Finding Unity in Computational Logic

Dale Miller
INRIA Saclay – Île-de-France 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 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 the proof theory of the sequent calculus: recent results in the area of focused proof systems will play a central role in developing this unity.

Computational logic, unity of logic, proof theory, sequent calculus, focused proof systems

1. SOFTWARE AND HARDWARE CORRECTNESS IS CRITICALLY IMPORTANT

Computer systems are everywhere in our society and their integration with all parts of our lives is constantly increasing: accompany this wide scale use of computing systems comes an increasing need to deal with their correctness. There are a host of computer systems—such as those in cars, airplanes, missiles, hospital equipment—where correctness of software is paramount. Big changes in the attitude towards correctness is also taking place in the area of consumer electronics. For example, years ago, establishing the correctness of, say, desktop PCs, music players, and telephones was not urgent since rebooting such systems to recover from errors or living without a feature due to bugs were mostly nuisances and not "life-threatening." But today, these same devices are now tightly integrated into networks and, hence, they must deal with information security and user anonymity while trying to keep safe from malicious software.

Attempting to establish various kinds of correctness-related properties of software systems is no longer an academic curiosity. The old chestnut "You can't build a tall building on a sandy beach," which is so often invoked to argue for solid foundations for engineering projects, needs a modern updating that requires moving off the beach to the sea: "If you are in a canoe, a small leak might be okay; 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. 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 elements of the world in William Gibson's *Virtual Light*, where network security was so bad that important data was transferred by bikers carrying hard-disks! If we cannot produce software that has some formal guarantees, then the development of all the new features and services—and the concomitant increases in efficiency and productivity—that we all hope to see soon will be greatly delayed.

2. LOGIC IS A KEY

It is to logic that researchers, 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 surrounding it. Logic can also play a number of roles viz-a-viz software systems. For example, logic can be used to formally establish correctness (*e.g.*, proving a program correct). Given its universal character, it can also be used as a language for communicating meaning between different entities. For example, designers of programming languages often use logic-based formalisms to communicate the precise semantics of their designs to users or language implementers. Also, when machines exchange data and programs, increasing use is made of logic-based expressions such as types, memory layout specifications, assertions, interface requirements, proof certificates (*e.g.*, in the proof carrying code setting), etc.

Proceedings of ACM-BCS Visions of Computer Science 2010

Indeed, it has been variously argued that logic plays a role in computer science similar to that played by the calculus of Newton and Leibniz in the physical sciences and engineering disciplines (20). Twenty years ago, Martin Davis (10) observed that mathematical logic had already formed 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.
- The number of adjectives that are routinely added to the word "logic" is extensive: first-order, higher-order, classical, intuitionistic, linear, modal, temporal, 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 in regular used and development: model checkers, interactive and automatic theorem provers, logic circuit simulators/testers, type inference systems, SAT solvers, 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 the algebraic 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. However, 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 computation logic as organized around a core set of concepts. We shall offer such a set of concepts.

4. A BRIEF HISTORY

We provide a quick overview of the origins and state-of-the-art of computational logic.

4.1. The early dreams for logic

One of the first lessons that one learns about logic is that it is *secure* and *universal*. These attributes were part of Leibniz's hope for a universal, formal language in which all disputes could be answered simply by "Let us calculate . . . and see who is right" (22). The first formal system that could claim both security of deduction (soundness) and universality (completeness) was first-order logic which can be formalized using both syntactic means (proofs) and semantic means (truth). On such a secure foundations, one can build set theory and much of mathematics. An early and natural goal for computational logic was the following:

> Let us implement logic (and why not, since its syntactic side is organized around a small set of proof principles): we shall then have an implementation of mathematics.

Early dreams in automated reasoning focused on simple and theoretically complete methods with the hope that they would be complete in practice. The hope was to have one framework, one implementation, and universal

applicability. We describe two threads in deploying universal methods for logic.

Automated and interactive provers In the 1960-1970's, there was a great deal of work done on implementing automated systems for first-order logic that were based on such complete paradigms as resolution (37) or on conditional rewriting (7). Such early work produced a great deal of information about proof strategies and methods to implement formal logical systems effectively (unification, backtracking search, etc). Another lesson was, however, disappointing: those systems came no where near to achieving the ambitions of effective, universal deployment. Furthermore, it was clear that the usual speed up in program execution that resulted from improvements in hardware and compilers would not make a dent in the "state-explosion" that occurred within such provers. Starting around the same time (and continuing today), a number of interactive proof environments for mathematics were developed: for example, Nqthm (7), Mizar (39), Coq (9), NuPRL (8), HOL (19), PVS (33), and Matita (3). Most of these systems chose either first-order logic with induction or a higher-order constructive logic based on intuitionistic logic as a suitable framework for encoding mathematics. The architecture of several of these systems were influenced directly or indirectly by Automath (11) and LCF (34). While interaction was central to the functioning of such systems, they all allowed automation to some extent, often using tactics and tacticals.

