JFP **26**, e24, 18 pages, 2016.
doi:10.1017/S0956796816000290

PhD Abstracts

GRAHAM HUTTON
University of Nottingham, UK
(*e-mail:* graham.hutton@nottingham.ac.uk)

Many students complete PhDs in functional programming each year. As a service to the community, the Journal of Functional Programming publishes the abstracts from PhD dissertations completed during the previous year.

The abstracts are made freely available on the JFP website, i.e. not behind any paywall. They do not require any transfer of copyright, merely a license from the author. A dissertation is eligible for inclusion if parts of it have or could have appeared in JFP, that is, if it is in the general area of functional programming. The abstracts are not reviewed.

We are delighted to publish 12 abstracts from 2015/16 in this round and hope that JFP readers will find many interesting dissertations in this collection that they may not otherwise have seen. If a student or advisor would like to submit a dissertation abstract for publication in this series, please contact the series editor for further details.

Graham Hutton
PhD Abstract Editor

Profiling a Parallel Domain Specific Language using Off-The-Shelf Tools

MAJED MOHAMMED ABDULLAH AL-SAEED
University of Glasgow, UK

Date: December 2015; Advisor: Phil Trinder
URL: http://tinyurl.com/h96wr5r

Profiling tools are essential for understanding and tuning the performance of both parallel programs and parallel language implementations. Assessing the performance of a program in a language with high-level parallel coordination is often complicated by the layers of abstraction present in the language and its implementation. This thesis investigates whether it is possible to profile parallel Domain Specific Languages (DSLs) using existing host language profiling tools. The key challenge is that the host language tools report the performance of the DSL, runtime system (RTS) executing the application rather than the performance of the DSL application. The key questions are whether a correct, effective and efficient profiler can be constructed using host language profiling tools; is it possible to effectively profile the DSL implementation, and what capabilities are required of the host language profiling tools?

The main contribution of this thesis is the development of an execution profiler for the parallel DSL, Haskell Distributed Parallel Haskell (HdpH) using the host language profiling tools. We show that it is possible to construct a profiler (HdpHProf) to support performance analysis of both the DSL applications and the DSL implementation. The implementation uses several new GHC features, including the GHC-Events Library and ThreadScope, develops two new performance analysis tools for DSL, HdpH internals, i.e. Spark Pool Contention Analysis and Registry Contention Analysis.

We present a critical comparative evaluation of the host language profiling tools that we used (GHC-PPS and ThreadScope) with another recent functional profilers, EdenTV, alongside four important imperative profilers. This is the first report on the performance of functional profilers in comparison with well established industrial standard imperative profiling technologies. We systematically compare the profilers for usability and data presentation. We found that the GHC-PPS performs well in terms of overheads and usability so using it to profile the DSL is feasible and would not have significant impact on the DSL performance.

We validate HdpHProf for functional correctness and measure its performance using six benchmarks. HdpHProf works correctly and can scale to profile HdpH programs running on up to 192 cores of a 32 nodes Beowulf cluster. We characterise the performance of HdpHProf in terms of profiling data size and profiling execution runtime overhead. It shows that HdpHProf does not alter the behaviour of the GHC-PPS and retains low tracing overheads close to the studied functional profilers; 18%

on average. Also, it shows a low ratio of HdpH trace events in GHC-PPS eventlog, less than 3% on average.

We show that HdpHProf is effective and efficient to use for performance analysis and tuning of the DSL applications. We use HdpHProf to identify performance issues and to tune the thread granularity of six HdpH benchmarks with different parallel paradigms, e.g. divide and conquer, flat data parallel, and nested data parallel. This include identifying problems such as, too small/large thread granularity, problem size too small for the parallel architecture, and synchronisation bottlenecks.

