References

[abianco97] Davide Ancona, Francesco Dagnino, Jurriaan Rot, and Elena Zucca. A big step from finite to infinite computations. Technical report.

Abstract: We provide a construction that, given a big-step semantics describing finite computations and their observations, extends it to include infinite computations as well. The basic idea is that the finite behavior uniquely determines the infinite behavior once observations and their composition operators are fixed. Technically, the construction relies on the framework of inference systems with corules. The effectiveness and scope of the approach are illustrated by several examples. The correctness is formally justified by proving that, starting from a big-step semantics equivalent to a reference small-step semantics, this equivalence is preserved by the construction. (C) 2020 Elsevier B.V. All rights reserved.

[aceherm121] Paolo Herms, Claude Marche, and Benjamin Monate. A certified multi-prover verification condition generator. Technical report.

Abstract: Deduction-based software verification tools have reached a maturity allowing them to be used in industrial context where a very high level of assurance is required. This raises the question of the level of confidence we can grant to the tools themselves. We present a certified implementation of a verification condition generator. An originality is its genericity with respect to the logical context, which allows us to produce proof obligations for a large class of theorem provers.

[acerodr36] Pagano M. Rodríguez L., Fridlender D. A certified extension of the krivine machine for a call-by-name higher-order imperative language. Technical report.

Abstract: In this paper we present a compiler that translates programs from an imperative higher-order language into a sequence of instructions for an abstract machine. We consider an extension of the Krivine machine for the call-by-name lambda calculus, which includes strict operators and imperative features. We show that the compiler is correct with respect to the big-step semantics of our language, both for convergent and divergent programs. Leonardo Rodr guez, Daniel Fridlender, and Miguel Pagano.

[acoalbe57] Alberto Ciaffaglione. A coinductive semantics of the unlimited register machine. Technical report.

Abstract: We exploit (co)inductive specifications and proofs to approach the evaluation of low-level programs for the Unlimited Register Machine (URM) within the Coq system, a proof assistant based on the Calculus of (Co)Inductive Constructions type theory. Our formalization allows us to certify the implementation of partial functions, thus it can be regarded as a first step towards the development of a workbench for the formal analysis and verification of both converging and diverging computations.

[acoazú10] A Zúniga. A correct compiler from mini-ml to a big-step machine verified using natural semantics in coq. Technical report.

Abstract: This work has the objective to present a simple, clear and intuitive framework for compilers verification of functional languages in the proof assistant Coq, that, as a final product, can obtain a standalone verified compiler capable of being used in real life. With this in mind, we propose to use natural semantics as unifying framework, that is to say, to use this formalism to define each of the compiler's components in order to perform this task. To show this method, we present a correct compiler of the small functional language with call by value Mini-ML, formalized in Coq. As a result of following this approach, we introduce a new big-step machine inspired by Landin's SECD and Leroy's Modern SECD as target machine. To the best of the authors' knowledge, this is the first correct compiler that is verified by using natural semantics as unifying framework in Coq from which we can obtain a verified compiler capable of being used in real life.

[acopéte11] Péter Bereczky. A comparison of big-step semantics definition styles. Technical report.

Abstract: Formal semantics provides rigorous, mathematically precise definitions of programming languages, with which we can argue about program behaviour and program equivalence by formal means; in particular, we can describe and verify our arguments with a proof assistant. There are various approaches to giving formal semantics to programming languages, at different abstraction levels and applying different mathematical machinery: the reason for using the semantics determines which approach to choose. In this paper we investigate some of the approaches that share their roots with traditional relational big-step semantics, such as (a) functional big-step semantics (or, equivalently, a definitional interpreter), (b) pretty-big-step semantics and (c) traditional natural semantics. We compare these approaches with respect to the following criteria: executability of the semantics definition, proof complexity for typical properties (e.g. determinism) and the conciseness of expression equivalence proofs in that approach. We also briefly discuss the complexity of these definitions and the coinductive big-step semantics, which enables reasoning about divergence. To enable the comparison in practice, we present an example language for comparing the semantics: a sequential subset of Core Erlang, a functional programming language, which is used in the intermediate steps of the Erlang/OTP compiler. We have already defined a relational big-step semantics for this language that includes treatment of exceptions and side effects. The aim of this current work is to compare our big-step definition for this language with a variety of other equivalent semantics in different styles from the point of view of testing and verifying code refactorings.

[acorama104] Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Jeremie Koenig, and Yuchen Fu. A compositional semantics for verified separate compilation and linking. Technical report.

Abstract: Recent ground-breaking efforts such as CompCert have made a convincing case that mechanized verification of the compiler correctness for realistic C programs is both viable and practical. Unfortunately, existing verified compilers can only handle whole programs-this severely limits their applicability and prevents the linking of verified C programs with verified external libraries. In this paper, we present a novel compositional semantics for reasoning about open modules and for supporting verified separate compilation and linking. More specifically, we replace external function calls with explicit events in the behavioral semantics. We then develop a verified linking operator that makes lazy substitutions on (potentially reacting) behaviors by replacing each external function call event with a behavior simulating the requested function. Finally, we show how our new semantics can be applied to build a refinement infrastructure that supports both vertical composition and horizontal composition.

[acołuka10] Łukasz Czajka. A coinductive confluence proof for infinitary lambda-calculus. Technical report.

Abstract: We give a coinductive proof of confluence, up to equivalence of root-active subterms, of infinitary lambda-calculus. We also show confluence of B hm reduction (with respect to root-active terms) in infinitary lambda-calculus. In contrast to previous proofs, our proof makes heavy use of coinduction and does not employ the notion of descendants.

[ademanu65] Manuel Eberl. A decision procedure for univariate real polynomials in isabelle/hol. Technical report.

Abstract: Sturm sequences are a method for computing the number of real roots of a univariate real polynomial inside a given interval efficiently. In this paper, this fact and a number of methods to construct Sturm sequences efficiently have been formalised with the interactive theorem prover Isabelle/HOL. Building upon this, an Isabelle/HOL proof method was then implemented to prove interesting statements about the number of real roots of a univariate real polynomial and related properties such as non-negativity and monotonicity.

[advachen33] James Cheney, Alberto Momigliano, and Matteo Pessina. Advances in property-based testing for alpha prolog. Technical report.

> **Abstract:** alpha Check is a light-weight property-based testing tool built on top of alpha Prolog, a logic programming language based on nominal logic. alpha Prolog is particularly suited to the validation of the meta-theory of formal systems, for example correctness of compiler translations involving name-binding, alpha-equivalence and capture-avoiding substitution. In this paper we describe an alternative to the negation elimination algorithm underlying alpha Check that substantially improves its effectiveness. To substantiate this claim we compare the checker performances w.r.t. two of its main competitors in the logical framework niche, namely the QuickCheck/Nitpick combination offered by Isabelle/HOL and the random testing facility in PLT-Redex.

[advachen60] Pessina M. Cheney J., Momigliano A. Advances in property-based testing for prolog. Technical report.

> **Abstract:** Check is a light-weight property-based testing tool built on top of Prolog, a logic programming language based on nominal logic. Prolog is particularly suited to the validation of the meta-theory of formal systems, for example correctness of compiler translations involving name-binding, alpha-equivalence and capture-avoiding substitution. In this paper we describe an alternative to the negation elimination algorithm underlying Check that substantially improves its effectiveness. To substantiate this claim we compare the checker performances w.r.t. two of its main competitors in the logical framework niche, namely the QuickCheck/Nitpick combination offered by Isabelle/HOL and the random testing facility in PLT-Redex. Springer International Publishing Switzerland 2016.

[afoallo77] Victor Allombert, Frederic Gava, and Julien Tesson. A formal semantics of the multi-ml language. Technical report.

> **Abstract:** In the context of high performance computing, it is important to avoid indeterminism and dead-locks. MULTI-ML is a functional parallel programming language a la ML, designed to program hierarchical architectures in a structured way. It is based of the MULTI-BSP bridging model. To ensure that a program cannot go wrong, we first need to define how a program goes. To do so, we propose a formal operational semantics of the MULTI-ML language to ensure the properties of the MULTI-BSP model. We first describe a core-language and then introduce the big step's semantics evaluation rules. Then, we propose a set of evaluation rules that describe the behaviour of the MULTI-ML language. The memory model is also precisely defined, as the MULTI-BSP model deals with multiple level of nested memories.

[afolero58] Leroy X. A formally verified compiler back-end. Technical report.

> **Abstract:** This article describes the development and formal verification (proof of semantic preservation) of a compiler back-end from Cminor (a simple imperative intermediate language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its soundness. Such a verified compiler is useful in the context of formal methods applied to the certification of critical software: the verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well. 2009 Springer Science+Business Media B.V.

[afrdagn64] Dagnino F. A framework for big-step semantics. Technical report.

> **Abstract:** [No abstract available]

[ahonaka71] Keiko Nakata and Tarmo Uustalu. A hoare logic for the coinductive trace-based big-step semantics of while. Technical report.

Abstract: In search for a foundational framework for reasoning about observable behavior of programs that may not terminate, we have previously devised a trace-based big-step semantics for While. In this semantics, both traces and evaluation (relating initial states of program runs to traces they produce) are defined coinductively. On terminating runs, this semantics agrees with the standard inductive state-based semantics. Here we present a Hoare logic counterpart of our coinductive trace-based semantics and prove it sound and complete. Our logic subsumes the standard partial-correctness state-based Hoare logic as well as the total-correctness variation: they are embeddable. In the converse direction, projections can be constructed: a derivation of a Hoare triple in our trace-based logic can be translated into a derivation iii the state-based logic of a translated, weaker Hoare triple. Since we work with a constructive underlying logic, the range of program properties we can reason about has a fine structure; in particular, we can distinguish between termination and nondivergence, e.g., unbounded classically total search fails to be terminating, but is nonetheless nondivergent. Our metatheorv is entirely constructive as well, and we have formalized it in Coq.

[aliappe11] Andrew W. Appel, Robert Dockins, and Xavier Leroy. A list-machine benchmark for mechanized metatheory. Technical report.

Abstract: We propose a benchmark to compare theorem-proving systems on their ability to express proofs of compiler correctness. In contrast to the first POPLmark, we emphasize the connection of proofs to compiler implementations, and we point out that much can be done without binders or alpha-conversion. We propose specific criteria for evaluating the utility of mechanized metatheory systems; we have constructed solutions in both Coq and Twelf metatheory, and we draw conclusions about those two systems in particular.

[alosimm2] Zerny I. Simmons R.J. A logical correspondence between natural semantics and abstract machines. Technical report.

Abstract: We present a logical correspondence between natural semantics and abstract machines. This correspondence enables the mechanical and fully-correct construction of an abstract machine from a natural semantics. Our logical correspondence mirrors the Reynolds functional correspondence, but we manipulate semantic specifications encoded in a logical framework instead of manipulating functional programs. Natural semantics and abstract machines are instances of substructural operational semantics. As a byproduct, using a substructural logical framework, we bring concurrent and stateful models into the domain of the logical correspondence. 2013 ACM.

[amedagn24] Francesco Dagnino. A meta-theory for big-step semantics. Technical report.

Abstract: It is well known that big-step semantics is not able to distinguish stuck and non-terminating computations. This is a strong limitation as it makes it very difficult to reason about properties involving infinite computations, such as type soundness, which cannot even be expressed.We show that this issue is only apparent: the distinction between stuck and diverging computations is implicit in any big-step semantics and it just needs to be uncovered. To achieve this goal, we develop a systematic study of big-step semantics: we introduce an abstract definition of what a big-step semantics is, we define a notion of computation by formalizing the evaluation algorithm implicitly associated with any big-step semantics, and we show how to canonically extend a big-step semantics to characterize stuck and diverging computations.Building on these notions, we describe a general proof technique to show that a predicate is sound, that is, it prevents stuck computation, with respect to a big-step semantics. One needs to check three properties relating the predicate and the semantics, and if they hold, the predicate is sound. The extended semantics is essential to establish this meta-logical result but is of no concerns to the user, who only needs to prove the three properties of the initial big-step semantics. Finally, we illustrate the technique by several examples, showing that it is applicable also in cases where subject reduction does not hold, and hence the standard technique for small-step semantics cannot be used.

[amugn14] G. Nerjes. A multimedia information server with mixed workload scheduling. Technical report.

Abstract: In contrast to specialized video servers, advanced multimedia applications for tele-shopping, tele-teaching and news-on-demand exhibit a mixed workload with massive access to conventional, discrete data such as text document, images and indexes as well as requests for continuous data such as video. The paper briefly describes the prototype of a multimedia information server that stores discrete and continuous data on a shared disk pool and is able to handle a mixed workload in a very efficient way

[aneczaj44] Lukasz Czajka. A new coinductive confluence proof for infinitary lambda calculus. Technical report.

Abstract: We present a new and formal coinductive proof of confluence and normalisation of Bohm reduction in infinitary lambda calculus. The proof is simpler than previous proofs of this result. The technique of the proof is new, i.e., it is not merely a coinductive reformulation of any earlier proofs. We formalised the proof in the Coq proof assistant.

[aprgian36] Paola Giannini and Albert Shaqiri. A provably correct compilation of functional languages into scripting languags. Technical report.

Abstract: In this paper we consider the problem of translating core F#, a typed functional language including mutable variables and exception handling, into scripting languages such as JavaScript or Python. In previous work, we abstracted the most significant characteristics of scripting languages in an intermediate language (IL for short). IL is a block-structured imperative language in which a definition of a name does not have to statically precede its use. We define a big-step operational semantics for core F# and for IL and formalise the translation of F# expressions into IL. The main contribution of the paper is the proof of correctness of the given translation, which is done by showing that the evaluation of a well-typed F# program converges to a primitive value if and only if the evaluation of its translation into IL converges to the same value.

[aprgian60] Shaqiri A. Giannini P. A provably correct compilation of functional languages into scripting languages. Technical report.

Abstract: In this paper we consider the problem of translating core F#, a typed functional language including mutable variables and exception handling, into scripting languages such as JavaScript or Python. In previous work, we abstracted the most significant characteristics of scripting languages in an intermediate language (IL for short). IL is a block-structured imperative language in which a definition of a name does not have to statically precede its use. We define a big-step operational semantics for core F# and for IL and formalise the translation of F# expressions into IL. The main contribution of the paper is the proof of correctness of the given translation, which is done by showing that the evaluation of a well-typed F# program converges to a primitive value if and only if the evaluation of its translation into IL converges to the same value. 2017, Alexandru Ioan Cuza University of Iasi. All rights reserved.

[astdann51] Royer J.S. Danner N., Paykin J. A static cost analysis for a higher-order language. Technical report.

Abstract: We develop a static complexity analysis for a higher-order functional language with structural list recursion. The complexity of an expression is a pair consisting of a cost and a potential. The former is defined to be the size of the expression's evaluation derivation in a standard big-step operational semantics. The latter is a measure of the 'future' cost of using the value of that expression. A translation function maps target expressions to complexities. Our main result is the following Soundness Theorem: If t is a term in the target language, then the cost component of ktk is an upper bound on the cost of evaluating t. The proof of the Soundness Theorem is formalized in Coq, providing certified upper bounds on the cost of any expression in the target language. Copyright 2013 ACM.