Model checkers and logic programming By shifting one's attention 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 (13; 36). While great successes can be claimed for such systems, the hope of having a universal approach to model checking quickly ran into the state explosion problem. Logic programming can be used to explore membership in still weaker relations. Prolog 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. Ultimately, the dream of deductively describing a relation and then getting an effective implementation of it turned out to be largely illusory.

4.2. Specialization and fracturing

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 a domain 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. Unfortunately, success in one narrow topic is seldom translated to a success in another topic.

Working within frameworks In (35), 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 theorem provers 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 design new logics (as is the case in Isabelle) or extend 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 a monolithic treatment

The universal applicability of logic and its associate proof methods for computer system verification has also been attacked from another angle. De Millo, Lipton, and Perlis (29) 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, how the undecidability of the halting problem lead 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 proof systems 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 formal properties can and has been applied to aid such communications. Not all things, of course, *are* logic but logic offers an extremely valuable aid in defining, formalizing, automating, and communicating many aspects of software systems. Thus, the impossibility of using one logic and one formal method leads to the need to have many specialized logics and methods.

5. METHODOLOGY

Our approach to unity for computational logic is based on the follow main methodological points.

5.1. Dropping mathematics as an intermediate

This proposal deals with the role of logic in 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, consider the following.

The traditional approach to reasoning about computation in almost all ambitious frameworks 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.

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 of this proposal is that we shall drop mathematics as an intermediate and attempt to find more direct and intimate connections between computation, reasoning, and logic. The main problem with having mathematics in the middle seems to be that many aspects of computation are rather "intensional" but a mathematical treatment requires an extensional encoding. The notion of *algorithm* is an example 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 intensional default can be managed instead, then function values are secondary (usually 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 for the w in $\lambda x.w$ can yield $\lambda x.x$.

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 treat intensionality, but experience with such treatments demonstrate that they do not reach the level of "canonicity" that is reached by similar encoding techniques that have been successful for functions and sets. We shall, instead, look for logical and proof theoretic treatments of many of these intensional aspects of computation.

5.2. Proof theory provides a framework

We shall propose using *proof theory*, particularly the proof theory of the *sequent calculus*, as the unifying framework behind computational logic. Proof theory seems to be far more intimately related to computation given its reliance on (mostly) finitary methods: this is in contrast, say, to the mathematics of model theory where there are generally infinitely many models to consider and these models generally have infinite extension. Proof theory provides elegant treatments of various intensional aspects of computation (for example, resources in linear logic) and provides some rich and flexible avenues of reasoning that go along way in providing a rich and flexible framework.

Gentzen (15) invented the sequent calculus to solve a problem in the unity of logic. Gentzen wished to prove that both classical and intuitionistic logic satisfied the *Hauptsatz*, namely, that inference did not need to have detours or lemmas. His goal was to prove this one result for both logics simultaneously. After failing to prove such a meta-theorem using natural deduction, he invented the sequent calculus along with the important notions of *structural rules* and *cut-elimination* (we provide a primer for the sequent calculus in Section 6). He was then able to describe one procedure for the elimination of the cut rule (corresponding to the in-lining of the proof of lemmas) that proved the Hauptsatz (now stated as the admissibility of the cut-rule) for both logics simultaneously.

With the introduction of *linear logic* (16), Girard has expanded on these themes of using the sequent calculus, the structural rules, and cut-elimination. As a result the expressiveness and utility of the sequent calculus for computational purposes has been greatly extended. It is this setting that we propose to use for an explicit attempt at unifying much of computational logic.

5.3. Logic considered broadly but with a standard

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, "Beauty" in (14)

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 (20); and, finally, logic is the *mean of many extremes* given its intimate use in a range of "extremes" including databases, programming languages, type systems, certificates, verification, 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 many ways that these can be elaborated (*eg.*, propositional versus quantificational, first-order versus higher-order, with or without equality) and there are many proof systems for these logics (sequent, natural deduction, tableaux, Hilbert-style, matrix methods, *etc.*).

5.4. A big logic with many sublogics

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 (6), and subexponentials (31)) 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 (17; 24).

5.5. Two-levels of logic

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. A PRIMER FOR PROOF THEORY AND SEQUENT CALCULUS

We shall assume that the reader has some familiarity with the sequent calculus. Below we recall some definitions and results.