We show that HdpHProf is effective and efficient for tuning the parallel DSL implementation. We use the Spark Pool Contention Analysis tool to examine how the spark pool implementation performs when accessed concurrently. We found that appropriate thread granularity can significantly reduce both conflict ratios and conflict durations, by more than 90%. We use the Registry Contention Analysis tool to evaluate three alternatives of the registry implementations. We found that the tools can give a better understanding of how different implementations of the HdpH RTS perform.

https://doi.org/10.1017/S0956796816000290 Published online by Cambridge University Press

Theorem Provers as Libraries – An Approach to Formally Verifying Functional Programs

EVAN AUSTIN
University of Kansas, USA

Date: May 2015; Advisor: Perry Alexander
URL: `http://tinyurl.com/h6p6mye`

Property-directed verification of functional programs tends to take one of two paths. First, is the traditional testing approach, where properties are expressed in the original programming language and checked with a collection of test data. Alternatively, for those desiring a more rigorous approach, properties can be written and checked with a formal tool; typically, an external proof system. This dissertation details a hybrid approach that captures the best of both worlds: the formality of a proof system paired with the native integration of an embedded, domain specific language (EDSL) for testing. At the heart of this hybridization is the titular concept – a theorem prover as a library. The verification capabilities of this prover, HaskHOL, are introduced to a Haskell development environment as a GHC compiler plugin. Operating at the compiler level provides for a comparatively simpler integration and allows verification to co-exist with the numerous other passes that stand between source code and program.

Certified Semantics and Analysis of JavaScript

MARTIN BODIN
Université de Rennes 1, France

Date: November 2016; Advisor: Alan Schmitt and Thomas Jensen
URL: http://tinyurl.com/hzrdpwn

JavaScript has become a trending programming language. It is now used in various applications, including some in which security is a crucial issue. It thus becomes important to be able to control the quality of softwares written in JavaScript. This thesis explores the approach of formal proofs: We aim to provide mathematical proofs that chosen JavaScript programs behave as expected. To build such proofs, we use proof assistants such as Coq: these programs are carefully written to trustworthily check proofs.

To state that a JavaScript program is behaving as expected, we first need a semantics of the JavaScript language. This thesis has naturally taken part on the JSCert project, whose aim is to provide a formal semantics for JavaScript. Because of the size of JavaScript's semantics, it is crucial to know how it can be trusted: a typing mistake could compromise the whole semantics. To trust JSCert, we based ourselves on two different trust sources. On one hand, JSCert has been designed to be as similar to the official JavaScript specification the ECMAScript standard as it can be. Both specifications use the same data structures. Furthermore, each derivation rule in JSCert can be related to a specification line of ECMAScript and conversely, each specification line of ECMAScript correspond to one derivation rule in JSCert. On the other hand, we defined an interpreter named JSRef. JSCert was proven correct with respect to JSCert. This enables us to run JSRef on JavaScript test suites, thus confronting JSCert to real JavaScript programs. The JSCert semantics is not the first formal semantics of JavaScript, but it is the first to propose two distinct ways to relate the formal semantics to the JavaScript language: by having a semantics close to the official specification, and by testing and comparing it to other interpreters.

Instead of independently proving that each JavaScript program behaves as expected, we chose to analyse programs using abstract interpretation. It consists of interpreting the semantics of a programming language with less precise abstract values instead of precise concrete values. For instance, the concrete value 1 can be replaced by the abstract value + and -1 can be replaced by the abstract value -, tracing the signs of values. Of course, abstract domains are usually much more complex and precise. Abstract interpretation is split into two steps: first, an abstract semantics is built and proven correct with respect to its concrete semantics, then, analysers are built from this abstract semantics. In the case of JavaScript, the first step is already a challenging target: this thesis only focusses on the construction of a certified abstract semantics.

The JSCert semantics is huge – more than 800 derivation rules. Building an abstract semantics using traditional techniques does not scale towards such sizes. We thus designed a new way to build abstract semantics from concrete semantics. Our approach is based on a careful analysis on the structure of JSCert's derivation rules. It aims at minimising the proof effort needed to build an abstract semantics. This approach can be applied on any pretty-big-step style (a variant of big-step style) semantics, given some abstract domains. In particular, we provide a precise definition of pretty-big-step semantic rules, making explicit each of their components. Our semantic rules contains both syntactic and semantic components: syntactic elements comprise the rule name and the terms on which it applies, whilst semantic elements are defined by transfer functions. The abstract semantic is built by only abstracting transfer functions: the syntactic parts of semantic rules are left unchanged. This immensely helps building abstract semantics from concrete semantics. Abstract transfer functions are required to be locally correct with respect to concrete transfer functions. We proved that the correctness of the whole semantics follows from these local properties. Notably, we do not have to prove anything about the interaction between abstract rules. Abstract semantics also comprise non-structural rules – the most common being the weakening rule. This kind of abstract rules are also present in our formalism, in a framework generic enough to prove usual abstract rules admissible. We applied our framework on several small languages to build and prove abstract analysers.