[asumomi67] Momigliano A. A supposedly fun thing i may have to do again: A hoas encoding of howe's method. Technical report.

Abstract: We formally verify in Abella that similarity in the call-by-name lambda calculus is a pre-congruence, using Howe's method. This turns out to be a very challenging task for HOAS-based systems, as it entails a demanding combination of inductive and coinductive reasoning on open terms, for which no other existing HOAS-based system is equipped for. We also offer a proof using a version of Abella supplemented with predicate quantification; this results in a more structured presentation that is largely independent of the operational semantics as well of the chosen notion of (bi)similarity. While the end result is significantly more succinct and elegant than previous attempts, the exercise highlights some limitations of the two-level approach in general and of Abella in particular. 2012 ACM.

[athmass29] Massimo Bartoletti. A theory of agreements and protection. Technical report.

Abstract: In this thesis we propose a theory of contracts. Contracts are modelled as interacting processes with an explicit association of obligations and objectives. Obligations are specified using event structures. In this model we formalise two fundamental notions of contracts, namely agreement and protection. These notions arise naturally by interpreting contracts as multi-player concurrent games. A participant agrees on a contract if she has a strategy to reach her objectives (or to make another participant sanctionable for a violation), whatever the moves of her counterparts. A participant is protected by a contract when she has a strategy to defend herself in all possible contexts, even in those where she has not reached an agreement. When obligations are represented using classical event structures, we show that agreement and protection mutually exclude each other for a wide class of contracts. To reconcile agreement with protection we propose a novel formalism for modelling contractual obligations: event structures with circular causality. We study this model from a foundational perspective, and we relate it with classical event structures. Using this model, we show how to construct contracts which guarantee both agreement and protection. We relate our contract model with Propositional Contract Logic, by establishing a correspondence between provability in the logic and the notions of agreement and strategies. This is a first step towards reducing the gap between two main paradigms for modelling contracts, that is the one which interprets them as interactive systems, and the one based on logic.

[autosjo88] S Jost. Automated amortised analysis. Technical report.

Abstract: Steffen Jost researched a novel static program analysis that automatically infers formally guaranteed upper bounds on the use of compositional quantitative resources. The technique is based on the manual amortised complexity analysis. Inference is achieved through a type system annotated with linear constraints. Any solution to the collected constraints yields the coefficients of a formula, that expresses an upper bound on the resource consumption of a program through the sizes of its various inputs. The main result is the formal soundness proof of the proposed analysis for a functional language. The strictly evaluated language features higher-order types, full mutual recursion, nested data types, suspension of evaluation, and can deal with aliased data. The presentation focuses on heap space bounds. Extensions allowing the inference of bounds on stack space usage and worst-case execution time are demonstrated for several realistic program examples. These bounds were inferred by the created generic implementation of the technique. The implementation is highly efficient, and solves even large examples within seconds. Steffen Jost stellt eine neuartige statische Programmanalyse vor, welche vollautomatisch Schranken an den Verbrauch quantitativer Ressourcen berechnet. Die Grundidee basiert auf der Technik der Amortisierten Komplexit tsanalyse, deren nicht-triviale Automatisierung durch ein erweitertes Typsystem erreicht wird. Das Typsystem berechnet als Nebenprodukt ein lineares Gleichungssystem, dessen L sungen Koeffizienten f r lineare Formeln liefern. Diese Formeln stellen garantierte obere Schranken an den Speicher- oder Zeitverbrauch des analysierten Programms dar, in Abh ngigkeit von den verschiedenen Eingabegr en des Programms. Die Relevanz der einzelnen Eingabegr en auf den Ressourcenverbrauch wird so deutlich beziffert. Die formale Korrektheit der Analyse wird f r eine funktionale Programmiersprache bewiesen. Die strikte Sprache erlaubt: Typen h herer Ordnung, volle Rekursion, verschachtelte Datentypen, explizites Aufschieben der Auswertung und Aliasing. Die formale Beschreibung der Analyse befasst sich prim r mit dem Verbrauch von dynamischen Speicherplatz. F r eine Reihe von realistischen Programmbeispielen wird demonstriert, dass die angefertigte generische Implementation auch gute Schranken an den Verbrauch von Stapelspeicher und der maximalen Ausf hrungszeit ermitteln kann. Die Analyse ist sehr effizient implementierbar, und behandelt auch gr ere Beispielprogramme vollst ndig in wenigen Sekunden.

[avechli116] Chlipala A. A verified compiler for an impure functional language. Technical report.

Abstract: We present a verified compiler to an idealized assembly language from a small, untyped functional language with mutable references and exceptions. The compiler is programmed in the Coq proof assistant and has a proof of total correctness with respect to big-step operational semantics for the source and target languages. Compilation is staged and includes standard phases like translation to continuation-passing style and closure conversion, as well as a common subexpression elimination optimization. In this work, our focus has been on discovering and using techniques that make our proofs easy to engineer and maintain. While most programming language work with proof assistants uses very manual proof styles, all of our proofs are implemented as adaptive programs in Coq's tactic language, making it possible to reuse proofs unchanged as new language features are added. In this paper, we focus especially on phases of compilation that rearrange the structure of syntax with nested variable binders. That aspect has been a key challenge area in past compiler verification projects, with much more effort expended in the statement and proof of binder-related lemmas than is found in standard pencil-and-paper proofs. We show how to exploit the representation technique of parametric higher-order abstract syntax to avoid the need to prove any of the usual lemmas about binder manipulation, often leading to proofs that are actually shorter than their pencil-and-paper analogues. Our strategy is based on a new approach to encoding operational semantics which delegates all concerns about substitution to the meta language, without using features incompatible with general-purpose type theories like Coq's logic. Copyright 2010 ACM.

[beatnils78] Nils Anders Danielsson. Beating the productivity checker using embedded languages. Technical report.

[biipatr77] Patrick Cousot. Bi-inductive structural semantics. Technical report.

Abstract: We propose a simple order-theoretic generalization, possibly non-monotone, of set-theoretic inductive definitions. This generalization covers inductive, co-inductive and bi-inductive definitions and is preserved by abstraction. This allows structural operational semantics to describe simultaneously the finite terminating and infinite diverging behaviors of programs. This is illustrated on grammars and the structural bifinitary small big-step trace relational operational semantics of the call-by-value -calculus (for which co-induction is shown to be inadequate).

[biorbent85] Hur C.-K. Benton N. Biorthogonality, step-indexing and compiler correctness. Technical report.

Abstract: We define logical relations between the denotational semantics of a simply typed functional language with recursion and the operational behaviour of low-level programs in a variant SECD machine. The relations, which are defined using biorthogonality and stepindexing, capture what it means for a piece of low-level code to implement a mathematical, domain-theoretic function and are used to prove correctness of a simple compiler. The results have been formalized in the Coq proof assistant. Copyright 2009 ACM.

[breajlo111] J Lopez. Breaking boundaries between programming languages and databases. Technical report.

Abstract: Several classes of solutions allow programming languages to express queries: Specific APIs such as JDBC, Object-Relational Mappings (ORMs) such as Hibernate, and language-integrated query frameworks such as Microsoft's LINQ. However, most of these solutions do not allow for efficient cross-databases queries, and none allow the use of complex application logic from the programming language in queries. In this thesis, we create a language-integrated query framework called BOLDR that, in particular, allows the evaluation in databases of queries written in general-purpose programming languages that contain application logic, and that target different databases of possibly different data models. In this framework, application queries are translated to an intermediate representation, then rewritten in order to avoid query avalanches and make the most out of database optimizations, and finally sent for evaluation to the corresponding databases and the results are converted back to the application. Our experiments show that the techniques we implemented are applicable to real-world database applications, successfully handling a variety of language-integrated queries with good performances.

[cakekuma7] Norrish M. Owens S. Kumar R., Myreen M.O. Cakeml: A verified implementation of ml. Technical report.

Abstract: We have developed and mechanically verified an ML system called CakeML, which supports a substantial subset of Standard ML. CakeML is implemented as an interactive read-eval-print loop (REPL) in x86-64 machine code. Our correctness theorem ensures that this REPL implementation prints only those results permitted by the semantics of CakeML. Our verification effort touches on a breadth of topics including lexing, parsing, type checking, incremental and dynamic compilation, garbage collection, arbitrary-precision arithmetic, and compiler bootstrapping. Our contributions are twofold. The first is simply in building a system that is end-to-end verified, demonstrating that each piece of such a verification effort can in practice be composed with the others, and ensuring that none of the pieces rely on any over-simplifying assumptions. The second is developing novel approaches to some of the more challenging aspects of the verification. In particular, our formally verified compiler can bootstrap itself: we apply the verified compiler to itself to produce a verified machine-code implementation of the compiler. Additionally, our compiler proof handles diverging input programs with a lightweight approach based on logical timeout exceptions. The entire development was carried out in the HOL4 theorem prover. 2014 ACM.

[certbodi71] Martin Bodin, Thomas Jensen, and Alan Schmitt. Certified abstract interpretation with pretty-big-step semantics. Technical report.

Abstract: This paper describes an investigation into developing certified abstract interpreters from big-step semantics using the Coq proof assistant. We base our approach on Schmidt's abstract interpretation principles for natural semantics, and use a pretty-big-step (PBS) semantics, a semantic format proposed by Chargueraud. We propose a systematic representation of the PBS format and implement it in Coq. We then show how the semantic rules can be abstracted in a methodical fashion, independently of the chosen abstract domain, to produce a set of abstract inference rules that specify an abstract interpreter. We prove the correctness of the abstract interpreter in Coq once and for all, under the assumption that abstract operations faithfully respect the concrete ones. We finally show how to define correct-by-construction analyses: their correction amounts to proving they belong to the abstract semantics.

[certgill48] Gilles Barthe. Certificate translation for optimizing compilers. Technical report.

[certphe123] P Herms. Certification of a tool chain for deductive program verification. Technical report.

Abstract: Frama-C1 is an extensible platform dedicated to source-code analysis of C software. It is organised with a plug-in architecture based on a common kernel and the common formal annotation language ACSL (ANSI/ISO C Specication Language)[1], able to express a wide range of functional properties of the C code. The Jessie plug-in aims to formally prove such ACSL properties. Based on the Why platform designed for program verication[3], it transforms annotated C programs into equivalent programs in the Why intermediate language. The Why tool then processes a Hoare-style weakest-precondition (WP) calculus obtaining a set of verication conditions (VCs). The validation of these generated VCs deductively implies the correctness of the C program with respect to its annotations. The purpose of this work is to formalise and certify this chain in the Coq Proof Assistant. At the end, we want to obtain a formalisation of a suciently large subset of C with ACSL to interface with Compcert's C-light[2], a formalisation of the Why language and its WP calculus and a certied compilation from the former to the latter.

[certxsh100] X Shi. Certification of an instruction set simulator. Technical report.

Abstract: Approaches based on axiomatic semantics (typically, Hoare logic) are the mostpopular for proving the correctness of imperative programs. However, we prefered totry a less usual but more direct approach, based on operational semantics : this wasmade possible in theory since the development of an operational semantics for theC language formalized in Coq in the CompCert project, and allowed us to use thecomfortable logic of Coq, of much help for managing the complexity of the specification.Up to our knowledge, this is the first development of formal correctness proofs basedon operational semantics, at least at this scale.We provide a formalized representation of the ARM instruction set and addressingmodes in Coq, using an automatic code generator from the instruction pseudo-code inthe ARM reference manual. We also generate a Coq representation of a correspondingsimulator in C, called Simlight, using the abstract syntax defined in CompCert.From these two Coq representations, we can then state and prove the correctnessof Simlight, using the operational semantics of C provided by CompCert. Currently,proofs are available for at least one instruction in each category of the ARM instructionset.During this work, we improved the technology available in Coq for performinginversions, a kind of proof steps which heavily occurs in our setting.

[charpohj18] Myreen M.O. Pohjola J.Å., Rostedt H. Characteristic formulae for liveness properties of non-terminating cakeml programs. Technical report.

Abstract: There are useful programs that do not terminate, and yet standard Hoare logics are not able to prove liveness properties about non-terminating programs. This paper shows how a Hoare-like programming logic framework (characteristic formulae) can be extended to enable reasoning about the I/O behaviour of programs that do not terminate. The approach is inspired by transfinite induction rather than coinduction, and does not require non-terminating loops to be productive. This work has been developed in the HOL4 theorem prover and has been integrated into the ecosystem of proof tools surrounding the CakeML programming language. Johannes man Pohjola, Henrik Rostedt, and Magnus O. Myreen.

[circbart32] Massimo Bartoletti, Tiziana Cimoli, G. Michele Pinna, and Roberto Zunino. Circular causality in event structures. Technical report.

> **Abstract:** We propose a model of events with circular causality, in the form of a conservative extension of Winskel's event structures. We study the relations between this new kind of event structures and Propositional Contract Logic. Provable atoms in the logic correspond to reachable events in our event structures. Furthermore, we show a correspondence between the configurations of this new brand of event structures and the proofs in a fragment of Propositional Contract Logic.

[coaxdagn13] Francesco Dagnino. Coaxioms: Flexible coinductive definitions by inference systems. Technical report.

> **Abstract:** We introduce a generalized notion of inference system to support more flexible interpretations of recursive definitions. Besides axioms and inference rules with the usual meaning, we allow also coaxioms, which are, intuitively, axioms which can only be applied at infinite depth in a proof tree. Coaxioms allow us to interpret recursive definitions as fixed points which arc not necessarily the least, nor the greatest one, whose existence is guaranteed by a smooth extension of classical results. This notion nicely subsumes standard inference systems and their inductive and coinductive interpretation, thus allowing formal reasoning in cases where the inductive and coinductive interpretation do not provide the intended meaning, but are rather mixed together.

[coinanco121] Ancona D. Coinductive big-step operational semantics for type soundness of java-like languages. Technical report.

> **Abstract:** We define a coinductive semantics for a simple Java-like language by simply interpreting coinductively the rules of a standard big-step operational semantics. We prove that such a semantics is sound w.r.t. the usual small-step operational semantics, and then prove soundness of a conventional nominal type system w.r.t. the coinductive semantics. From these two results, soundness of the type system w.r.t. the small-step semantics can be easily deduced. This new proposed approach not only opens up new possibilities for proving type soundness, but also provides useful insights on the connection between coinductive big-step operational semantics and type systems. Copyright 2011 ACM.

[coinhyeo79] Hyeonseung Im. Coinductive subtyping for recursive and union types. Technical report.

> **Abstract:** Induction and coinduction are well-established proof principles, which are widely used in mathematics and computer science. In particular, induction is taught in most undergraduate programs and well understood in the field of computer science. In contrast, coinduction is not as widespread or well understood as induction. In this paper, we introduce coinduction by defining a subtype system for recursive and union types and proving the transitivity property of the system. This paper will help to promote familiarity with coinduction and provides a basis for a subtype system for recursive types with other advanced type constructors and connectives.

[coinuust78] Uustalu T. Coinductive big-step semantics for concurrency. Technical report.