Sequents are generally presented as a pair $\Gamma \vdash \Delta$ of two (possibly empty) collections of formulas. For Gentzen, these collections were lists but multisets and sets are also used. Such sequents are also called *two-sided* sequents. Intuitively, the formulas on the left-hand-side (in Γ) are viewed as assumptions and formulas on the right-hand-side (in Δ) are viewed as possible conclusions: thus, an informal reading of the judgment described by the sequent $\Gamma \vdash \Delta$ is "if all the formulas in Γ are true then some formula in Δ is true." This reading, however, is only suggestive since sequents will be used in settings where the notion of truth is either not present

or is meant to be developed independently. In calculi with an involutive negation (such as classical logic), two-sided sequents are often replaced by simpler *one-side* sequents: in particular, the sequent $\Gamma \vdash \Delta$ is written, instead, as $\vdash \neg\Gamma, \Delta$ (placing $\neg$ in front of a collection of formulas is taken as the collection of negated formulas). In the remainder of this paper, we shall assume that the collections of formulas used in sequents are multisets.

A formula is *atomic* if its top-level constant is a non-logical symbol, namely, a predicate symbol. A *literal* is an atom or a negated atom.

6.1. Three classes of inference rules

Sequent calculus proof systems come with inference rules in which a sequent is the conclusion and zero or more sequents are premises. These rules are usually broken down into three classes of rules. The *structural rules* are the following two, applied to either the left or right side.

$$\text{Contraction:}\quad \frac{\Gamma, B, B \vdash \Delta}{\Gamma, B \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, B, B}{\Gamma \vdash \Delta, B}$$

$$\text{Weakening:}\quad \frac{\Gamma \vdash \Delta}{\Gamma, B \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, B}$$

The *identity rules* are also just two, called Initial and Cut.

$$\frac{}{B \vdash B}\ Initial \qquad \frac{\Gamma_1 \vdash \Delta_1, B \quad \Gamma_2, B \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ Cut$$

The meta-theory of most sequent calculus presentations of logic includes results that say that most instances of these identity rules are, in fact, not necessary. The cut-elimination theorem states that removing the cut-rule does not change the set of provable sequents. Furthermore, the initial rule can usually be eliminated for all cases except when B is an atomic formula.

The third and final collection of inference rules are the *introduction rules* which describe the role of the logical connectives in proof. In two-sided sequent calculus proofs, these are usually organized as right and left introduction rules for the same connective. In one-sided sequent calculus proofs, these rules are organized as right-introduction rules for a connective and its De Morgan dual. For example, here are two examples of pairs of introduction rules.

$$\frac{\Gamma, B_1, B_2 \vdash \Delta}{\Gamma, B_1 \wedge B_2 \vdash \Delta} \qquad \frac{\Gamma_1 \vdash \Delta_1, B_1 \qquad \Gamma_2 \vdash \Delta_2, B_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, B_1 \wedge B_2}$$

$$\frac{\Gamma, B[t/x] \vdash \Delta}{\Gamma, \forall x B \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, B[y/x]}{\Gamma \vdash \Delta, \forall x B}$$

The right-introduction rule for $\forall$ has the proviso that the *eigenvariable* y does not have a free occurrence in any formula in $\Gamma \cup \Delta$. Notice that in both of these sets of rules, there is exactly one new occurrence of a logical connective in the premise when compared to the premises.

Notice that some inference rules are invertible: that is, if their conclusion is provable then all their premises are provable. Of the four introduction rules above, the left rule for $\wedge$ and the left rule for $\forall$ are invertible: the other two introduction rules are not necessarily invertible.

When presenting a sequent calculus proof system for a specific logic, one usually presents the introduction rules for the logical connectives of the logic and usually accepts both identity inference rules (initial and cut). The structural rules are, however, seldom adopted without restriction. For example, intuitionistic logic can be understood as a two-sided sequent calculus in which the contraction-right rule is not allowed. Multiplicative-additive linear logic (MALL) admits neither weakening nor contraction and full linear logic allows those structural rules only for specially marked formulas (formulas marked with the so-called *exponentials* ! and ?). Classical logic, however, generally admits the structural rules unrestricted.

6.2. Capturing computation via proof search

When considered abstractly, the operational semantics of functional programming can be viewed as a systematic process for eliminating cuts from a proof. (In certain sequent calculi for intuitionistic logic, there are nice connections to draw between cut-elimination and β-reduction (38).) Our framework here for unity involves, instead, using the *proof search* approach to computational specification. With this approach, one considers the dynamics of computation as the process of attempting to build a cut-free proof from conclusion to premises. As one moves up a proof tree, sequents change and that change captures the dynamics of computation. The cut rule and the cut-elimination theorem is generally used not as part of computation but as a means to reason about computation. The proof search approach to specification has been used to formalize the operational semantics of logic programming (27).

