification for inference rules. Thus it should be possible to

create a single, formal framework for specifying and implementing many proof systems.

Accounting for rewriting proofs Can we view rewriting and reasoning with functions as naturally fitting within a relational setting. Functions can, of course, encode relations using set-valued functions and different kinds of order relations (Plotkin/Hoare/Smyth). Conversely, relations can directly encode functions: the graphs of functions are, simply, graphs of relations. It is also interesting to note that the restriction on relations that make them into functions (for all input values there is a *unique* output value) has a precise connections to focusing: in particular, if the binary predicate P represents a function (first argument denoting the input and the second argument, the output), then the formulas

$$\forall x.P(t,x) \supset Q(x) \quad \text{and} \quad \exists x.P(t,x) \wedge Q(x)$$

are logically equivalent: the underlying P-typed quantifier is, in fact, self dual. This observation immediately relates to how one can structure focused proofs.

Model-checking-as-deduction, deduction-as-model-checking As we have mentioned, many (high-level aspects of) model checking can be seen as focused deduction with fixed points [6, 7, 5]. Other recent work [14] has shown that (in certain weak logics), deduction can be achieved by looking for winning strategies in suitable games. Of course, the search for winning strategies is a typical and important example of something model checkers can do well. These two lines of research make it possible to hope that model checkers and theorem provers might ultimately be seen as sharing many common features that might allow their implementations to be tightly integrated.

9. SYNTHESIS

While it is tempting to declare that *logic-in-computer-science* is an *applied science*, it is embarrassing to look at the many splinters that form the core of that would-be science. We have proposed proof theory as a framework for integrating a great deal of the work on computational aspects of classical, intuitionistic, and linear logics.

So far we have not mentioned model theoretic semantics, a common topic used to justify the design and applications of logic. Any serious and deep formalism, such as logic, should have *multiple* explanations. Classical and intuitionistic first-order logics have been given appealing and natural model theories using models due to Gödel, Tarski, and Kripke. Linear logic has provided fresh new aspects of logic for which numerous researchers are currently developing semantics along numerous lines: stable models, phase semantics, categorical semantics, game semantics, etc. Attempting to develop such model theoretic notions is certainly an important activity to do along side of developing a more unified approach to proof.

10. SOME SPECIFIC CHALLENGES

Specialization and compartmentalization will continue to be important activities from both an industrial and academic point-of-view. But we must ask for more: we should insist also on the *unity of logic* from which one would expect deep new insights into the foundations of computer science and greatly improved and integrated tools for dealing with the correctness of software and hardware systems. We conclude by listing some specific challenges.

Challenge 1: Unify a wide range of logical features into a single framework. How best can we explain the many enhancements that have been designed for logic: for example, classical / intuitionistic / linear, fixed points, first-order / higher-order quantification, modalities, and temporal operators? Can we explain these as involving orthogonal compositions as is the case for quantification and classical propositional connectives?

Challenge 2: Unify a wide spectrum of proof systems. Computer logic systems often build, explicitly or implicitly, some kinds of proof systems, based on, for example, sequent calculus, natural deduction, tableaux, Hilbert-style proof, resolution refutations, DPLL-trees, tabled deduction, matrix-based proofs, rewriting, etc. There is strong, recent evidence that several of these style of proof systems can be accounted for uniformly within a single (focused) proof system [24, 29, 32, 34]. Can a single, declarative proof checker be built that can check all of these forms of proofs?

Challenge 3: Unify the disciplines of theorem proving, model checking, and computation. Although these disciplines can all be viewed as certain kinds of deduction in a logic with fixed points, the literature and systems behind these disciplines are wildly different. Integrating these systems is known to be of great importance but most research focuses on *ad hoc* engineering solutions. Can we develop a deeper and more principled integration?

Challenge 4: Design new architectures for supporting a wide range of deduction techniques within a single, integrated framework. A great number of algorithms and data structures have been developed to build working model checkers and theorem provers. These different domains share little in common. If we can establish common proof theoretic explanations of these different activities, can we also develop common, universally agreed upon implementation architectures and techniques than can be shared across these activities?

Acknowledgments I wish to thank the following people for valuable discussions related to the topic of this paper: David Baelde, Olivier Delande, Chuck Liang, Lutz Straßburger, Gopalan Nadathur, Vivek Nigam, Alexis Saurin, Alexandre Viel, as well as my colleagues on the REDO project, namely, Alessio Guglielmi, François Lamarche, and Michel Parigot.

REFERENCES

[1] S. Abramsky. Sequentiality vs. concurrency in games and logic. *Mathematical Structures in Computer Science*, 13(4):531–565, Aug. 2003.

[2] S. Abramsky, D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Applying game semantics to compositional software modeling and verification. In K. Jensen and A. Podelski, editors, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 421–435. Springer, 2004.

[3] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.

[4] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. A compact kernel for the calculus of inductive constructions. *Sādhanā*, 34:71–144, 2009.

[5] D. Baelde. *A linear approach to the proof-theory of least and greatest fixed points*. PhD thesis, Ecole Polytechnique, Dec. 2008.