> **Abstract:** In a paper presented at SOS 2010 [13], we developed a framework for big-step semantics for interactive input-output in combination with divergence, based on coinductive and mixed inductive-coinductive notions of resumptions, evaluation and termination-sensitive weak bisimilarity. In contrast to standard inductively defined big-step semantics, this framework handles divergence properly; in particular, runs that produce some observable effects and then diverge, are not 'lost'. Here we scale this approach for shared-variable concurrency on a simple example language. We develop the metatheory of our semantics in a constructive logic.

[coinzuni57] Angel Zuniga and Gemma Bel-Enguix. Coinductive natural semantics for compiler verification in coq. Technical report.

Abstract: (Coinductive) natural semantics is presented as a unifying framework for the verification of total correctness of compilers in Coq (with the feature that a verified compiler can be obtained). In this way, we have a simple, easy, and intuitive framework; to carry out the verification of a compiler, using a proof assistant in which both cases are considered: terminating and non-terminating computations (total correctness).

[combcorr21] Signoles J. Correnson L. Combining analyses for c program verification. Technical report.

Abstract: Static analyzers usually return partial results. They can assert that some properties are valid during all possible executions of a program, but generally leave some other properties to be verified by other means. In practice, it is common to combine results from several methods manually to achieve the full verification of a program. In this context, Frama-C is a platform for analyzing C source programs with multiple analyzers. Hence, one analyzer might conclude about properties assumed by another one, in the same environment. We present here the semantical foundations of validity of program properties in such a context. We propose a correct and complete algorithm for combining several partial results into a fully consolidated validity status for each program property. We illustrate how such a framework provides meaningful feedback on partial results. 2012 Springer-Verlag.

[comppara39] Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. Compositional optimizations for certicoq. Technical report.

Abstract: Compositional compiler verification is a difficult problem that focuses on separate compilation of program components with possibly different verified compilers. Logical relations are widely used in proving correctness of program transformations in higher-order languages; however, they do not scale to compositional verification of multi-pass compilers due to their lack of transitivity. The only known technique to apply to compositional verification of multi-pass compilers for higher-order languages is parametric inter-language simulations (PILS), which is however significantly more complicated than traditional proof techniques for compiler correctness. In this paper, we present a novel verification framework for lightweight compositional compiler correctness. We demonstrate that by imposing the additional restriction that program components are compiled by pipelines that go through the same sequence of intermediate representations, logical relation proofs can be transitively composed in order to derive an end-to-end compositional specification for multi-pass compiler pipelines. Unlike traditional logical-relation frameworks, our framework supports divergence preservationDeven when transformations reduce the number of program steps. We achieve this by parameterizing our logical relations with a pair of relational invariants.We apply this technique to verify a multi-pass, optimizing middle-end pipeline for CertiCoq, a compiler from Gallina (Coq's specification language) to C. The pipeline optimizes and closure-converts an untyped functional intermediate language (ANF or CPS) to a subset of that language without nested functions, which can be easily code-generated to low-level languages. Notably, our pipeline performs more complex closure-allocation optimizations than the state of the art in verified compilation. Using our novel verification framework, we prove an end-to-end theorem for our pipeline that covers both termination and divergence and applies to whole-program and separate compilation, even when different modules are compiled with different optimizations. Our results are mechanized in the Coq proof assistant.

[compwang34] Peng Wang, Santiago Cuellar, and Adam Chlipala. Compiler verification meets cross-language linking via data abstraction. Technical report.

Abstract: Many real programs are written in multiple different programming languages, and supporting this pattern creates challenges for formal compiler verification. We describe our Coq verification of a compiler for a high-level language, such that the compiler correctness theorem allows us to derive partial-correctness Hoare-logic theorems for programs built by linking the assembly code output by our compiler and assembly code produced by other means. Our compiler supports such tricky features as storable cross-language function pointers, without giving up the usual benefits of being able to verify different compiler phases (including, in our case, two classic optimizations) independently. The key technical innovation is a mixed operational and axiomatic semantics for the source language, with a built-in notion of abstract data types, such that compiled code interfaces with other languages only through axiomatically specified methods that mutate encapsulated private data, represented in whatever formats are most natural for those languages.

[confczaj63] Czajka U. Confluence of nearly orthogonal infinitary term rewriting systems. Technical report.

Abstract: terms, of nearly orthogonal infinitary term rewriting systems. Nearly orthogonal systems allow certain root overlaps, but no non-root overlaps. Using a slightly more complicated method we also show confluence modulo equivalence of hypercollapsing terms. The condition we impose on root overlaps is similar to the condition used by Toyama in the context of finitary rewriting. Lukasz Czajka.

[coqchrz102] Zakrzewski J. Chrzaszcz J., Schubert A. Coq support in haha. Technical report.

Abstract: HAHA is a tool that helps in teaching and learning Hoare logic. It is targeted at an introductory course on software verification. We present a set of new features of the HAHA verification environment that exploit Coq. These features are (1) generation of verification conditions in Coq so that they can be explored and proved interactively and (2) compilation of HAHA programs into CompCert certified compilation tool-chain. With the interactive Coq proving support we obtain an interesting functionality that makes it possible to carefully examine step-by-step verification conditions and systematically discover flaws in their formulation. As a result Coq back-end serves as a kind of specification debugger. Jacek Chrzaszcz, Aleksy Schubert, and Jakub Zakrzewski; licensed under Creative Commons License CC-BY 22nd International Conference on Types for Proofs and Programs (TYPES 2016).

[coredavi121] Davide Ancona. Corecursive featherweight java. Technical report.

Abstract: Despite cyclic data structures occur often in many application domains, object-oriented programming languages provide poor abstraction mechanisms for dealing with cyclic objects. Such a deficiency is reflected also in the research on theoretical foundation of object-oriented languages; for instance, Featherweigh Java (FJ), which is one of the most widespread object-oriented calculi, does not allow creation and manipulation of cyclic objects. We propose an extension to Featherweight Java, called COFJ, where it is possible to define cyclic objects, abstractly corresponding to regular terms, and where an abstraction mechanism, called regular corecursion, is provided for supporting implementation of coinductive operations on cyclic objects. We formally define the operational semantics of COFJ, and provide a handful of examples showing the expressive power of regular corecursion; such a mechanism promotes a novel programming style particularly well-suited for implementing cyclic data structures, and for supporting coinductive reasoning.

[deepliu61] Prajapati R. Wu D. Liu X., Li X. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. Technical report.

Abstract: Compilers are among the most fundamental programming tools for building software. However, production compilers remain buggy. Fuzz testing is often

leveraged with newly-generated, or mutated inputs in order to find new bugs or security vulnerabilities. In this paper, we propose a grammar-based fuzzing tool called DEEPFUZZ. Based on a generative Sequence-to-Sequence model, DEEPFUZZ automatically and continuously generates well-formed C programs. We use this set of new C programs to fuzz off-the-shelf C compilers, e.g., GCC and Clang/LLVM. We present a detailed case study to analyze the success rate and coverage improvement of the generated C programs for fuzz testing. We analyze the performance of DEEPFUZZ with three types of sampling methods as well as three types of generation strategies. Consequently, DEEPFUZZ improved the testing efficacy in regards to the line, function, and branch coverage. In our preliminary study, we found and reported 8 bugs of GCC, all of which are actively being addressed by developers. 2019, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[denodann113] Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational cost semantics for functional languages with inductive types. Technical report.

Abstract: A central method for analyzing the asymptotic complexity of a functional program is to extract and then solve a recurrence that expresses evaluation cost in terms of input size. The relevant notion of input size is often specific to a datatype, with measures including the length of a list, the maximum element in a list, and the height of a tree. In this work, we give a formal account of the extraction of cost and size recurrences from higher-order functional programs over inductive datatypes. Our approach allows a wide range of programmer-specified notions of size, and ensures that the extracted recurrences correctly predict evaluation cost. To extract a recurrence from a program, we first make costs explicit by applying a monadic translation from the source language to a complexity language, and then abstract datatype values as sizes. Size abstraction can be done semantically, working in models of the complexity language, or syntactically, by adding rules to a preorder judgement. We give several different models of the complexity language, which support different notions of size. Additionally, we prove by a logical relations argument that recurrences extracted by this process are upper bounds for evaluation cost; the proof is entirely syntactic and therefore applies to all of the models we consider.

[deripoul124] Casper Bach Poulsen and Peter D. Mosses. Deriving pretty-big-step semantics from small-step semantics. Technical report.

Abstract: Big-step semantics for languages with abrupt termination and/or divergence suffer from a serious duplication problem, addressed by the novel 'pretty-big-step' style presented by Chargueraud at ESOP' 13. Such rules are less concise than corresponding small-step rules, but they have the same advantages as big-step rules for program correctness proofs. Here, we show how to automatically derive pretty-big-step rules directly from small-step rules by 'refocusing'. This gives the best of both worlds: we only need to write the relatively concise small-step specifications, but our reasoning can be big-step as well as small-step. The use of strictness annotations to derive small-step congruence rules gives further conciseness.

[dynascha76] Max Schaefer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. Technical report.

Abstract: We present an analysis for identifying determinate variables and expressions that always have the same value at a given program point. This information can be exploited by client analyses and tools to, e. g., identify dead code or specialize uses of dynamic language constructs such as e v a l, replacing them with equivalent static constructs. Our analysis is completely dynamic and only needs to observe a single execution of the program, yet the determinacy facts it infers hold for any execution. We present a formal soundness proof of the analysis for a simple imperative language, and a prototype implementation that handles full JavaScript. Finally, we report on two case studies that explored how static analysis for JavaScript could

leverage the information gathered by dynamic determinacy analysis. We found that in some cases scalability of static pointer analysis was improved dramatically, and that many uses of runtime code generation could be eliminated.

[featraco88] Buchs D. Racordon D. Featherweight swift: A core calculus for swift's type system. Technical report.

Abstract: Swift is a modern general-purpose programming language, designed to be a replacement for C-based languages. Although primarily directed at development of applications for Apple's operating systems, Swift's adoption has been growing steadily in other domains, ranging from server-side services to machine learning. This success can be partly attributed to a rich type system that enables the design of safe, fast, and expressive programming interfaces. Unfortunately, this richness comes at the cost of complexity, setting a high entry barrier to exploit Swift's full potential. Furthermore, existing documentation typically only relies on examples, leaving new users with little help to build a deeper understanding of the underlying rules and mechanisms. This paper aims to tackle this issue by laying out the foundations for a formal framework to reason about Swift's type system. We introduce Featherweight Swift, a minimal language stripped of all features not essential to describe its typing rules. Featherweight Swift features classes and protocol inheritance, supports retroactive modeling, and emulates Swift's overriding mechanisms. Yet its formalization fits on a few pages. We present Featherweight Swift's syntax and semantics. We then elaborate on the usability of our framework to reason about Swift's features, future extensions, and implementation by discussing a bug in Swift's compiler, discovered throughout the design of our calculus. 2020 ACM.

[findchen42] Chengnian Sun. Finding compiler bugs via live code mutation. Technical report.

Abstract: Validating optimizing compilers is challenging because it is hard to generate valid test programs (i.e., those that do not expose any undefined behavior). Equivalence Modulo Inputs (EMI) is an effective, promising methodology to tackle this problem. Given a test program with some inputs, EMI mutates the program to derive variants that are semantically equivalent w.r.t. these inputs. The state-of-the-art instantiations of EMI are Orion and Athena, both of which rely on deleting code from or inserting code into code regions that are not executed under the inputs. Although both have demonstrated their ability in finding many bugs in GCC and LLVM, they are still limited due to their mutation strategies that operate only on dead code regions. This paper presents a novel EMI technique that allows mutation in the entire program (i.e., both live and dead regions). By removing the restriction of mutating only the dead regions, our technique significantly increases the EMI variant space. It also helps to more thoroughly stress test compilers as compilers must optimize mutated live code, whereas mutated dead code might be eliminated. Finally, our technique also makes compiler bugs more noticeable as miscompilations on mutated dead code may not be observable. We have realized the proposed technique in Hermes. The evaluation demonstrates Hermes s effectiveness. In 13 months, Hermes found 168 confirmed, valid bugs in GCC and LLVM, of which 132 have already been fixed.

[fixeede90] E de Vries. Fixed-point vs proof theoretic approach to inference systems. Technical report.

Abstract: We recall the basic definitions and results of elementary fixed point theory: we show that the least fixed point of a monotone function F over a powerset is given by the intersection of all F-closed sets, and dually that the greatest fixed point of F is given by the union of all F-consistent sets. We recall the def-initions of continuity and cocontinuity, give an alternative construction of least and greatest fixed points for continuous and cocontinuous functions, and give examples of non-continuous and non-cocontinuous functions. We recall the definition of an inference system , show that the associated function F is always monotone and hence has a least fixed point F (the inductive interpretation of) and greatest fixed point F (the coinductive interpretation of). We give conditions over that guarantee that F is continous or cocontinuous, and give examples of inference systems that do not meet these conditions. We show that the inductive and coinductive interpretation of give rise to the proof principles of rule induction and Park's principle. Finally, we show that the set of the conclusions of finite proofs over corresponds to the inductive interpretation of , and dually that the set of the conclusions of finite and infinite proofs over corresponds to its coinductive interpretation; i.e. that F = and F = The results in this report are not new, but we hope the reader finds the exposition and examples useful.

[flagpoul115] Casper Bach Poulsen and Peter D. Mosses. Flag-based big-step semantics. Technical report.

Abstract: Structural operational semantic specifications come in different styles: small-step and big-step. A problem with the big-step style is that specifying divergence and abrupt termination gives rise to annoying duplication. We present a novel approach to representing divergence and abrupt termination in big-step semantics using status flags. This avoids the duplication problem, and uses fewer rules and premises for representing divergence than previous approaches in the literature. (C) 2016 Elsevier Inc. All rights reserved.

[flexcicc61] Zucca E. Ciccone L., Dagnino F. Flexible coinduction in agda. Technical report.

Abstract: We provide an Agda library for inference systems, also supporting their recent generalization allowing flexible coinduction, that is, interpretations which are neither inductive, nor purely coinductive. A specific inference system can be obtained as an instance by writing a set of meta-rules, in an Agda format which closely resembles the usual one. In this way, the user gets for free the related properties, notably the inductive and coinductive intepretation and the corresponding proof principles. Moreover, a significant modularity is achieved. Indeed, rather than being defined from scratch and with a built-in interpretation, an inference system can also be obtained by composition operators, such as union and restriction to a smaller universe, and its semantics can be modularly chosen as well. In particular, flexible coinduction is obtained by composing in a certain way the interpretations of two inference systems. We illustrate the use of the library by several examples. The most significant one is a big-step semantics for the -calculus, where flexible coinduction allows to obtain a special result ( ) for all and only the diverging computations, and the proof of equivalence with small-step semantics is carried out by relying on the proof principles offered by the library. Luca Ciccone, Francesco Dagnino, and Elena Zucca.

[flexdagn83] Dagnino F. Flexible coinduction for infinite behaviour. Technical report.

Abstract: Generalized inference systems have been recently defined to overcome the strong dichotomy between inductive and coinductive interpretations. They support a flexible form of coinduction, subsuming even induction, which allows one to mediate between the two standard semantics. Recently, this framework has been successfully adopted to define semantic judgments which uniformly model finite and infinite computations. In this communication, we survey these results and outline directions for further developments. 2018 CEUR-WS. All rights reserved.