6.3. Proof theory as an 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 (21). Such a style semantics uses the inference rules (the "uses" of the connectives) as the origin of meaning and then employs the meta-theory of, for example, sequent calculus as the formal setting for organizing that meaning. Girard's slogan for motivating Ludics, "From the rules of logic to the logic of rules" (18), is another illustration of placing inference rules at the center of system of meaning. The author has similarly described the value of replacing model

theory with proof theory as the vehicle for describing the meaning of logic-based programming languages (25).

Example: Alan Turing encoded computation using strings, machines, and computation traces. He used these to reason about the power of computing via standard mathematical techniques (inductive definitions, set-theoretical constructions, encodings as functions, etc). While this mathematical framework was 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. On the other hand, there is a great deal of structure and many formal results surrounding sequent calculus proofs that make it possible to reason richly about computation using abstractions, substitutions, and cut-elimination (26).

7. FOCUSED PROOF SYSTEMS

If we try to take the construction of proofs literally as a model for performing computation, one is immediately struck by the inappropriateness of sequents calculus for this task: there are just too many ways to build proofs and most of them differ in truly minor ways. While permuting the application of inference rules may yield proofs of different sequents, quite often such permutations yield different proofs of the same sequent. One would wish to have a much tighter correspondence between the application of an inference rule and sometime that might appear as an interesting "action" within a computation.

One of the first attempts to provide the sequent calculus with normal forms that could correspond to computation was the work on *uniform proofs* and *backchaining* (27) that was used to provide a proof theory foundation of logic programming. It was, however, Andreoli's *focused proof system* (1) for linear logic that really allowed one to more richly restriction and organize the sequent calculus. The earlier work on uniform proofs was rather limited, both in its "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. We provide here a high-level outline of the key ideas behind focused proof systems.

Focused proofs are generally divided into two, alternating phases. The first phase incorporates the inference rules that are invertible. This phase, sometimes also called the *negative* or *asynchronous* phase, ends (reading the proof from conclusion to premises) when all invertible inference rules have been applied. The second phase selects a formula on which to "focus": the inference rule that is applied to this formula is not necessarily invertible. Furthermore, if after the (reverse) application of that introduction rule, a subformula of that focused formula appears that also requires a non-invertible inference rule, then the phase continues with that subformula as the new focus. This second phase, also called the *positive* or *synchronous* phase, ends when either the proof ends with an instance of the initial rule or when only formulas with invertible inference rules are encountered. Certain "structural" rules are used to recognize the end of a phase or the switch from one phase to another.

7.1. A focused proof system for classical logic

This description of a focused proof system is only approximate. It is better to present a complete focused proof system to see a concrete example. Consider a presentation of first-order classical logic in which negations are applied only to atomic formulas and where the propositional connectives t, f, $\wedge$, and $\vee$ are replaced by two "polarized" versions: t^-, t^+, f^-, f^+, $\wedge^-$, $\wedge^+$, $\vee^-$, $\vee^+$. We shall also assume that to complete the notion of polarized formula, the atomic formulas are also polarized either positively or negatively. A formula is *negative* if it is a negative atom, the negation of a positive atom, or if its top-level connective is one of t^-, f^-, $\wedge^-$, $\vee^-$. A formula is *positive* if it is a positive atom, the negation of a negative atom, or if its top-level connective is one of t^+, f^+, $\wedge^+$, $\vee^+$. Notice that taking the De Morgan dual of a formula causes its polarity to flip.

The inference rules for the LKF focused proof system (23) for classical logic is given in Figure 1. Sequents are divided into *negative sequents* $\vdash \Theta \Uparrow \Gamma$ and *positive sequents* $\vdash \Theta \Downarrow B$, where Θ and Γ are multisets of formulas and B is a formula. (These sequents are formally one-sided sequents: formulas on the left of $\Uparrow$ and $\Downarrow$ are *not* negated as they are in two-sided sequents.) Notice that in this focused proof system, we have reused the term "structural rule" for a different set of rules which formally contains instances of weakening (Id) and contraction ($Focus$). Notice also that in any proof that has a conclusion of the form $\vdash \cdot \Uparrow B$, the only formulas that are to the left of an $\Uparrow$ or $\Downarrow$ occurring in that proof are either positive formulas or negative literals: it is only these formulas that are weakened (in the Id rule). The only formulas contracted (in the $Focus$ rule) are positive formulas. Thus, although linear logic is not used here directly, non-atomic negative formulas are treated linearly in the sense that they are never duplicated nor weakened in an LKF proof.