[6] D. Baelde, A. Gacek, D. Miller, G. Nadathur, and A. Tiu. The Bedwyr system for model checking over syntactic expressions. In F. Pfenning, editor, *21th Conference on Automated Deduction (CADE)*, number 4603 in LNAI, pages 391–397. Springer, 2007.

[7] D. Baelde and D. Miller. Least and greatest fixed points in linear logic. In N. Dershowitz and A. Voronkov, editors, *International Conference on Logic for Programming and Automated Reasoning (LPAR)*, volume 4790 of *LNCS*, pages 92–106, 2007.

[8] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[9] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[10] T. Coquand and G. Huet. *Constructions: A Higher Order Proof System for Mechanizing Mathematics*, volume 203 of *EUROCAL85, Springer-Verlag LNCS*, pages 151–184. Springer-Verlag, Linz, 1985.

[11] V. Danos, J.-B. Joinet, and H. Schellinx. The structure of exponentials: Uncovering the dynamics of linear logic proofs. In G. Gottlob, A. Leitsch, and D. Mundici, editors, *Kurt Gödel Colloquium*, volume 713 of *LNCS*, pages 159–171. Springer, 1993.

[12] M. Davis. Influences of mathematical logic on computer science. In R. Herkin, editor, *The Universal Turing Machine: A Half-Century Survey*, pages 315–326. Oxford University Press, Oxford, 1988.

[13] N. G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on Automatic Demonstration*, pages 29–61. Lecture Notes in Mathematics, 125, Springer-Verlag, 1970.

[14] O. Delande and D. Miller. A neutral approach to proof and refutation in MALL. In F. Pfenning, editor, *23th Symp. on Logic in Computer Science*, pages 498–508. IEEE Computer Society Press, 2008.

[15] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. W. de Bakker and J. van Leeuwen, editors, *Proceedings of the 7th International Colloquium on Automata, Languages and Programming, ICALP'80 (Noordwijkerhout, NL, July 14-18, 1980)*, volume 85 of *LNCS*, pages 169–181. Springer, 1980.

[16] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. *ACM SIGPLAN Notices*, 28(6):237–247, 1993.

[17] H. Geuvers. Proof assistants: History, ideas and future. *Sādhanā*, 34:3–25, 2009.

[18] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[19] J.-Y. Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[20] J.-Y. Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, June 2001.

[21] J. Y. Halpern, R. Harper, N. Immerman, P. G. Kolaitis, M. Y. Vardi, and V. Vianu. On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic*, 7(1):213–236, Mar. 2001.

[22] R. Kahle and P. Schroeder-Heister. Introduction to proof theoretic semantics. *Special issue of Synthese*, 148, 2006.

[23] C. Liang and D. Miller. Focusing and polarization in intuitionistic logic. In J. Duparc and T. A. Henzinger, editors, *CSL 2007: Computer Science Logic*, volume 4646 of *LNCS*, pages 451–465. Springer, 2007.

[24] C. Liang and D. Miller. A unified sequent calculus for focused proofs. In *LICS: 24th Symp. on Logic in Computer Science*, pages 355–364, 2009.

[25] P. Lorenzen. Ein dialogisches konstruktivitätskriterium. In *Infinitistic Methods: Proceed. Symp. Foundations of Math.*, pages 193–200. PWN, 1961.

[26] D. Miller. Proof theory as an alternative to model theory. *Newsletter of the Association for Logic Programming*, Aug. 1991. Guest editorial.

[27] D. Miller. A proof-theoretic approach to the static analysis of logic programs. In *Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday*, number 17 in Studies in Logic, pages 423–442. College Publications, 2008.

[28] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[29] D. Miller and V. Nigam. Incorporating tables into proofs. In J. Duparc and T. A. Henzinger, editors, *CSL 2007: Computer Science Logic*, volume 4646 of *LNCS*, pages 466–480. Springer, 2007.

[30] D. Miller and A. Saurin. A game semantics for proof search: Preliminary results. In *Proceedings of the Mathematical Foundations of Programming Semantics (MFPS05)*, number 155 in Electronic Notes in Theoretical Computer Science, pages 543–563, 2006.

[31] R. A. D. Millo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Communications of the Association of Computing Machinery*, 22(5):271–280, May 1979.

[32] V. Nigam and D. Miller. Focusing in linear metalogic. In *Proceedings of IJCAR: International Joint Conference on Automated Reasoning*, volume 5195 of *LNAI*, pages 507–522. Springer, 2008.

[33] V. Nigam and D. Miller. Algorithmic specifications in linear logic with subexponentials. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 129–140, 2009.

[34] V. Nigam and D. Miller. A framework for proof systems. Extended version of IJCAR08 paper. Submitted, Mar. 2009.

[35] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[36] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[37] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.

[38] J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12:23–41, Jan. 1965.

[39] A. Trybulec and H. A. Blair. Computer aided reasoning. In R. Parikh, editor, *Proceedings of the Conference on Logic of Programs*, volume 193 of *LNCS*, pages 406–412, Brooklyn, NY, June 1985. Springer.