[flexdagn93] F. R. A. N. C. E. S. C. O. DAGNINO, D. A. V. I. D. E. ANCONA, and E. L. E. N. A. ZUCCA. Flexible coinductive logic programming. Technical report.

Abstract: Recursive definitions of predicates are usually interpreted either inductively or coinductively. Recently, a more powerful approach has been proposed, calledflexible coinduction, to express a variety of intermediate interpretations, necessary in some cases to get the correct meaning. We provide a detailed formal account of an extension of logic programming supporting flexible coinduction. Syntactically, programs are enriched bycoclauses, clauses with a special meaning used to tune the interpretation of predicates. As usual, the declarative semantics can be expressed as a fixed point which, however, is not necessarily the least, nor the greatest one, but is determined by the coclauses. Correspondingly, the operational semantics is a combination of standard SLD resolution and coSLD resolution. We prove that the operational semantics is sound and complete with respect to declarative semantics restricted to finite comodels.

[formkunz74] Forster Y. Kunze F., Smolka G. Formal small-step verification of a call-by-value lambda calculus machine. Technical report.

Abstract: We formally verify an abstract machine for a call-by-value -calculus with de Bruijn terms, simple substitution, and small-step semantics. We follow a stepwise refinement approach starting with a naive stack machine with substitution. We then refine to a machine with closures, and finally to a machine with a heap providing structure sharing for closures. We prove the correctness of the three refinement steps with compositional small-step bottom-up simulations. There is an accompanying Coq development verifying all results. 2018, Springer Nature Switzerland AG.

[formmizu61] Masayuki Mizuno and Eijiro Sumii. Formal verifications of call-by-need and call-by-name evaluations with mutual recursion. Technical report.

Abstract: We present new proofs-formalized in the Coq proof assistant-of the correspondence among call-by-need and (various definitions of) call-by-name evaluations of lambda-calculus with mutually recursive bindings.For non-strict languages, the equivalence between high-level specifications (call-by-name) and typical implementations (call-by-need) is of foundational interest. A particular milestone is Launchbury's natural semantics of call-by-need evaluation and proof of its adequacy with respect to call-by-name denotational semantics, which are recently formalized in Isabelle/HOL by Breitner (2018). Equational theory by Ariola et al. is another well-known formalization of call-by-need. Mutual recursion is especially challenging for their theory: reduction is complicated by the traversal of dependency (the need relation), and the correspondence of call-by-name and call-by-need reductions becomes non-trivial, requiring sophisticated structures such as graphs or infinite trees.In this paper, we give arguably simpler proofs solely based on (finite) terms and operational semantics, which are easier to handle for proof assistants (Coq in our case). Our proofs can be summarized as follows: (1) we prove the equivalence between Launchbury's call-by-need semantics and heap-based call-by-name natural semantics, where we define a sufficiently (but not too) general correspondence between the two heaps, and (2) we also show the correspondence among three styles of call-by-name semantics: (i) the natural semantics used in (1); (ii) closure-based natural semantics that informally corresponds to Launchbury's denotational semantics; and (iii) conventional substitution-based semantics.

[formmizu65] Sumii E. Mizuno M. Formal verification of functional programs performing infinite input/output. Technical report.

Abstract: Although formal verification of compilers is extensively studied, compilers for higher-order functional programming languages with side effects such as input and output are rarely verified. This is due to the difficulty of formalizing the semantics of programs performing infinite input and output. We have mechanically verified the K-normalization of call-by-value higher-order functional programs with recursive functions, pairs, and external function calls that can possibly cause side effects, by the Coq proof assistant. K-normalization is a program transformation that gives explicit names to all subexpressions via let-expressions. Its for-malization is non-trivial because of the manipulation of bindings. We defined the meanings of programs as infinite sequences of external function calls, using coinductive big-step operational semantics. We also adopted de Bruijn indices by comparison with other techniques to represent bindings.

[foundagn80] Francesco Dagnino. Foundations of regular coinduction. Technical report.

Abstract: Inference systems are a widespread framework used to define possibly recursive predicates by means of inference rules. They allow both inductive and coinductive interpretations that are fairly well-studied. In this paper, we consider a middle way interpretation, called regular, which combines advantages of both approaches: it allows non-well-founded reasoning while being finite. We show that the natural proof-theoretic definition of the regular interpretation, based on regular trees, coincides with a rational fixed point. Then, we provide an equivalent inductive characterization, which leads to an algorithm which looks for a regular derivation of a judgment. Relying on these results, we define proof techniques for regular reasoning: the regular coinduction principle, to prove completeness, and an inductive technique to prove soundness, based on the inductive characterization of the regular interpretation. Finally, we show the regular approach can be smoothly extended to inference systems with corules, a recently introduced, generalised framework, which allows one to refine the coinductive interpretation, proving that also this flexible regular interpretation admits an equivalent inductive characterisation.

[fromciob96] Ciobâcă S. From small-step semantics to big-step semantics, automatically. Technical report.

Abstract: Small-step semantics and big-step semantics are two styles for operationally defining the meaning of programming languages. Small-step semantics are given as a relation between program configurations that denotes one computational step; big-step semantics are given as a relation directly associating to each program configuration the corresponding final configuration. Small-step semantics are useful for making precise reasonings about programs, but reasoning in big-step semantics is easier and more intuitive. When both small-step and big-step semantics are needed for the same language, a proof of the fact that the two semantics are equivalent should also be provided in order to trust that they both define the same language. We show that the big-step semantics can be automatically obtained from the small-step semantics when the small-step semantics are given by inference rules satisfying certain assumptions that we identify. The transformation that we propose is very simple and we show that when the identified assumptions are met, it is sound and complete in the sense that the two semantics are equivalent. For a strict subset of the identified assumptions, we show that the resulting big-step semantics is sound but not necessarily complete. We discuss our transformation on a number of examples. 2013 Springer-Verlag Berlin Heidelberg.

[funcowen113] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. Technical report.

Abstract: When doing an interactive proof about a piece of software, it is important that the underlying programming language's semantics does not make the proof unnecessarily difficult or unwieldy. Both small-step and big-step semantics are commonly used, and the latter is typically given by an inductively defined relation. In this paper, we consider an alternative: using a recursive function akin to an interpreter for the language. The advantages include a better induction theorem, less duplication, accessibility to ordinary functional programmers, and the ease of doing symbolic simulation in proofs via rewriting. We believe that this style of semantics is well suited for compiler verification, including proofs of divergence preservation. We do not claim the invention of this style of semantics: our contribution here is to clarify its value, and to explain how it supports several language features that might appear to require a relational or small-step approach. We illustrate the technique on a simple imperative language with C-like for-loops and a break statement, and compare it to a variety of other approaches. We also provide ML and lambda-calculus based examples to illustrate its generality.

[geneanco114] Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing inference systems by coaxioms. Technical report.

Abstract: We introduce a generalized notion of inference system to support structural recursion on non well-founded datatypes. Besides axioms and inference rules with the usual meaning, a generalized inference system allows coaxioms, which are, intuitively, axioms which can only be applied at infinite depth in a proof tree. This notion nicely subsumes standard inference systems and their inductive and coinductive interpretation, while providing more flexibility. Indeed, the classical results on the existence and constructive characterization of least and greatest fixed points can be extended to our generalized framework, interpreting recursive definitions as fixed points which are not necessarily the least, nor the greatest one. This allows formal reasoning in cases where the inductive and coinductive interpretation do not provide the intended meaning, or are mixed together.

[grampatr126] Patrick Cousot. Grammar semantics, analysis and parsing by abstract interpretation. Technical report.

Abstract: We study abstract interpretations of a fixpoint protoderivation semantics defining the maximal derivations of a transitional semantics of context-free grammars akin to pushdown automata. The result is a hierarchy of bottom-up or top-down semantics refining the classical equational and derivational language semantics and including Knuth grammar problems, classical grammar flow analysis algorithms and parsing algorithms.

[greaach107] A Charguéraud. Great-step semantics. Technical report.

Abstract: The small-step semantics and the big-step semantics are the two main approaches to describing the operational semantics of a pro-gramming language. When specifying a language with exceptions, or when specifying the behavior of non-terminating programs, big-step definitions exhibit a significant amount of duplication that does not appear in the small-step definitions. This duplication makes the big-step definitions more tedious to write. Moreover, it can grow the size of the trusted base of a formal development and leads to redundancy in formal proofs based on the semantics. In this paper, we introduce the great-step semantics, which is a variant of the big-step semantics that eliminates duplication in the definitions. The great-step semantics consists of one set of rules that, when inter-preted inductively, specifies the behavior of terminating programs, and, when interpreted coinductively, specifies non-termination. While the great-step semantics eliminates the duplication in the definitions, proofs still involve some redundancy. Indeed, one needs to carry out a first proof for terminating programs, by induction, and a second proof for diverging programs, by coinduction. To allow factorizing these two proofs, we introduce the combined great-step semantics. It is obtained by taking the rules from the great-step semantics, interpreted coinductively, and integrating in these rules a partial order on the output of the terms. This partial order is used to ensure the finiteness of sub-derivations associated with terminating terms. We illustrate the interest of the combined semantics through two developments: a formalization of the encoding of exceptions into sum types, and a formalization of an interpreter for the lambda-calculus with exceptions. In each of the two proofs, we are able to handle both terminating and diverging terms in a single proof.

[habifga9] F Gava. Habilitation thesis. Technical report.

Abstract: This habilitation thesis gives an overview of some recent results obtained by the author together with various collaborators in the area of parameterized approximation algorithms. This research area in the intersection of fixed-parameter tractability and approximation algorithms has gained growing attention in recent years. While a large variety of algorithmic topics have seen advancements using the paradigm of parameterized approximations (for an overview see the recent survey in [9]), the author s contributions are mostly concentrated on the topics of network design and clustering. Accordingly, this thesis presents the author s results on these two topics in separate sections. Each of these two sections gives a brief overview of the obtained results, after which the used techniques are presented in more detail.

[howanco115] Ancona D. How to prove type soundness of java-like languages without forgoing big-step semantics. Technical report.

Abstract: Small-step operational semantics is the most commonly employed formalism for proving type soundness of statically typed programming languages, because of its ability to distinguish stuck from non-terminating computations, as opposed to big-step operational semantics. Despite this, big-step operational semantics is more abstract, and more useful for specifying interpreters. In previous work we have proposed a new proof technique to prove type soundness of a Java-like language expressed in terms of its big-step operational semantics. However the presented proof is rather involved, since it requires showing that the set of proof trees defining the semantic judgment forms a complete metric space when equipped with a specific distance function. In this paper we propose a more direct and abstract approach that exploits a standard and general compactness property of the metric space of values, that allows approximation of the coinductive big-step semantics in terms of the small-step one; in this way type soundness can be proved by standard mathematical induction. 2014 ACM.

[ideaanco71] Lagorio G. Ancona D. Idealized coinductive type systems for imperative object-oriented programs. Technical report.

Abstract: In recent work we have proposed a novel approach to define idealized type systems for object-oriented languages, based on abstract compilation of programs into Horn formulas which are interpreted w.r.t. the coinductive (that is, the greatest) Herbrand model. In this paper we investigate how this approach can be applied also in the presence of imperative features. This is made possible by considering a natural translation of Static Single Assignment intermediate form programs into Horn formulas, where functions correspond to union types. 2011 EDP Sciences.

[impepoul3] Casper Bach Poulsen, Peter D. Mosses, and Paolo Torrini. Imperative polymorphism by store-based types as abstract interpretations. Technical report.

Abstract: Dealing with polymorphism in the presence of imperative features is a long-standing open problem for Hindley-Milner type systems. A widely adopted approach is the value restriction, which inhibits polymorphic generalisation and unfairly rejects various programs that cannot go wrong. We consider abstract interpretation as a tool for constructing safe and precise type systems, and investigate how to derive store-based types by abstract interpretation. We propose store-based types as a type discipline that holds potential for interesting and flexible alternatives to the value restriction.

[implraco94] Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. Implementation strategies for mutable value semantics. Technical report.

Abstract: Mutable value semantics is a programming discipline that upholds the independence of values to support local reasoning. In the discipline's strictest form, references become second-class citizens: they are only created implicitly, at function boundaries, and cannot be stored in variables or object fields. Hence, variables can never share mutable state. Unlike pure functional programming, however, mutable value semantics allows part-wise in-place mutation, thereby eliminating the memory traffic usually associated with functional updates of immutable data.This paper presents implementation strategies for compiling programs with mutable value semantics into efficient native code. We study Swift, a programming language based on that discipline, through the lens of a core language that strips some of Swift's features to focus on the semantics of its value types. The strategies that we introduce leverage the inherent properties of mutable value semantics to unlock aggressive optimizations. Fixed-size values are allocated on the stack, thereby enabling numerous off-the-shelf compiler optimizations, while dynamically sized containers use copy-on-write to mitigate copying costs.

[imprricc41] Riccardo Cantoro. Improved test solutions for cots-based systems in space applications. Technical report.

Abstract: In order to widen the spectrum of available products, companies involved in space electronics are exploring the possible adoption of COTS components instead of space-qualified ones. However, the adoption of COTS devices and boards requires suitable solutions able to guarantee the same level of dependability. A mix of different solutions can be considered for this purpose. Test techniques play a major role, since they must guarantee that a high percentage of permanent faults can be detected (both at the end of the manufacturing and during the mission) while matching several constraints in terms of system accessibility and hardware complexity. In this paper we focus on the test of the electronics used within launchers, and outline an approach based on Software-based Self-test. The proposed solutions are currently being adopted within the MaMMoTH-Up project, targeting the development of an innovative COTS-based system to be used on the Ariane5 launcher. The approach aims at testing both the OR1200 processor and the different peripheral modules adopted in the system, while providing new techniques for the identification of safe faults. The results show the effectiveness and current limitations of the method, also including a comparison between functional and structural test approaches.

[induvelt48] Voorneveld N.F.W. Veltri N. Inductive and coinductive predicate liftings for effectful programs. Technical report.

Abstract: We formulate a framework for describing behaviour of effectful higher-order recursive programs. Examples of effects are implemented using effect operations, and include: execution cost, nondeterminism, global store and interaction with a user. The denotational semantics of a program is given by a coinductive tree in a monad, which combines potential return values of the program in terms of effect operations. Using a simple test logic, we construct two sorts of predicate liftings, which lift predicates on a result type to predicates on computations that produce results of that type, each capturing a facet of program behaviour. Firstly, we study inductive predicate liftings which can be used to describe effectful notions of total correctness. Secondly, we study coinductive predicate liftings, which describe effectful notions of partial correctness. The two constructions are dual in the sense that one can be used to disprove the other. The predicate liftings are used as a basis for an endogenous logic of behavioural properties for higher-order programs. The program logic has a derivable notion of negation, arising from the duality of the two sorts of predicate liftings, and it generates a program equivalence which subsumes a notion of bisimilarity. Appropriate definitions of inductive and coinductive predicate liftings are given for a multitude of effect examples. The whole development has been fully formalized in the Agda proof assistant. N. Veltri & N.F.W. Voorneveld

[inficorr104] Frassetto F. Corradi A. Infinite derivations as failures. Technical report.