We applied our framework on a domain close to JavaScript's memory. Our abstract domain is based on separation logic. This logic requires several adaptations to apply in the context of abstract interpretation: the frame rule of separation logic does not follow our local correctness constraints as-is. In particular, the weakening rule of abstract interpretation enables to rename abstract location identifiers, which can then unexpectedly interact through the frame rule. To solve this issue, we introduced membraned formulae: these formulae are equipped by a membrane, whose aim is to carry identifier changes along abstract derivations. The frame rule is updated to apply these identifier changes. Our domains, although very simple compared to the memory model of JavaScript, seem expressive enough to enable the proof of already existing analysers.

This thesis presents three main contributions: a trusted formal semantics for the JavaScript language, a generic framework to build abstract semantics, and a non-trivial domain for this formalism. Each of these contributions are associated with a Coq development. We think that this thesis provides solid bases to formally certify real-world JavaScript analysers. This thesis focusses on JavaScript, but we believe that the given formalism can be applied to other semantics.

Lazy Evaluation: From Natural Semantics to a Machine-Checked Compiler Transformation

JOACHIM BREITNER
Karlsruhe Institute of Technology, Germany

Date: April 2016; Advisor: Gregor Snelting
URL: http://tinyurl.com/jyd3srx

In order to solve a long-standing problem with list fusion, a new compiler transformation, 'Call Arity' is developed and implemented in the Haskell compiler GHC. It is formally proven to not degrade program performance; the proof is machine-checked using the interactive theorem prover Isabelle. To that end, a formalization of Launchbury's Natural Semantics for Lazy Evaluation is modelled in Isabelle, including a correctness and adequacy proof.

An Information Theoretic Approach to the Expressiveness of Programming Languages

JOSEPH RAY DAVIDSON
University of Glasgow, UK

Date: June 2016; Advisor: Greg Michaelson and Phil Trinder
URL: http://tinyurl.com/hm62xln

The conciseness conjecture is a long-standing notion in computer science that programming languages with more built-in operators, that is more expressive languages with larger semantics, produce smaller programs on average. Chaitin defines the related concept of an elegant program such that there is no smaller program in some language which, when run, produces the same output.

This thesis investigates the conciseness conjecture in an empirical manner. Influenced by the concept of elegant programs, we investigate several models of computation, and implement a set of functions in each programming model. The programming models are Turing Machines, λ-Calculus, SKI, RASP, RASP2, and RASP3. The information content of the programs and models are measured as characters. They are compared to investigate hypotheses relating to how the mean program size changes as the size of the semantics change, and how the relationship of mean program sizes between two models compares to that between the sizes of their semantics.

We show that the amount of information present in models of the same paradigm, or model family, is a good indication of relative expressivity and average program size. Models that contain more information in their semantics have smaller average programs for the set of tested functions. In contrast, the relative expressiveness of models from differing paradigms, is not indicated by their relative information contents.

RASP and Turing Machines have been implemented as Field Programmable Gate Array (FPGA) circuits to investigate hardware analogues of the hypotheses above. Namely that the amount of information in the semantics for a model directly influences the size of the corresponding circuit, and that the relationship of mean circuit sizes between models is comparable to the relationship of mean program sizes.

We show that the number of components in the circuits that realise the semantics and programs of the models correlates with the information required to implement the semantics and program of a model. However, the number of components to implement a program in a circuit for one model does not relate to the number of components implementing the same program in another model. This is in contrast to the more abstract implementations of the programs.

Information is a computational resource and therefore follows the rules of Blum's axioms. These axioms and the speedup theorem are used to obtain an alternate proof of the undecidability of elegance.