Let B be a formula of first-order logic. By a *polarization* of B we mean a formula, say B', where all the propositional connectives are replaced by *polarized versions* of the same connective and where all atomic formulas are assigned either a positive or negative polarity. Thus, an occurrence of the disjunction $\vee$ is replaced by

Structural Rules

$$\frac{\vdash \Theta, C \Uparrow \Gamma}{\vdash \Theta \Uparrow \Gamma, C}\ Store \qquad \frac{\vdash \Theta \Uparrow N}{\vdash \Theta \Downarrow N}\ Release$$

$$\frac{\vdash P, \Theta \Downarrow P}{\vdash P, \Theta \Uparrow \cdot}\ Focus \qquad \frac{}{\vdash \neg P, \Theta \Downarrow P}\ Id \text{ (literal } P\text{)}$$

Introduction of negative connectives

$$\frac{}{\vdash \Theta \Uparrow \Gamma, t^-} \qquad \frac{\vdash \Theta \Uparrow \Gamma, A \quad \vdash \Theta \Uparrow \Gamma, B}{\vdash \Theta \Uparrow \Gamma, A \wedge^- B}$$

$$\frac{\vdash \Theta \Uparrow \Gamma}{\vdash \Theta \Uparrow \Gamma, f^-} \qquad \frac{\vdash \Theta \Uparrow \Gamma, A, B}{\vdash \Theta \Uparrow \Gamma, A \vee^- B} \qquad \frac{\vdash \Theta \Uparrow \Gamma, A}{\vdash \Theta \Uparrow \Gamma, \forall x A}$$

Introduction of positive connectives

$$\frac{}{\vdash \Theta \Downarrow t^+} \qquad \frac{\vdash \Theta \Downarrow A \quad \vdash \Theta \Downarrow B}{\vdash \Theta \Downarrow A \wedge^+ B}$$

$$\frac{\vdash \Theta \Downarrow A_i}{\vdash \Theta \Downarrow A_1 \vee^+ A_2} \qquad \frac{\vdash \Theta \Downarrow A[t/x]}{\vdash \Theta \Downarrow \exists x A}$$

Figure 1: *The focused proof system LKF for classical logic. Here, P is positive, N is negative, C is a positive formula or a negative literal, Θ consists of positive formulas and negative literals, and x is not free in Θ, Γ. Endsequents have the form $\vdash \cdot \Uparrow \Gamma$.*

an occurrence of either $\vee^+$ or $\vee^-$; similarly with $\wedge$ and with the logical constants for true t and false f. For simplicity, we shall assume that polarization for atomic formulas is a global assignment to all atomic formulas. Properly speaking, focused proof systems contain *polarized* formulas and not simply formulas.

Theorem LKF is sound and complete for classical logic. More precisely, let B be a first order formula and let B' be a polarization of B. Then B is provable in classical logic if and only if there is an LKF proof of $\vdash \cdot \Uparrow B'$ (23).

Notice that polarization does not affect provability but it does affect the shape of possible LKF proofs. To illustrate an application of the correctness of LKF, we show how it provides a direct proof the following theorem.

Herbrand's Theorem Let B is quantifier-free formula and let $\bar{x}$ be a (non-empty) list of variables containing the free variables of B. The formula $\exists \bar{x} B$ is classically provable if and only if there is a list of substitutions $\theta_1, \ldots, \theta_m$ ($m \geq 1$), all with domain $\bar{x}$, such that the (quantifier-free) disjunction $B\theta_1 \vee \cdots \vee B\theta_m$ is provable (*i.e.*, tautologous).

Proof. The converse direction is straightforward. Thus, assume that $\exists \bar{x} B$ is provable. Let B' be the result of polarizing all occurrences of propositional connectives negatively. By the completeness of LKF, there is an LKF proof Ξ of $\vdash \exists \bar{x} B \Uparrow \cdot$. The only sequents of the form $\vdash \Theta \Uparrow \cdot$ in Ξ are such that Θ is equal to $\{\exists \bar{x} B'\} \cup \mathcal{L}$ for $\mathcal{L}$ a multiset of literals. Such a sequent can only be proved by a $Decide$ rule by focusing on either a positive literal in $\mathcal{L}$ or the original formula $\exists \bar{x} B'$: in the latter case, the synchronous phase above it provides a substitution for all the variables in $\bar{x}$. One only needs to collect all of these substitutions into a list $\theta_1, \ldots, \theta_m$ and then show that the proof Ξ is essentially also a proof of $\vdash B'\theta_1 \vee^+ \cdots \vee^+ B'\theta_m \Uparrow \cdot$. QED.