Abstract: When operating on cyclic data, programmers have to take care in assuring that their programs will terminate; in our opinion, this is a task for the interpreter. We present a Prolog meta-interpreter that checks for the presence of cyclic computations at runtime and returns a failure if this is the case, thus allowing inductive predicates to properly deal with cyclic terms.

[infiugo114] Ugo Dal Lago. Infinitary lambda calculi from a linear perspective. Technical report.

Abstract: We introduce a linear infinitary -calculus, called &ell; , in which two exponential modalities are available, the first one being the usual, finitary one, the other being the only construct interpreted coinductively. The obtained calculus embeds the infinitary applicative -calculus and is universal for computations over infinite strings. What is particularly interesting about &ell; , is that the refinement induced by linear logic allows to restrict both modalities so as to get calculi which are terminating inductively and productive coinductively. We exemplify this idea by analysing a fragment of &ell; built around the principles of SLL and 4LL. Interestingly, it enjoys confluence, contrarily to what happens in ordinary infinitary -calculi.

[infiugo98] Ugo Dal Lago. Infinitary $\lambda$-calculi from a linear perspective (long version). Technical report.

Abstract: We introduce a linear infinitary $\lambda$-calculus, called $\ell\Lambda_{\infty}$, in which two exponential modalities are available, the first one being the usual, finitary one, the other being the only construct interpreted coinductively. The obtained calculus embeds the infinitary applicative $\lambda$-calculus and is universal for computations over infinite strings. What is particularly interesting about $\ell\Lambda_{\infty}$, is that the refinement induced by linear logic allows to restrict both modalities so as to get calculi which are terminating inductively and productive coinductively. We exemplify this idea by analysing a fragment of $\ell\Lambda$ built around the principles of $\mathsf{SLL}$ and $\mathsf{4LL}$. Interestingly, it enjoys confluence, contrarily to what happens in ordinary infinitary $\lambda$-calculi.

[intecohe65] Rowe R.N.S. Cohen L. Integrating induction and coinduction via closure operators and proof cycles. Technical report.

Abstract: Coinductive reasoning about infinitary data structures has many applications in computer science. Nonetheless developing natural proof systems (especially ones amenable to automation) for reasoning about coinductive data remains a challenge. This paper presents a minimal, generic formal framework that uniformly captures applicable (i.e. finitary) forms of inductive and coinductive reasoning in an intuitive manner. The logic extends transitive closure logic, a general purpose logic for inductive reasoning based on the transitive closure operator, with a dual co-closure operator that similarly captures applicable coinductive reasoning in a natural, effective manner. We develop a sound and complete non-well-founded proof system for the extended logic, whose cyclic subsystem provides the basis for an effective system for automated inductive and coinductive reasoning. To demonstrate the adequacy of the framework we show that it captures the canonical coinductive data type: streams. 2020, Springer Nature Switzerland AG.

[inteliy124] Li yao Xia. Interaction trees: representing recursive and impure programs in coq. Technical report.

Abstract: (ITrees) are a general-purpose data structure for representing the behaviors of recursive programs that interact with their environments. A coinductive variant of free monads, ITrees are built out of uninterpreted events and their continuations. They support compositional construction of interpreters from , which give meaning to events by defining their semantics as monadic actions. ITrees are expressive enough to represent impure and potentially nonterminating, mutually recursive computations, while admitting a rich equational theory of equivalence up to weak bisimulation. In contrast to other approaches such as relationally specified operational semantics, ITrees are executable via code extraction, making them suitable for debugging, testing, and implementing software artifacts that are amenable to formal verification.

[intexia16] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees representing recursive and impure programs in coq. Technical report.

Abstract: Interaction trees (ITrees) are a general-purpose data structure for representing the behaviors of recursive 5 programs that interact with their environments. A coinductive variant of free monads, ITrees are built out of uninterpreted events and their continuations. They support compositional construction of interpreters from event handlers, which give meaning to events by defining their semantics as monadic actions. ITrees are expressive enough to represent impure and potentially nonterminating, mutually recursive computations, while admitting a rich equational theory of equivalence up to weak bisimulation. In contrast to other approaches such as relationally specified operational semantics. ITrees re executable via code extraction, making them suitable for debugging, testing, and implementing software artifacts that are amenable to formal verification.We have implemented ITrees and their associated theory as a Coq library, mechanizing classic domain- and category-theoretic results about program semantics, iteration, monadic structures, and equational reasoning. Although the internals of the library rely heavily on coinductive proofs, the interface hides these details so that clients can use and reason about ITrees without explicit use of Coq's coinduction tactics.To showcase the utility of our theory, we prove the termination-sensitive correctness of a compiler from a simple imperative source language to an assembly-like target whose meanings are given in an ITree-based denotational semantics. Unlike previous results using operational techniques, our bisimulation proof follows straightforwardly by structural induction and elementary rewriting via an equational theory of combinators for control-flow graphs.

[intrsang71] Sangiorgi D. Introduction to bisimulation and coinduction. Technical report.

Abstract: Induction is a pervasive tool in computer science and mathematics for defining objects and reasoning on them. Coinduction is the dual of induction and as such it brings in quite different tools. Today, it is widely used in computer science, but also in other fields, including artificial intelligence, cognitive science, mathematics, modal logics, philosophy and physics. The best known instance of coinduction is bisimulation, mainly employed to define and prove equalities among potentially infinite objects: processes, streams, non-well-founded sets, etc. This book presents bisimulation and coinduction: the fundamental concepts and techniques and the duality with induction. Each chapter contains exercises and selected solutions, enabling students to connect theory with practice. A special emphasis is placed on bisimulation as a behavioural equivalence for processes. Thus the book serves as an introduction to models for expressing processes (such as process calculi) and to the associated techniques of operational and algebraic analysis. D. Sangiorgi 2012.

[intusylv24] Sylvain Boulmé. Intuitionistic refinement calculus. Technical report.

Abstract: Refinement calculi are program logics which formalize the top-down methodology of software development promoted by Dijkstra and Wirth in the early days of structured programming. I present here the shallow embedding of a refinement calculus into constructive type theory. This embedding involves monad transformers and the computational reflexion of weakest-preconditions, using a continuation passing style. It should allow to reason about many programs combining non-functional features (state, exceptions, etc) with purely functional ones (higher-order functions, structural recursion, etc).

[locakosk39] Eric Koskinen and Tachio Terauchi. Local temporal reasoning. Technical report.

Abstract: We present the first method for reasoning about temporal logic properties of higher-order, infinite-data programs. By distinguishing between the finite traces and infinite traces in the specification, we obtain rules that permit us to reason about the temporal behavior of program parts via a type-and-effect system, which is then able to compose these facts together to prove the overall target property of the program. The type system alone is strong enough to derive many temporal safety properties using refinement types and temporal effects. We also show how existing techniques can be used as oracles to provide liveness information (e.g. termination) about program parts and that the type-and-effect system can combine this information with temporal safety information to derive nontrivial temporal properties. Our work has application toward verification of higher-order software, as well as modular strategies for procedural programs.

[mechblaz5] Leroy X. Blazy S. Mechanized semantics for the clight subset of the c language. Technical report.

Abstract: This article presents the formal semantics of a large subset of the C language called Clight. Clight includes pointer arithmetic, struct and union types, C loops and structured switch statements. Clight is the source language of the CompCert verified compiler. The formal semantics of Clight is a big-step operational semantics that observes both terminating and diverging executions and produces traces of input/output events. The formal semantics of Clight is mechanized using the Coq proof assistant. In addition to the semantics of Clight, this article describes its integration in the CompCert verified compiler and several ways by which the semantics was validated. 2009 Springer Science+Business Media B.V.

[mechlero107] Leroy X. Mechanized semantics for compiler verification. Technical report.

Abstract: The formal verification of compilers and related programming tools depends crucially on the availability of appropriate mechanized semantics for the source, intermediate and target languages. In this invited talk, I review various forms of operational semantics and their mechanization, based on my experience with the formal verification of the CompCert C compiler. Springer-Verlag 2012.

[mechtimo19] Timothy Bourke. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. Technical report.

[modeanco14] Zucca E. Ancona D., Dagnino F. Modeling infinite behaviour by corules. Technical report.

Abstract: Generalized inference systems have been recently introduced, and used, among other applications, to define semantic judgments which uniformly model terminating computations and divergence. We show that the approach can be successfully extended to more sophisticated notions of infinite behaviour, that is, to express that a diverging computation produces some possibly infinite result. This also provides a motivation to smoothly extend the theory of generalized inference systems to include, besides coaxioms, also corules, a more general notion for which significant examples were missing until now. We first illustrate the approach on a -calculus with output effects, for which we also provide an alternative semantics based on standard notions, and a complete proof of the equivalence of the two semantics. Then, we consider a more involved example, that is, an imperative Java-like language with I/O primitives. Davide Ancona, Francesco Dagnino, and Elena Zucca.

[modupavi62] Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. Modular relaxed dependencies in weak memory concurrency. Technical report.

Abstract: We present a denotational semantics for weak memory concurrency that avoids thin-air reads, provides data-race free programs with sequentially consistent semantics (DRF-SC), and supports a compositional refinement relation for validating optimisations. Our semantics identifies false program dependencies that might be removed by compiler optimisation, and leaves in place just the dependencies necessary to rule out thin-air reads. We show that our dependency calculation can be used to rule out thin-air reads in any axiomatic concurrency model, in particular C++. We present a tool that automatically evaluates litmus tests, show that we can augment C++ to fix the thin-air problem, and we prove that our augmentation is compatible with the previously used compilation mappings over key processor architectures. We argue that our dependency calculation offers a practical route to fixing the longstanding problem of thin-air reads in the C++ specification.

[moduzako26] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for llvm ir. Technical report.

Abstract: This paper presents a novel formal semantics, mechanized in Coq, for a large, sequential subset of the LLVM IR. In contrast to previous approaches, which use relationally-specified operational semantics, this new semantics is based on monadic interpretation of interaction trees, a structure that provides a more compositional approach to defining language semantics while retaining the ability to extract an executable interpreter. Our semantics handles many of the LLVM IR's non-trivial language features and is constructed modularly in terms of event handlers, including those that deal with nondeterminism in the specification. We show how this semantics admits compositional reasoning principles derived from the interaction trees equational theory of weak bisimulation, which we extend here to better deal with nondeterminism, and we use them to prove that the extracted reference interpreter faithfully refines the semantic model. We validate the correctness of the semantics by evaluating it on unit tests and LLVM IR programs generated by HELIX.

[multallo94] V. Allombert, F. Gava, and J. Tesson. Multi-ml: Programming multi-bsp algorithms in ml. Technical report.

Abstract: bsp is a bridging model between abstract execution and concrete parallel systems. Structure and abstraction brought by bsp allow to have portable parallel programs with scalable performance predictions, without dealing with low-level details of architectures. In the past, we designed bsml for programming bsp algorithms in ml. However, the simplicity of the bsp model does not fit the complexity of today's hierarchical architectures such as clusters of machines with multiple multi-core processors. The multi-bsp model is an extension of the bsp model which brings a tree-based view of nested components of hierarchical architectures. To program multi-bsp algorithms in ml, we propose the multi-ml language as an extension of bsml where a specific kind of recursion is used to go through a hierarchy of computing nodes. We define a formal semantics of the language and present preliminary experiments which show performance improvements with respect to bsml.

[multjan73] Jan Hoffmann. Multivariate amortized resource analysis. Technical report.

Abstract: We study the problem of automatically analyzing the worst-case resource usage of procedures with several arguments. Existing automatic analyses based on amortization or sized types bound the resource usage or result size of such a procedure by a sum of unary functions of the sizes of the arguments. In this article we generalize this to arbitrary multivariate polynomial functions thus allowing bounds of the form mn which had to be grossly overestimated by m &plus; n before. Our framework even encompasses bounds like i,j n mi mj where the mi are the sizes of the entries of a list of length n. This allows us for the first time to derive useful resource bounds for operations on matrices that are represented as lists of lists and to considerably improve bounds on other superlinear operations on lists such as longest common subsequence and removal of duplicates from lists of lists. Furthermore, resource bounds are now closed under composition which improves accuracy of the analysis of composed programs when some or all of the components exhibit superlinear resource or size behavior. The analysis is based on a novel multivariate amortized resource analysis. We present it in form of a type system for a simple first-order functional language with lists and trees, prove soundness, and describe automatic type inference based on linear programming. We have experimentally validated the automatic analysis on a wide range of examples from functional programming with lists and trees. The obtained bounds were compared with actual resource consumption. All bounds were asymptotically tight, and the constants were close or even identical to the optimal ones.

[noncohe55] Liron Cohen. Non-well-founded deduction for induction and coinduction. Technical report.

Abstract: Induction and coinduction are both used extensively within mathematics and computer science. Algebraic formulations of these principles make the duality between them apparent, but do not account well for the way they are commonly used in deduction. Generally, the formalization of these reasoning methods employs inference rules that express a general explicit (co)induction scheme. Non-well-founded proof theory provides an alternative, more robust approach for formalizing implicit (co)inductive reasoning. This approach has been extremely successful in recent years in supporting implicit inductive reasoning, but is not as well-developed in the context of coinductive reasoning. This paper reviews the general method of non-well-founded proofs, and puts forward a concrete natural framework for (co)inductive reasoning, based on (co)closure operators, that offers a concise framework in which inductive and coinductive reasoning are captured as we intuitively understand and use them. Through this framework we demonstrate the enormous potential of non-well-founded deduction, both in the foundational theoretical exploration of (co)inductive reasoning and in the provision of proof support for (co)inductive reasoning within (semi-)automated proof tools.

[objebe3] B. Emir. Object-oriented pattern matching. Technical report.

Abstract: Pattern matching is a programming language construct considered essential in functional programming. Its purpose is to inspect and decompose data. Instead, object-oriented programming languages do not have a dedicated construct for this purpose. A possible reason for this is that pattern matching is useful when data is defined separately from operations on the data a scenario that clashes with the object-oriented motto of grouping data and operations. However, programmers are frequently confronted with situations where there is no alternative to expressing data and operations separately because most data is neither stored in nor does it originate from an object-oriented context. Consequently, object-oriented programmers, too, are in need for elegant and concise solutions to the problem of decomposing data. To this end, we propose a built-in pattern matching construct compatible with object-oriented programming. We claim that it leads to more concise and readable code than standard object-oriented approaches. A pattern in our approach is any computable way of testing and deconstructing an object and binding relevant parts to local names. We introduce pattern matching in two variants, case classes and extractors. We compare the readability, extensibility and performance of built-in pattern matching in these two variants with standard decomposition techniques. It turns out that standard object-oriented approaches to decomposing data are not extensible. Case classes, which have been studied before, require a low notational overhead, but expose their representation, making them hard to change later. The novel extractor mechanism offers loose coupling and extensibility, but comes with a performance overhead. We present a formalization of object-oriented pattern matching with extractors. This is done by giving definitions and proving standard properties for a calculus that provides pattern matching as described before. We then give a formal, optimizing translation from the calculus including pattern matching to its fragment without pattern matching, and prove it correct. Finally, we consider non-obvious interactions between the pattern matching and parametric polymorphism. We review the technique of generalized algebraic data types from functional programming, and show how it can be carried over to the object-oriented style. The main tool is the extension of the type system with subtype constraints, which leads to a very expressive metatheory. Through this theory, we are able to express patterns that operate on existentially quantified types purely by universally quantified extractors.

[oneana11] ANA-NLN Film Service Review Committee. One step at a time. Technical report.

[onerosu85] Grigore Rosu, Andrei Stefanescu, Stefan Ciobaca, and Brandon M. Moore. One-path reachability logic. Technical report.

Abstract: This paper introduces (one-path) reachability logic, a language-independent proof system for program verification, which takes an operational semantics as axioms and derives reachability rules, which generalize Hoare triples.

This system improves on previous work by allowing operational semantics given with conditional rewrite rules, which are known to support all major styles of operational semantics. In particular, Kahn's big-step and Plotkin's small-step semantic styles are now supported. The reachability logic proof system is shown sound (i.e., partially correct) and (relatively) complete. Reachability logic thus eliminates the need to independently define an axiomatic and an operational semantics for each language, and the non-negligible effort to prove the former sound and complete w.r.t. the latter. The soundness result has also been formalized in Coq, allowing reachability logic derivations to serve as formal proof certificates that rely only on the operational semantics.

[onevese13] Fisher K. Vesely F. One step at a time: A functional derivation of small-step evaluators from big-step counterparts. Technical report.

Abstract: Big-step and small-step are two popular flavors of operational semantics. Big-step is often seen as a more natural transcription of informal descriptions, as well as being more convenient for some applications such as interpreter generation or optimization verification. Small-step allows reasoning about non-terminating computations, concurrency and interactions. It is also generally preferred for reasoning about type systems. Instead of having to manually specify equivalent semantics in both styles for different applications, it would be useful to choose one and derive the other in a systematic or, preferably, automatic way. Transformations of small-step semantics into big-step have been investigated in various forms by Danvy and others. However, it appears that a corresponding transformation from big-step to small-step semantics has not had the same attention. We present a fully automated transformation that maps big-step evaluators written in direct style to their small-step counterparts. Many of the steps in the transformation, which include CPS-conversion, defunctionalisation, and various continuation manipulations, mirror those used by Danvy and his co-authors. For many standard languages, including those with either call-by-value or call-by-need and those with state, the transformation produces small-step semantics that are close in style to handwritten ones. We evaluate the applicability and correctness of the approach on 20 languages with a range of features. The Author(s) 2019.

[onevese8] Ferdinand Vesely and Kathleen Fisher. One step at a time a functional derivation of small-step evaluators from big-step counterparts. Technical report.

Abstract: Big-step and small-step are two popular flavors of operational semantics. Big-step is often seen as a more natural transcription of informal descriptions, as well as being more convenient for some applications such as interpreter generation or optimization verification. Small-step allows reasoning about non-terminating computations, concurrency and interactions. It is also generally preferred for reasoning about type systems. Instead of having to manually specify equivalent semantics in both styles for different applications, it would be useful to choose one and derive the other in a systematic or, preferably, automatic way.Transformations of small-step semantics into big-step have been investigated in various forms by Danvy and others. However, it appears that a corresponding transformation from big-step to small-step semantics has not had the same attention. We present a fully automated transformation that maps big-step evaluators written in direct style to their small-step counterparts. Many of the steps in the transformation, which include CPS-conversion, defunctionalisation, and various continuation manipulations, mirror those used by Danvy and his co-authors. For many standard languages, including those with either call-by-value or call-by-need and those with state, the transformation produces small-step semantics that are close in style to handwritten ones. We evaluate the applicability and correctness of the approach on 20 languages with a range of features.

[ontilia51] Ilias Garnier. On the reaction time of some synchronous systems. Technical report.

Abstract: This paper presents an investigation of the notion of reaction time in some synchronous systems. A state-based description of such systems is given, and the reaction time of such systems under some classic composition primitives is studied. Reaction time is shown to be non-compositional in general. Possible solutions are proposed, and applications to verification are discussed. This framework is illustrated by some examples issued from studies on real-time embedded systems.

[operdani60] Nils Anders Danielsson. Operational semantics using the partiality monad. Technical report.

Abstract: The operational semantics of a partial, functional language is often given as a relation rather than as a function. The latter approach is arguably more natural: if the language is functional, why not take advantage of this when defining the semantics? One can immediately see that a functional semantics is deterministic and, in a constructive setting, computable.This paper shows how one can use the coinductive partiality monad to define big-step or small-step operational semantics for lambda-calculi and virtual machines as total, computable functions (total definitional interpreters). To demonstrate that the resulting semantics are useful type soundness and compiler correctness results are also proved. The results have been implemented and checked using Agda, a dependently typed programming language and proof assistant.

[ordejérô28] Jérôme Vouillon. Order theory for big-step semantics. Technical report.

Abstract: We show that tools from order theory, such as Kleene fixpoint theorem, can be used to define bigstep semantics that simultaneously account for both converging and diverging behaviors of programs. These semantics remain very concrete. In particular, values are defined syntactically: the semantics of a function abstraction is a function closure rather than some abstract continuous function.

[partthor49] Thorsten Altenkirch. Partiality, revisited: The partiality monad as a quotient inductive-inductive type. Technical report.

Abstract: Capretta's delay monad can be used to model partial computations, but it has the "wrong" notion of built-in equality, strong bisimilarity. An alternative is to quotient the delay monad by the "right"notion of equality, weak bisimilarity. However, recent work by Chapman et al. suggests that it is impossible to define a monad structure on the resulting construction in common forms of type theory without assuming (instances of) the axiom of countable choice. Using an idea from homotopy type theory—a higher inductive-inductive type—we construct a partiality monad without relying on countable choice. We prove that, in the presence of countable choice, our partiality monad is equivalent to the delay monad quotiented by weak bisimilarity. Furthermore we outline several applications.

[partthor89] Thorsten Altenkirch. Partiality, revisited. Technical report.

Abstract: Capretta s delay monad can be used to model partial computations, but it has the wrong notion of built-in equality, strong bisimilarity. An alternative is to quotient the delay monad by the right notion of equality, weak bisimilarity. However, recent work by Chapman et al. suggests that it is impossible to define a monad structure on the resulting construction in common forms of type theory without assuming (instances of) the axiom of countable choice. Using an idea from homotopy type theory a higher inductive-inductive type we construct a partiality monad without relying on countable choice. We prove that, in the presence of countable choice, our partiality monad is equivalent to the delay monad quotiented by weak bisimilarity. Furthermore we outline several applications.

[posidju89] D Juliano. Position paper: Towards transparent machine learning. Technical report.

Abstract: Transparent machine learning is introduced as an alternative form of machine learning, where both the model and the learning system are represented in source code form. The goal of this project is to enable direct human understanding of machine learning models, giving us the ability to learn, verify, and refine them as programs. If solved, this technology could represent a best-case scenario for the safety and security of AI systems going forward.

[pretarth35] Arthur Charguéraud. Pretty-big-step semantics. Technical report.

Abstract: In spite of the popularity of small-step semantics, big-step semantics remain used by many researchers. However, big-step semantics suffer from a serious duplication problem, which appears as soon as the semantics account for exceptions and/or divergence. In particular, many premises need to be copy-pasted across several evaluation rules. This duplication problem, which is particularly visible when scaling up to full-blown languages, results in formal definitions growing far bigger than necessary. Moreover, it leads to unsatisfactory redundancy in proofs. In this paper, we address the problem by introducing pretty-big-step semantics. Pretty-big-step semantics preserve the spirit of big-step semantics, in the sense that terms are directly related to their results, but they eliminate the duplication associated with big-step semantics.

[probdal68] Ugo Dal Lago and Margherita Zorzi. Probabilistic operational semantics for the lambda calculus. Technical report.

Abstract: Probabilistic operational semantics for a nondeterministic extension of pure lambda-calculus is studied. In this semantics, a term evaluates to a (finite or infinite) distribution of values. Small-step and big-step semantics, inductively and coinductively defined, are given. Moreover, small-step and big-step semantics are shown to produce identical outcomes, both in call-by-value and in call-by-name. Plotkin's CPS translation is extended to accommodate the choice operator and shown correct with respect to the operational semantics. Finally, the expressive power of the obtained system is studied: the calculus is shown to be sound and complete with respect to computable probability distributions.

[prodtoll109] Dubois C. Tollitte P.-N., Delahaye D. Producing certified functional code from inductive specifications. Technical report.

Abstract: Proof assistants based on type theory allow the user to adopt either a functional style, or a relational style (e.g., by using inductive types). Both styles have pros and cons. Relational style may be preferred because it allows the user to describe only what is true, discard momentarily the termination question, and stick to a rule-based description. However, a relational specification is usually not executable. This paper proposes to turn an inductive specification into a functional one, in the logical setting itself, more precisely Coq in this work. We define for a certain class of inductive specifications a way to extract functions from them and automatically produce the proof of soundness of the extracted function w.r.t. its inductive specification. In addition, using user-defined modes which label inputs and outputs, we are able to extract several computational contents from a single inductive type. 2012 Springer-Verlag Berlin Heidelberg.

[progcave89] Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. Technical report.

Abstract: We show how to combine a general purpose type system for an existing language with support for programming with binders and contexts by refining the type system of ML with a restricted form of dependent types where index objects are drawn from contextual LF. This allows the user to specify formal systems within the logical framework LF and index ML types with contextual LF objects. Our language design keeps the index language generic only requiring decidability of equality of the index language providing a modular design. To illustrate the elegance and effectiveness of our language, we give programs for closure conversion and normalization by evaluation.Our three key technical contribution are: 1) We give a bidirectional type system for our core language which is centered around refinement substitutions instead of constraint solving. As a consequence, type checking is decidable and easy to trust, although constraint solving may be undecidable. 2) We give a big-step environment based operational semantics with environments which lends itself to efficient implementation. 3) We prove our language to be type safe and have mechanized our theoretical development in the proof assistant Coq using the fresh approach to binding.

[progjust16] Justin Lubin. Program synthesis with live bidirectional evaluation. Technical report.

Abstract: We present an algorithm for completing program sketches (partial programs, with holes), in which evaluation and example-based synthesis are interleaved until the program is complete and produces a value. Our approach combines and extends recent advances in live programming with holes and type-and-example-directed synthesis of recursive functions over algebraic data types. Novel to our formulation is the ability to simultaneously solve interdependent synthesis goals — the key technique, called live bidirectional evaluation, iteratively solves constraints that arise during 'forward' evaluation of candidate completions and propagates examples 'backward' through partial results. We implement our approach in a prototype system, called Sketch-n-Myth, and develop several examples that demonstrate how live bidirectional evaluation enables a novel workflow for programming with synthesis. On benchmarks used to evaluate a state-of-the-art example-based synthesis technique, Sketch-n-Myth requires on average 55% of the number of examples (even without sketches) by overcoming the example trace-completeness requirement of previous work. Our techniques thus contribute to ongoing efforts to develop synthesis algorithms that can be deployed in future programming environments.

[progjust74] Justin Lubin. Program sketching with live bidirectional evaluation. Technical report.

[proomyre88] Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ml from higher-order logic. Technical report.

Abstract: The higher-order logic found in proof assistants such as Coq and various HOL systems provides a convenient setting for the development and verification of pure functional programs. However, to efficiently run these programs, they must be converted (or extracted) to functional programs in a programming language such as ML or Haskell. With current techniques, this step, which must be trusted, relates similar looking objects that have very different semantic definitions, such as the set-theoretic model of a logic and the operational semantics of a programming language.In this paper, we show how to increase the trustworthiness of this step with an automated technique. Given a functional program expressed in higher-order logic, our technique provides the corresponding program for a functional language defined with an operational semantics, and it provides a mechanically checked theorem relating the two. This theorem can then be used to transfer verified properties of the logical function to the program.We have implemented our technique in the HOL4 theorem prover, translating functions to a core subset of Standard ML, and have applied it to examples including functional data structures, a parser generator, cryptographic algorithms, and a garbage collector.

[propkoma117] Momigliano A. Komauli F. Property-based testing of the meta-theory of abstract machines: An experience report. Technical report.

Abstract: Contrary to Dijkstra's diktat, testing, and more in general validation has found an increasing niche in formal verification, prior or even in alternative to theorem proving. In particular, property-based testing (PBT) is quite effective in mechanized meta-theory of programming languages, where theorems have shallow but tedious proofs that may go wrong for fairly banal mistakes. In this report, we

abandon the comfort of high-level object languages and address the validation of abstract machines and typed assembly languages. We concentrate on Appel et al.'s list-machine benchmark [ADL12], which we tackle with Check, the simple model-checker on top of the nominal logic programming Prolog. We uncover one major bug in the published version of the paper plus several typos and ambiguities thereof. This is particularly striking, as the paper is accompanied by two full formalizations, in Coq and Twelf. Finally, we carry out some mutation testing on the given model, to asses the trade-off between exhaustive and randomized data generation, using for the latter the PBT library FSCheck for F#. Spoiler alert: aProlog performs better. Copyright 2018 for the individual papers by the papers' authors.

[provhgr121] H Grall. Proving fixed points. Technical report.

Abstract: We propose a method to characterize the fixed points described in Tarski's theorem for complete lattices. The method is deductive: the least and greatest fixed points are 'proved' in some inference system defined from deduction rules. We also apply the method to two other fixed point theorems, a generalization of Tarski's theorem to chain-complete posets and Bourbaki-Witt's theorem. Finally, we compare the method with the traditional iterative method resorting to ordinals and the original impredicative method used by Tarski.

[provrodr71] Leonardo Rodriguez, Miguel Pagano, and Daniel Fridlender. Proving correctness of a compiler using step-indexed logical relations. Technical report.

Abstract: In this paper we prove the correctness of a compiler for a call-by-name language using step-indexed logical relations and biorthogonality. The source language is an extension of the simply typed lambda-calculus with recursion, and the target language is an extension of the Krivine abstract machine. We formalized the proof in the Coq proof assistant.

[préscésa80] César Kunz. Pr servation des preuves et transformation de programmes. Technical report.