7.2. Positive and negative macro inference rules

Focused proof systems such as LKF allow us to change the size of inference rules with which we work. Let us call individual introduction rules (such as displayed in Section 6.1 and in Figure 1) "micro-rules". An entire phase within a focused proof can be seen as a "macro-rule". In particular, consider the following derivation.

$$\frac{\dfrac{\vdash \Theta, D \Uparrow N_1 \quad \cdots \quad \vdash \Theta, D \Uparrow N_n}{\vdash \Theta, D \Downarrow D}}{\vdash \Theta, D \Uparrow \cdot}$$

Here, the selection of the formula D for the focus can be taken as selecting among several macro-rules: this derivation illustrates one such macro-rule: the inference rule with conclusion $\vdash \Theta, D \Uparrow \cdot$ and with $n \geq 0$ premises $\vdash \Theta, D \Uparrow N_1, \ldots, \vdash \Theta, D \Uparrow N_n$ (where $N_1, \ldots, N_n$ are negative formulas). We shall say that this macro-rule is positive. Similarly, there is a corresponding negative macro-rule with conclusion, say, $\vdash \Theta, D \Uparrow N_i$, and with $m \geq 0$ premises of the form $\vdash \Theta, D, \mathcal{C} \Uparrow \cdot$, where $\mathcal{C}$ is a multiset of positive formulas or negative literals.

In this way, focused proofs allow us to view the construction of proofs from conclusions of the form $\vdash \Theta \Uparrow \cdot$ as first attaching a positive macro rule (by focusing on some formula in Θ) and then attaching negative inference rules to the resulting premises until one is again to sequents of the form $\vdash \Theta' \Uparrow \cdot$. Such a combination of a positive macro rule below negative macro rules is often called a *bipole* (2).

7.3. Fixed points and equality

In order for capture some interesting computational problems, the logic of propositional connectives and first-order quantifiers can be augmented with equality and fixed point operators. Consider the left and right introduction rules for $=$ and μ given in Figure 2. Notice that since the left and right introduction rules for μ are the same, μ is *self-dual*: that is, the De Morgan dual of μ is μ. It is possible to have a more expressive proof theory for fixed points that provides also for least and greatest fixed points (see, for example, (6; 4)): in that case, the De Morgan dual of the least fixed point is the greatest fixed point.

Example Identify the natural numbers as terms involving 0 for zero and s for successor. The following

$$\frac{\Gamma, B(\mu B)\bar{t} \vdash \Delta}{\Gamma, \mu B\bar{t} \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, B(\mu B)\bar{t}}{\Gamma \vdash \Delta, \mu B\bar{t}}$$

$$\frac{\Gamma\sigma \vdash \Delta\sigma}{\Gamma, s = t \vdash \Delta}\ \dagger \qquad \frac{}{\Gamma, s = t \vdash \Delta}\ \ddagger \qquad \frac{}{\Gamma \vdash \Delta, t = t}$$

Figure 2: *Introduction rules for $=$ and μ. B is a formula with $n \geq 0$ variables abstracted and $\bar{t}$ is a list of n terms. The $\dagger$ proviso requires the terms s and t to be unifiable and σ to be their most general unifier. The $\ddagger$ proviso requires that the terms s and t are not unifiable.*

$$\frac{\vdash \Theta\sigma \Uparrow \Gamma\sigma}{\vdash \Theta \Uparrow \Gamma, s \neq t}\ \dagger \qquad \frac{}{\vdash \Theta \Uparrow \Gamma, s \neq t}\ \ddagger \qquad \frac{}{\vdash \Theta \Downarrow t = t}$$

$$\frac{\vdash \Theta \Uparrow \Gamma, B(\mu B)\bar{t}}{\vdash \Theta \Uparrow \Gamma, \mu B\bar{t}} \qquad \frac{\vdash \Theta \Downarrow B(\mu B)\bar{t}}{\vdash \Theta \Downarrow \mu B\bar{t}}$$

Figure 3: *Focused inference rules for $=$ and μ. The proviso $\dagger$ and $\ddagger$ and the definition of σ are the same as above.*

simple logic program defines two predicates on natural numbers.

$$\begin{array}{rcl} nat\ 0 & \subset & true. \\ nat\ (s\ X) & \subset & nat\ X. \\ leq\ 0\ Y & \subset & true. \\ leq\ (s\ X)\ (s\ Y) & \subset & leq\ X\ Y. \end{array}$$

The predicate nat can be written as the fixed point

$$\mu(\lambda p \lambda x.(x = 0) \vee \exists y.(s\ y) = x \wedge p\ y)$$

and binary predicate leq (less-than-or-equal) can be written as the fixed point

$$\mu(\lambda q \lambda x \lambda y.(x = 0) \vee \exists u \exists v.(s\ u) = x \wedge (s\ v) = y \wedge q\ u\ v).$$

In a similar fashion, any Horn clause specification can be made into fixed point specifications (mutual recursions requires standard encoding techniques).