Abstract: Le paradigme du code mobile implique la distribution des applications par les producteurs de code environnements h t rog nes dans lesquels elles sont ex cut es. Une pratique tendue de ce paradigme est constitu e par le d veloppement d'applications telles que les applets ou les scripts Web, transfer s travers un r seau non s curis comme Internet des syst mes distants, par exemple un ordinateur, un t l phone mobile ou un PDA (Assistant personnel). Naturellement, cet environnement peux ouvrir la porte au d ploiement de programmes malveillants dans des plateformes distantes. Dans certains cas, la mauvaise conduite du code mobile ne constitue pas un risque grave, par exemple lorsque l'int grit des donn es affect es par l'ex cution n'est pas critique ou lorsque l'architecture d'ex cution impose de fortes contraintes sur les capacit s d'ex cution du code non s curis . Il y a toujours, toutefois, des situations dans lesquelles il est indispensable de v rifier la correction fonctionnelle du code avant son ex cution, par exemple lorsque la confidentialit de donn es critiques comme l'information des cartes de cr dit pourrait tre en danger, ou lorsque l'environnement d'ex cution ne poss de pas un m canisme sp cial pour surveiller la consommation excessive des ressources. George Necula a propos une technique pour apporter de la confiance aux consommateurs sur la correction du code sans faire confiance aux producteurs. Cette technique, Proof Carrying Code (PCC), consiste d ploier le code avec une preuve formelle de sa correction. La correction est une propri t inh rente du code re uu qui ne peut pas tre directement d duite du producteur du code. Naturellement, cela donne un avantage PCC quant-aux m thodes bas es sur la confiance l'autorit d'un tiers. En effet, une signature d'une autorit ne suffit pas fournir une confiance absolue sur l'ex cution du code re u. Depuis les origines du PCC, le type de m canisme utilis pour g n rer des certificats repose sur des analyses statiques qui font partie du compilateur. Par cons quent, en restant automatique, il est intrins quement limit des propri t s tr s simples. L'augmentation de l'ensemble des propri t s considerer est difficile et, dans la plupart des cas, cela

exige l'interaction humaine. Une possibilit consiste v rifier directement le code ex cutable. Toutefois, l'absence de structure rend la v rification du code de bas niveau moins naturelle, et donc plus laborieuse. Ceci, combin avec le fait que la plupart des outils de v rification ciblent le code de haut niveau, rend plus appropri e l'id e de transferer la production de certificats au niveau du code source. Le principal inconv nient de produire des certificats pour assurer l'exactitude du code source est que les preuves ne comportent pas la correction du code compil . Plusieurs techniques peuvent etre propos es pour transf rer la preuve de correction d'un programme sa version ex cutable. Cela implique, par exemple, de d ployer le programme source et ses certificats originaux (en plus du code ex cutable) et de certifier la correction du processus de compilation. Toutefois, cette approche n'est pas satisfaisante, car en plus d'exiger la disponibilit du code source, la longueur du certificat garantissant la correction du compilation peut tre prohibitive. Une alternative plus viable consiste proposer un m canisme permettant de g n rer des certificats de code compil partir des certificats du programme source. Les compilateurs sont des proc dures complexes compos es de plusieurs tapes, parmi lesquelles le programme original est progressivement transform en repr sentations interm diaires. Barthe et al. et Pavlova ont montr que les certificats originaux sont conserv s, quelques diff rences pr s non significatives, par la premi re phase de compilation: la compilation non optimale du code source vers une repr sentation non structur e de niveau interm diaire. Toutefois, les optimisations des compilateurs sur les repr sentations interm diaires repr sentent un d fi, car a priori les certificats ne peuvent pas tre r utilis s. Dans cette th se, nous analysons comment les optimisations affectent la validit des certificats et nous proposons un m canisme, Certificate Translation, qui rend possible la g n ration des certificats pour le code mobile ex cutable partir des certificats au niveau du code source. Cela implique transformer les certificats pour surmonter les effets des optimisations de programme.

[reasanco101] Davide Ancona, Francesco Dagnino, and Elena Zucca. Reasoning on divergent computations with coaxioms. Technical report.

Abstract: Coaxioms have been recently introduced to enhance the expressive power of inference systems, by supporting interpretations which are neither purely inductive, nor coinductive. This paper proposes a novel approach based on coaxioms to capture divergence in semantic definitions by allowing inductive and coinductive semantic rules to be merged together for defining a unique semantic judgment. In particular, coinduction is used to derive a special result which models divergence. In this way, divergent, terminating, and stuck computations can be properly distinguished even in semantic definitions where this is typically difficult, as in big-step style. We show how the proposed approach can be applied to several languages; in particular, we first illustrate it on the paradigmatic example of the lambda-calculus, then show how it can be adopted for defining the big-step semantics of a simple imperative Java-like language. We provide proof techniques to show classical results, including equivalence with small-step semantics, and type soundness for typed versions of both languages.

[reguanco82] Davide Ancona. Regular corecursion in prolog. Technical report.

Abstract: Corecursion is the ability of defining a function that produces some infinite data in terms of the function and the data itself, as supported by lazy evaluation. However, in languages such as Haskell strict operations fail to terminate even on infinite regular data, that is, cyclic data.Regular corecursion is naturally supported by coinductive Prolog, an extension where predicates can be interpreted either inductively or coinductively, that has proved to be useful for formal verification, static analysis and symbolic evaluation of programs.In this paper we use the meta-programming facilities offered by Prolog to propose extensions to coinductive Prolog aiming to make regular corecursion more expressive and easier to program with.First, we propose a new interpreter to solve the problem of non-terminating failure as experienced with the standard semantics of coinduction (as supported,

for instance, in SWI-Prolog). Another problem with the standard semantics is that predicates expressed in terms of existential quantification over a regular term cannot directly defined by coinduction; to this aim, we introduce finally clauses, to allow more flexibility in coinductive definitions.Then we investigate the possibility of annotating arguments of coinductive predicates, to restrict coinductive definitions to a subset of the arguments; this allows more efficient definitions, and further enhance the expressive power of coinductive Prolog.We investigate the effectiveness of such features by showing different example programs manipulating several kinds of cyclic values, ranging from automata and context free grammars to graphs and repeating decimals; the examples show how computations on cyclic values can be expressed with concise and relatively simple programs.The semantics defined by these vanilla meta-interpreters are an interesting starting point for a more mature design and implementation of coinductive Prolog. (C) 2013 Elsevier Ltd. All rights reserved.

[resunaka64] Uustalu T. Nakata K. Resumptions, weak bisimilarity and big-step semantics for while with interactive i/o: An exercise in mixed induction-coinduction. Technical report.

Abstract: We look at the operational semantics of languages with interactive I/O through the glasses of constructive type theory. Following on from our earlier work on coinductive trace-based semantics for While [17], we define several big-step semantics for While with interactive I/O, based on resumptions and termination-sensitive weak bisimilarity. These require nesting inductive definitions in coinductive definitions, which is interesting both mathematically and from the point-of-view of implementation in a proof assistant. After first defining a basic semantics of statements in terms of resumptions with explicit internal actions (delays), we introduce a semantics in terms of delay-free resumptions that essentially removes finite sequences of delays on the fly from those resumptions that are responsive. Finally, we also look at a semantics in terms of delay-free resumptions supplemented with a silent divergence option. This semantics hinges on decisions between convergence and divergence and is only equivalent to the basic one classically. We have fully formalized our development in Coq.

[rmlanco59] Davide Ancona, Luca Franceschini, Angelo Ferrando, and Viviana Mascardi. Rml: Theory and practice of a domain specific language for runtime verification. Technical report.

Abstract: Runtime verification (RV) is an approach to verification consisting in dynamically checking that the event traces generated by single runs of a system under scrutiny (SUS) are compliant with the formal specification of its expected correct behavior.RML (Runtime Monitoring Language) is a simple but powerful Domain Specific Language (DSL) for RV which is able to express non context-free properties. When designing RML, particular care has been taken to favor abstraction and simplicity, to better support reusability and portability of specifications and interoperability of the monitors generated from them; this is mainly achieved by decoupling the two problems of property specification and event generation, and by minimizing the available primitive constructs.The formalization and implementation of RML is based on a trace calculus with a fully deterministic rewriting semantics. The semantics of RML is defined by translation into such a calculus, which, in fact, is used as intermediate representation (IR) by the RML compiler. The effectiveness of RML and its methodological impact on RV are presented through interesting patterns that can be adapted to different contexts requiring verification of standard properties. A collection of tested examples is provided, together with benchmarks showing that the deterministic semantics and the performed dynamic optimizations based on the laws of the trace calculus significantly improve the performances of the generated monitors. (C) 2021 Elsevier B.V. All rights reserved.

[scopcasp25] Casper Bach Poulsen. Scopes describe frames: A uniform model for memory layout in dynamic semantics (artifact). Technical report.

Abstract: Our paper introduces a systematic approach to the alignment of names in the static structure of a program, and memory layout and access during its execution. We develop a uniform memory model consisting of frames that instantiate the scopes in the scope graph of a program. This provides a language-independent correspondence between static scopes and run-time memory layout, and between static resolution paths and run-time memory access paths. The approach scales to a range of binding features, supports straightforward type soundness proofs, and provides the basis for a language-independent specification of sound reachability-based garbage collection. This Coq artifact showcases how our uniform model for memory layout in dynamic semantics provides structure to type soundness proofs. The artifact contains type soundness proofs mechanized in Coq for (supersets of) all languages in the paper. The type soundness proofs rely on a language-independent framework formalizing scope graphs and frame heaps.

[scoppoul25] Tolmach A. Visser E. Poulsen C.B., Néron P. Scopes describe frames: A uniform model for memory layout in dynamic semantics. Technical report.

Abstract: Semantic specifications do not make a systematic connection between the names and scopes in the static structure of a program and memory layout, and access during its execution. In this paper we introduce a systematic approach to the alignment of names in static semantics and memory in dynamic semantics, building on the scope graph framework for name resolution. We develop a uniform memory model consisting of frames that instantiate the scopes in the scope graph of a program. This provides a language-independent correspondence between static scopes and run-time memory layout, and between static resolution paths and run-time memory access paths. The approach scales to a range of binding features, supports straightforward type soundness proofs, and provides the basis for a language-independent specification of sound reachabilitybased garbage collection. Casper Bach Poulsen, Pierre N ron, Andrew Tolmach, and Eelco Visser; licensed under Creative Commons License CC-BY.

[semaanco9] Davide Ancona and Andrea Corradi. Semantic subtyping for imperative object-oriented languages. Technical report.

Abstract: Semantic subtyping is an approach for defining sound and complete procedures to decide subtyping for expressive types, including union and intersection types; although it has been exploited especially in functional languages for XML based programming, recently it has been partially investigated in the context of object-oriented languages, and a sound and complete subtyping algorithm has been proposed for record types, but restricted to immutable fields, with union and recursive types interpreted coinductively to support cyclic objects.In this work we address the problem of studying semantic subtyping for imperative object-oriented languages, where fields can be mutable; in particular, we add read/write field annotations to record types, and, besides union, we consider intersection types as well, while maintaining coinductive interpretation of recursive types. In this way, we get a richer notion of type with a flexible subtyping relation, able to express a variety of type invariants useful for enforcing static guarantees for mutable objects.The addition of these features radically changes the definition of subtyping, and, hence, the corresponding decision procedure, and surprisingly invalidates some subtyping laws that hold in the functional setting.We propose an intuitive model where mutable record values contain type information to specify the values that can be correctly stored in fields. Such a model, and the corresponding subtyping rules, require particular care to avoid circularity between coinductive judgments and their negations which, by duality, have to be interpreted inductively.A sound and complete subtyping algorithm is provided, together with a prototype implementation.

[semassc51] S Schneider. Semantics of an intermediate language for program transformation. Technical report.

Abstract: We present an idealized intermediate language designed to investigate the translation between a functional intermediate representation and an imperative register transfer language as it occurs in the back-end of a compiler. A key feature of our language is its dual semantics: there is a functional and an imperative interpretation. The functional interpretation is equipped with a fully compositional notion of program equivalence that is useful for the integration of advanced optimizations. The imperative interpretation is close to assembly and can serve as a faithful model of a low-level (virtual) machine. Programs on which both interpretations coincide are identified via a novel condition we call coherence. Translating between the two interpretations reduces to establishing coherence. Establishing coherence under preservation of the imperative semantics can be seen as a form of SSA construction. To establish coherence under preservation of the functional semantics it suffices to -rename. An -renaming that establishes coherence can be understood as a register assignment. From coherence, decidable correctness conditions for the translations between the two interpretations are derived. The language together with its theory is implemented using the Coq proof assistant without axioms. Translations between the two interpretations are implemented as extractable, translation-validated transformations realizing SSA construction and register assignment.

[sepaaa96] A. Appel. Separation logic for small-step cminor (extended version). Technical report.

Abstract: Cminor is a mid-level imperative programming language (just below C), and there exist proved-correct optimizing compilers from C to Cminor and from Cminor to machine language. We have redesigned Cminor so that it is suitable for Hoare Logic reasoning, we have designed a Separation Logic for Cminor, we have given a small-step operational semantics so that extensions to concurrent Cminor will be possible, and we have a machine-checked proof of soundness of our Separation Logic. This is the first large-scale machine-checked proof of a Separation Logic w.r.t. a small-step semantics, or for a language with nontrivial reducible control-flow constructs. Our sequential soundness proof of the sequential Separation Logic for the sequential language features will be reusable change within a soundness proof of Concurrent Separation Logic w.r.t. Concurrent Cminor. In addition, we have a machine-checked proof of the relation between our small-step semantics and Leroy's original big-step semantics; thus sequential programs can be compiled by Leroy's compiler with formal end-to-end correctness guarantees.

[shalnaka43] Nakano K. Shall we juggle, coinductively? Technical report.

Abstract: Buhler et al. presented a mathematical theory of toss juggling by regarding a toss pattern as an arithmetic function, where the function must satisfy a condition for the pattern to be valid. In this paper, the theory is formalized in terms of coinduction, reflecting the fact that the validity of toss juggling is related to a property of infinite phenomena. A tactic is implemented for proving the validity of toss patterns in Coq. Additionally, the completeness and soundness of a well-known algorithm for checking the validity is demonstrated. The result exposes a practical aspect of coinductive proofs. 2012 Springer-Verlag Berlin Heidelberg.

[sounanco17] Davide Ancona and Andrea Corradi. Sound and complete subtyping between coinductive types for object-oriented languages. Technical report.

Abstract: Structural subtyping is an important notion for effective static type analysis; it can be defined either axiomatically by a collection of subtyping rules, or by means of set inclusion between type interpretations, following the more intuitive approach of semantic subtyping, which allows simpler proofs of the expected properties of the subtyping relation.In object-oriented programming, recursive types are typically interpreted inductively; however, cyclic objects can be represented more precisely by coinductive types.We study semantic subtyping between coinductive types with records and unions, which are particularly interesting for object-oriented programming, and develop and implement a sound and complete top-down direct and effective algorithm for deciding it. To our knowledge, this is the first proposal for a sound and complete top-down direct algorithm for semantic subtyping between coinductive types.

[sounanco2] Davide Ancona. Soundness of object-oriented languages with coinductive big-step semantics. Technical report.

Abstract: It is well known that big-step operational semantics are not suitable for proving soundness of type systems, because of their inability to distinguish stuck from non-terminating computations. We show how this problem can be solved by interpreting coinductively the rules for the standard big-step operational semantics of a Java-like language, thus making the claim of soundness more intuitive: whenever a program is well-typed, its coinductive operational semantics returns a value.Indeed, coinduction allows non-terminating computations to return values; this is proved by showing that the set of proof trees defining the semantic judgment forms a complete metric space when equipped with a proper distance function.In this way, we are able to prove soundness of a nominal type system w.r.t. the coinductive semantics. Since the coinductive semantics is sound w.r.t. the usual small-step operational semantics, the standard claim of soundness can be easily deduced.

[sounanco92] Dagnino F. Zucca E. Ancona D., Barbieri P. Sound regular corecursion in cofj. Technical report.

Abstract: The aim of the paper is to provide solid foundations for a programming paradigm natively supporting the creation and manipulation of cyclic data structures. To this end, we describe coFJ, a Java-like calculus where objects can be infinite and methods are equipped with a codefinition (an alternative body). We provide an abstract semantics of the calculus based on the framework of inference systems with corules. In coFJ with this semantics, FJ recursive methods on finite objects can be extended to infinite objects as well, and behave as desired by the programmer, by specifying a codefinition. We also describe an operational semantics which can be directly implemented in a programming language, and prove the soundness of such semantics with respect to the abstract one. Davide Ancona, Pietro Barbieri, Francesco Dagnino, and Elena Zucca.

[soundagn94] Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. Soundness conditions for big-step semantics. Technical report.

Abstract: We propose a general proof technique to show that a predicate is sound, that is, prevents stuck computation, with respect to a big-step semantics. This result may look surprising, since in big-step semantics there is no difference between non-terminating and stuck computations, hence soundness cannot even be expressed. The key idea is to define constructions yielding an extended version of a given arbitrary big-step semantics, where the difference is made explicit. The extended semantics are exploited in the meta-theory, notably they are necessary to show that the proof technique works. However, they remain transparent when using the proof technique, since it consists in checking three conditions on the original rules only, as we illustrate by several examples.

[squealti125] Devismes S. Altisen K., Corbineau P. Squeezing streams and composition of self-stabilizing algorithms. Technical report.

Abstract: Composition is a fundamental tool when dealing with complex systems. We study the hierarchical collateral composition which is used to combine self-stabilizing distributed algorithms. The PADEC library is a framework developed with the Coq proof assistant and dedicated to the certification of self-stabilizing algorithms. We enrich PADEC with the composition operator and a sufficient condition to show its correctness. The formal proof of the condition leads us to develop new tools and methods on potentially infinite streams, these latter ones being used to model the algorithms executions. The cornerstone has been the definition of the function which removes duplicates from streams. 2019, IFIP International Federation for Information Processing.

[strucolv59] Hayes I.J. Colvin R.J. Structural operational semantics through context-dependent behaviour. Technical report.

Abstract: We present a new approach to providing a structural operational semantics for imperative programming languages with concurrency and procedures. The approach is novel because we expose the building block operations - variable assignment and condition checking - in the labels on the transitions; these form the context-dependent behaviour of a program. Using this style results in two main advantages over standard formalisms for imperative programming language semantics: firstly, our individual transition rules are more concise, and secondly, we are able to more abstractly and intuitively describe the semantics of procedures, including by-value and by-reference parameters. Standard techniques in the literature tend to result in complex and hard-to-read rules for even simple language constructs when procedures and parameters are dealt with. Our semantics for procedures utilises the context-dependent behaviour in the transition label to handle variable name scoping, and defines the semantics of recursion without requiring additional rules. In contrast with Plotkin's seminal structural operational semantics paper, we do not use locations to describe some of the more complex language constructs. Novel aspects of the abstract syntax include local states (in contrast to a single global store), which simplifies the reasoning about local variables, and a command for dynamically renaming variables (in contrast to mapping variables to locations), which simplifies the reasoning about the effect of procedures on by-reference parameters. 2011 Elsevier Inc. All rights reserved.

[strurv70] R. V. Glabbeek. Structural operational semantics. Technical report.

[subtdani93] Altenkirch T. Danielsson N.A. Subtyping, declaratively: An exercise in mixed induction and coinduction. Technical report.

Abstract: It is natural to present subtyping for recursive types coinductively. However, Gapeyev, Levin and Pierce have noted that there is a problem with coinductive definitions of non-trivial transitive inference systems: they cannot be 'declarative'-as opposed to 'algorithmic' or syntax-directed-because coinductive inference systems with an explicit rule of transitivity are trivial. We propose a solution to this problem. By using mixed induction and coinduction we define an inference system for subtyping which combines the advantages of coinduction with the convenience of an explicit rule of transitivity. The definition uses coinduction for the structural rules, and induction for the rule of transitivity. We also discuss under what conditions this technique can be used when defining other inference systems. The developments presented in the paper have been mechanised using Agda, a dependently typed programming language and proof assistant. 2010 Springer-Verlag Berlin Heidelberg.

[syntach86] A Chlipala. Syntactic proofs of compositional compiler correctness. Technical report.

Abstract: Semantic preservation by compilers for higher-order languages can be veried using simple syntactic methods. At the heart of classic techniques are relations between source-level and target-level values. Un- fortunately, these relations are specic to particular compilers, leading to correctness theorems that have nothing to say about linking programs with functions compiled by other compilers or written by hand in the tar- get language. Theorems based on logical relations manage to avoid this problem, but at a cost: standard logical relations do not apply directly to programs with non-termination or impurity, and extensions to han- dle those features are relatively complicated, compared to the classical compiler verication literature. In this paper, we present a new approach to \open' compiler correctness theorems that is \syntactic' in the sense that the core relations do not refer to semantics. Though the technique is much more elementary than previous proposals, it scales up nicely to realistic languages. In particular, untyped and impure programs may be handled simply, while previous work has addressed neither in this context. Our approach is based on the observation that it is an unnecessary hand- icap to consider proofs as black boxes. We identify some theorem-specic proof skeletons, such that we can dene an algebra of nondeterministic compilations and their proofs, and we can compose any two compila- tions to produce a correct-by-construction result. We have prototyped these ideas with a Coq implementation of multiple CPS translations for an untyped Mini-ML source language with recursive functions, sums, products, mutable references, and exceptions.

[synthard108] Pohjola J.Å. Sproul M. Hardin D., Slind K. Synthesis of verified architectural components for critical systems hosted on a verified microkernel. Technical report.

Abstract: We describe a method and tools for the creation of formally verified components that run on the verified seL4 microkernel. This synthesis and verification environment provides a basis to create safe and secure critical systems. The mathematically proved space and time separation properties of seL4 are particularly well-suited for the miniaturised electronics of smaller, lower-cost Unmanned Aerial Vehicles (UAVs), as multiple, independent UAV applications can be hosted on a single CPU with high assurance. We illustrate our method and tools with an example that implements security-improving transformations on system architectures captured in the Architecture Analysis and Design Language (AADL). We show how input validation filter components can be synthesised from regular expressions, and verified to meet arithmetic constraints extracted from the AADL model. Such filters comprise efficient guards on messages to/from the autonomous system. The correctness proofs for filters are automatically lifted to proofs of the corresponding properties on the lazy streams that model the communications of the generated seL4 threads. Finally, we guarantee that the intent of the autonomy application logic is accurately reflected in the application binary code hosted on seL4 through the use of the verified CakeML compiler. 2020 IEEE Computer Society. All rights reserved.

[sémamaa103] MA Asperti. S mantiques formelles. Technical report.

Abstract: Ce m moire pr sente plusieurs d finitions de s mantiques formelles et de transformations de programmes, et expose les choix de conception associ s. En particulier, ce m moire d crit une transformation de programmes inspir e de l' valuation partielle et d di e la compr hension de programmes scientifiques crits en Fortran. Il d taille galement le front-end d'un compilateur r aliste du langage C, ayant t formellement v rifi en C.

[toolmjp84] MJ Parreira Pereira. Tools and techniques for the verification of modular stateful code. Technical report.

Abstract: This thesis is set in the field of formal methods, more precisely in the domain of deductive program verification. Our working context is the Why3 framework, a set of tools to implement, formally specify, and prove programs usingoff-the-shelf theorem provers. Why3 features a programming language,called WhyML, designed with verification in mind. An important feature of WhyML is ghost code: portions of the program that are introduced for the sole purpose of specification andverification. When it comes to get an executable implementation, ghost code is removed by an automatic process called extraction. One of the main contributions of this thesis is the formalization and implementation of Why3's extraction. The formalization consists in showing that the extracted program preserves the same operational behavior as the original source code, based on a type and effect system. The new extraction mechanism has been successfully used to get correct-by-construction OCaml modules, which are part of averified OCaml library of data structures and algorithms. This verification effort led to two other contributions of this thesis.The first is a systematic approach to the verification ofpointer-based data structures using ghost models of fragments of the heap. A fully automatic verification of a union-find data structure was achieved using this technique. The second contribution is a modular way to reason about iteration, independently of the underlying implementation. Several cursors and higher-orderiterators have been specified and verified with this approach.

[towaalbe77] Alberto Ciaffaglione. Towards turing computability via coinduction. Technical report.

Abstract: We adopt corecursion and coinduction to formalize Turing Machines and their operational semantics in the Coq proof assistant. By combining the formal analysis of converging and diverging computations, via big-step and small-step predicates, our approach allows us to certify the correctness of concrete Turing Machines. An immediate application of our methodology is the proof of the undecidability of the halting problem, therefore our effort may be seen as a first step towards the formal development of basic computability theory.

[towarosu126] Grigore Rosu and Andrei Stefanescu. Towards a unified theory of operational and axiomatic semantics. Technical report.

Abstract: This paper presents a nine-rule language-independent proof system that takes an operational semantics as axioms and derives program reachability properties, including ones corresponding to Hoare triples. This eliminates the need for language-specific Hoare-style proof rules to verify programs, and, implicitly, the tedious step of proving such proof rules sound for each language separately. The key proof rule is Circularity, which is coinductive in nature and allows for reasoning about constructs with repetitive behaviors (e. g., loops). The generic proof system is shown sound and has been implemented in the MatchC verifier.

[tracnaka88] Uustalu T. Nakata K. Trace-based coinductive operational semantics for while. Technical report.

Abstract: We present four coinductive operational semantics for the While language accounting for both terminating and non-terminating program runs: big-step and small-step relational semantics and big-step and small-step functional semantics. The semantics employ traces (possibly infinite sequences of states) to record the states that program runs go through. The relational semantics relate statement-state pairs to traces, whereas the functional semantics return traces for statement-state pairs. All four semantics are equivalent. We formalize the semantics and their equivalence proofs in the constructive setting of Coq. 2009 Springer.

[tranbura52] Burak Emir. Translation correctness for first-order object-oriented pattern matching. Technical report.

Abstract: Pattern matching makes ML programs more concise and readable, and these qualities are also sought in object-oriented settings. However, objects and classes come with open class hierarchies, extensibility requirements and the need for data abstraction, which all conflict with matching on concrete data types. Extractor-based pattern matching has been proposed to address this conflict. Extractors are user-defined methods that perform the task of value discrimination and deconstruction during pattern matching. In this paper, we give the first formalization of extractor-based matching, using a first-order object-oriented calculus. We give a direct operational semantics and prove it sound. We then present an optimizing translation to a target language without matching, and prove a correctness result stating that an expression is equivalent to its translation.

[trannest11] Nestra H. Transfinite semantics in the form of greatest fixpoint. Technical report.

Abstract: Transfinite semantics is a semantics according to which program executions can continue working after an infinite number of steps. Such a view of programs can be useful in the theory of program transformations. So far, transfinite semantics have been succesfully defined for iterative loops. This paper provides an exhaustive definition for semantics that enable also infinitely deep recursion. The definition is actually a parametric schema that defines a family of different transfinite semantics. As standard semantics also match the same schema, our framework describes both standard and transfinite semantics in a uniform way. All semantics are expressed as greatest fixpoints of monotone operators on some complete lattices.

It turns out that, for transfinite semantics, the corresponding lattice operators are cocontinuous. According to Kleene's theorem, this shows that transfinite semantics can be expressed as a limit of iteration which is not transfinite. 2009 Elsevier Inc. All rights reserved.

[twofréd1] Frédéric Gava. Two formal semantics of a subset of the paderborn university bsplib. Technical report.

Abstract: PUB (Paderborn University BSPLib) is a C library supporting the development of Bulk-Synchronous Parallel (BSP) algorithms. The BSP model allows an estimation of the execution time, avoids deadlocks and non-determinism. This paper presents two formal operational semantics for a C+PUB subset language using the Coq proof assistant, one for classical BSP operations and one that emphasises high performance primitives.

[typeamin85] Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. Technical report.

Abstract: While type soundness proofs are taught in every graduate PL class, the gap between realistic languages and what is accessible to formal proofs is large. In the case of Scala, it has been shown that its formal model, the Dependent Object Types (DOT) calculus, cannot simultaneously support key metatheoretic properties such as environment narrowing and subtyping transitivity, which are usually required for a type soundness proof. Moreover, Scala and many other realistic languages lack a general substitution property.The first contribution of this paper is to demonstrate how type soundness proofs for advanced, polymorphic, type systems can be carried out with an operational semantics based on high-level, definitional interpreters, implemented in Coq. We present the first mechanized soundness proofs in this style for System F-<: and several extensions, including mutable references. Our proofs use only straightforward induction, which is significant, as the combination of big-step semantics, mutable references, and polymorphism is commonly believed to require coinductive proof techniques.The second main contribution of this paper is to show how DOT-like calculi emerge from straightforward generalizations of the operational aspects of F-< :, exposing a rich design space of calculi with path-dependent types inbetween System F and DOT, which we dub the System D Square.By working directly on the target language, definitional interpreters can focus the design space and expose the invariants that actually matter at runtime. Looking at such runtime invariants is an exciting new avenue for type system design.

[typejan104] Jan Hoffmann. Type-based amortized resource analysis with integers and arrays. Technical report.

Abstract: Proving bounds on the resource consumption of a program by statically analyzing its source code is an important and well-studied problem. Automatic approaches for numeric programs with side effects usually apply abstract interpretation based invariant generation to derive bounds on loops and recursion depths of function calls. This paper presents an alternative approach to resource-bound analysis for numeric, heap-manipulating programs that uses type-based amortized resource analysis. As a first step towards the analysis of imperative code, the technique is developed for a first-order ML-like language with unsigned integers and arrays. The analysis automatically derives bounds that are multivariate polynomials in the numbers and the lengths of the arrays in the input. Experiments with example programs demonstrate two main advantages of amortized analysis over current abstract interpretation based techniques. For one thing, amortized analysis can handle programs with non-linear intermediate values like f((n + m)2). For another thing, amortized analysis is compositional and works naturally for compound programs like f(g(x)).

[typejan116] Jan Hoffmann. Type-based amortized resource analysis with integers and arrays*. Technical report.

[varipatr103] Patrick COUSOT. Varieties of static analyzers: A comparison with astree. Technical report.

Abstract: We discuss the characteristic properties of ASTREE, an automatic static analyzer for proving the absence of runtime errors in safety-critical real-time synchronous control command C programs, and compare it with a variety of other program analysis tools.

[verijalo80] Maalej M. Moy Y. Paskevich A. Jaloyan G.-A., Dross C. Verification of programs with pointers in spark. Technical report.

Abstract: In the field of deductive software verification, programs with pointers present a major challenge due to pointer aliasing. In this paper, we introduce pointers to SPARK, a well-defined subset of the Ada language, intended for formal verification of mission-critical software. Our solution uses a permission-based static alias analysis method inspired by Rust s borrow-checker and affine types. To validate our approach, we have implemented it in the SPARK GNATprove formal verification toolset for Ada. In the paper, we give a formal presentation of the analysis rules for a core version of SPARK and discuss their implementation and scope. 2020, Springer Nature Switzerland AG.

[wellrust57] Rustan Leino. Well-founded functions and extreme predicates in dafny: A tutorial. Technical report.

[whymomi16] Momigliano A. Why proof-theory matters in specification-based testing. Technical report.

Abstract: We survey some recent developments in giving a logical reconstruction of specification-based testing via the lenses of structural proof-theory. 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0). Partially supported by GNCS project METALLIC #2: METodi di prova per il ragionamento Automatico per Logiche non-cLassICche .