These two logical connectives can be added to LKF as follows. First, we classify both $=$ and μ as positive connectives (this choice is forced for equality while μ can be polarized either way). The (one-sided) focused versions of the introduction rules above are given in Figure 3.

Example Consider proving the positive focused sequent

$$\vdash \Theta \Downarrow (leq\ m\ n \wedge^+ N_1) \vee^+ (leq\ n\ m \wedge^+ N_2),$$

where m and n are natural numbers and leq is the fixed point expression displayed above but this time with all occurrences of $\wedge$ and $\vee$ polarized with their positive variants. If both N_1 and N_2 are negative formulas, then there are exactly two possible macro rules: one with premise $\vdash \Theta \Uparrow N_1$ when $m \leq n$ and one with premise $\vdash \Theta \Uparrow N_2$ when $n \leq m$ (thus, if $m = n$, either premise is possible). In this sense, a macro inference rule can contain an entire Prolog-style computation.

Example Macro rules can be built to match many computational situations. Consider, for example, defining simulation as the (greatest) fixed point of the equivalence

$$sim\ P\ Q \equiv \forall P' \forall A[P \xrightarrow{A} P' \supset \exists Q'[Q \xrightarrow{A} Q' \wedge sim\ P'\ Q']].$$

Although the right-hand-side of this definition looks complex, we show how it is possible to see proof search with this formula as being *exactly two* macro inference rules. First, the expression $P \xrightarrow{A} P'$ is, presumably, given via some SOS (structured operational semantic) specifications. Such specifications are simple, syntax-directed inference rules that can be captured as a least fixed point expression. As above, we will view such fixed point expressions as purely positive formulas. Thus, the expression $\forall P' \forall A[P \xrightarrow{A} P' \supset \cdot]$ is a negative macro rule: since all possible actions A and continuations P' must be computed, there are no choices to be made in building a proof for this expression. (Here, we are assuming that the implication $B \supset C$ is rendered as $\neg B \vee^- C$ in the polarized setting.) On the other hand, focusing on the expression $\exists Q'[Q \xrightarrow{A} Q' \wedge^+ \cdot]$ yields a non-invertible, positive macro rule. In this way, the focused proof system is aligned directly with the structure of the actual (model-checking) problem. Notice that if one wishes to communicate a proof of a simulation to a proof checker, no information regarding the use of the negative macro rule needs to be communicated since the proof checker can also perform the computation behind that inference rule (*i.e.*, enumerating all possible transitions of a given process P).

7.4. The engineering of proof systems

The fact that an entire computation can fit within a macro rule (using purely positive fixed point expressions) provides a great deal of flexibility in designing inference rules. Such flexibility allows inference rules to be designed so that they correspond to an "action" within a given computational system. One should note that placing arbitrary computation within an inference rule is probably too much: we usually use the term "inference rule" for some step of a proof for which it is decidable to check validity. Thus, some care should be exercised in balancing the complexity of a macro rule with the needs of proof systems to have their correctness be decidable.

Another technique in applying focusing proofs is the use of *delays*. Within LKF, we can define the delaying operators

$$\partial^+(B) = B \wedge^+ t^+ \quad \text{and} \quad \partial^-(B) = B \wedge^- t^-.$$

Clearly, B, $\partial^-(B)$, and $\partial^+(B)$ are all logically equivalent but $\partial^-(B)$ is always negative and $\partial^+(B)$ is always positive. If one wishes to break a positive macro rule resulting from focusing on a given positive formula into smaller pieces, then one can insert $\partial^-(\cdot)$ into that formula. Similarly, inserting $\partial^+(\cdot)$ can limit the size of a negative macro rule. By inserting many delay operators, a focused proof can be made to emulate an unfocused proof.

In general, proofs can be very large objects. If they are to be checked and communicated, then there must be some mechanisms that allow for exploring trade-offs between their size and their checking time. Focused proof systems here can help. First, although not illustrated here, focused proof systems can certainly contain versions of the cut-rule so lemmas can be incorporated into focused proof objects. Second, since macro inference rules correspond to computational steps, the designer of a proof system should often have a good sense of whether or not the search for short proofs could be transferred to the proof checker. For example, in the case of the judgment $P \xrightarrow{A} P'$, one might expect that the proof checker could search for proofs of such judgments given that this judgment is defined over (finite) syntactic expressions. Thus, by asking the proof checker to do some proof search, the size of proof certificates could be variably reduced.

8. THE UNITY OF PROOF SYSTEMS

Recent work in applying proof theory to the foundations of computational logic systems reveals that there are significant chances that a range of proof systems for, at least, classical, intuitionistic, and linear logics can be given a common foundation. We describe some of this recent work below.

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 proposals that try to provide alternatives to the exponentials. For example, Baelde and Miller (6; 4) 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 (5) 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 (17), 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, which involves unfolding fixed points and non-deterministically picking 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 (or have specialized methods for handling loops) while (inductive) theorem provers use invariants to characterize possible 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 reduce search space. Similarly, tabling within model checkers can be seen as lemma generation in theorem provers (28).

Many proof systems just differ in polarization Nigam and Miller (30) have recently shown that a range of commonly used proof systems (sequent calculus, natural deduction, tableaux, Hilbert-style, etc) are, in fact, just different polarizations 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 Algebraic-style rewriting is an important proof technique for reasoning about functional expressions. To what extent can rewriting with functions be captured within a relational setting? Functions can, of course, encode relations using set-valued functions and different 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 connection 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 and deduction-as-model-checking As we have mentioned, many (high-level aspects of) model checking can be seen as focused deduction with fixed points (4; 5; 6). Other

recent work (12) has shown that (in certain weak logics), deduction can be achieved by looking for winning strategies in suitable games. 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. BENEFITS OF A UNIFYING FRAMEWORK

The fractured nature of logical systems can be addressed by *ad hoc* solutions: standardized challenge problems, standardized frameworks, XML formats for formulas and proofs, protocols for plugging one tool into another tool, etc. While well engineered and inter-operating systems can have important practical consequences, one should insist that broader and more expressive foundations for computational logic systems be developed.

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

Flexible proof certificates An emphasis on unity (along with the technical results required to make it real) can be a *game changer*. Consider, for example, what can happen if a *highly flexible form of proof certificate* (one of the promises of focused proof system) were available. No longer are formal systems isolated. If proof certificates can be transmitted and easily checked, then one formal system can accept proofs from other systems no matter how well those formal systems trust each other. The existence of such proof certificates open an entirely new world of possibilities for attacking the problems of proving systems formally correct. For example, it should be possible to have libraries of proofs that take contributions from a wide range of deductive engines and not be limited to submissions from just one deduction engine. Also, since the proof certificates here are based on proof theory principles, rich search conditions against the library should be possible. Also possible should be a marketplace of proofs: the most successful practitioners in formal methods will be those who can choose the right combination of tools for getting formal results. Such a marketplace could lead to many new tools being developed for specialize but important domains (*e.g.*, avionics).

Transfer of implementation techniques A formal and rich foundation for a wide variety of computational logic systems will allow researchers and developers to transfer solutions they have developed in one area—*e.g.*, data structures, algorithms, or search heuristics—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 possible. For example, model checking and inductive/coinductive theorem proving can be seen as building sequent calculus proofs in a logic with fixed points (6; 4). Such a common deductive setting can be used to more tightly integrate these two rather different styles of deduction.

New breeds 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 (16) 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.

10. CONCLUSION AND SOME 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. If computer logic systems build proofs objects (explicitly or implicitly), those proofs can come from a wide range of different proof systems: 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; 30; 32). 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. Can we develop a principled approach to their 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, Gopalan Nadathur, Vivek Nigam, Alexis Saurin, Lutz Straßburger, Alexandre Viel, as well as my colleagues on the REDO project, namely, Alessio Guglielmi, François Lamarche, and Michel Parigot.

11. REFERENCES

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

[2] J.-M. Andreoli. Focussing and proof construction. *Annals of Pure and Applied Logic*, 107(1):131–163, 2001.

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

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

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] O. Delande, D. Miller, and A. Saurin. Proof and refutation in MALL as a game. *Annals of Pure and Applied Logic*, 161(5):654–672, Feb. 2010.

[13] 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.

[14] R. W. Emerson. *The Conduct of Life*. Smith, Elder, 2nd edition, 1860.

[15] G. Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969.

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

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

[18] 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.

[19] M. Gordon. From LCF to HOL: a short history. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 169–186. The MIT Press, 2000.

[20] 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.

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

[22] M. Kulstad and L. Carlin. Leibnizś philosophy of mind. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2008 edition, 2008.

[23] C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] V. Nigam and D. Miller. A framework for proof systems. Accepted by the J. of Automated Reasoning, Mar. 2009.

[33] 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.

[34] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

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

[36] 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.

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

[38] J. E. Santo. Revisiting the correspondence between cut elimination and normalisation. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, volume 1853 of *LNCS*, pages 600 – 611, 2000.

[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.