This work is a step towards unifying the formal notion of expressiveness with the notion of algorithmic information theory and exposes a number of interesting research directions. A start has been made on integrating the results of the thesis with the formal framework for the expressiveness of programming languages.

---

https://doi.org/10.1017/S0956796816000290 Published online by Cambridge University Press

Formalisation and Execution of Linear Algebra: Theorems and Algorithms

JOSE DIVASÓN
Universidad de La Rioja, Spain

Date: June 2016; Advisor: Jesús María Aransay Azofra and Julio Jesús Rubio García
URL: http://tinyurl.com/zt2cnfo

This thesis studies the formalisation and execution of Linear Algebra algorithms in Isabelle/HOL, an interactive theorem prover. The work is based on the HOL Multivariate Analysis library, whose matrix representation has been refined to datatypes that admit a representation in functional programming languages. This enables the generation of programs from such verified algorithms. In particular, several well-known Linear Algebra algorithms have been formalised involving both the computation of matrix canonical forms and decompositions (such as the Gauss-Jordan algorithm, echelon form, Hermite normal form, and QR decomposition). The formalisation of these algorithms is also accompanied by the formal proofs of their particular applications such as calculation of the rank of a matrix, solution of systems of linear equations, orthogonal matrices, least squares approximations of systems of linear equations, and computation of determinants of matrices over Bzout domains. Some benchmarks of the generated programs are presented as well where matrices of remarkable dimensions are involved, illustrating the fact that they are usable in real-world cases. The formalisation has also given place to side-products that constitute themselves standalone reusable developments: serialisations to SML and Haskell, an implementation of algebraic structures in Isabelle/HOL, and generalisations of well-established Isabelle/HOL libraries. In addition, an experiment involving Isabelle, its logics, and the formalisation of some underlying mathematical concepts presented in Voevodsky's simplicial model for Homotopy Type Theory is presented.

Dependent Types in Haskell: Theory and Practice

RICHARD A. EISENBERG
University of Pennsylvania, USA

Date: December 2016; Advisor: Stephanie Weirich
URL: http://tinyurl.com/ha39bzn

Haskell, as implemented in the Glasgow Haskell Compiler (GHC), has been adding new type-level programming features for some time. Many of these features—chiefly: generalized algebraic datatypes (GADTs), type families, kind polymorphism, and promoted datatypes—have brought Haskell to the doorstep of dependent types. Many dependently typed programs can even currently be encoded, but often the constructions are painful.

In this dissertation, I describe Dependent Haskell, which supports full dependent types via a backward-compatible extension to today's Haskell. An important contribution of this work is an implementation, in GHC, of a portion of Dependent Haskell, with the rest to follow. The features I have implemented are already released, in GHC 8.0. This dissertation contains several practical examples of Dependent Haskell code, a full description of the differences between Dependent Haskell and today's Haskell, a novel type-safe dependently typed lambda-calculus (called Pico) suitable for use as an intermediate language for compiling Dependent Haskell, and a type inference and elaboration algorithm, Bake, that translates Dependent Haskell to type-correct Pico.

Certified Algorithms for Context-Free Grammars

DENIS FIRSOV
Tallinn University of Technology, Estonia

Date: August 2016; Advisor: Tarmo Uustalu
URL: http://tinyurl.com/hy334vw

Context-free grammars are a widely used formalism in compiler construction for defining the syntactical structure of programming languages. In this thesis, our main goal is to implement and certify a parser generator for context-free languages. A parser generator takes a context-free grammar and returns a function that tries to find a parse tree for a given string. A certified parser generator delivers valid parse trees and will find one, if it exists, for the given context-free grammar.

There are different approaches to show that programs are correct: model checking, axiomatic semantics, expressive type systems. We are interested in constructive branches of logic. Proofs carried out within a constructive logic may be considered as programs in a functional language. It is important because of the possibility of extraction of the purported object from an existence proof. Our work is done in the Agda dependently typed programming language. Agda provides a single framework to write functional code and prove properties about it.

The main constituents of a context-free grammar are finite sets of terminals, nonterminals, and rules. Therefore, in the first part of the thesis, we investigate various encodings and properties of finiteness in constructive mathematics. A set is considered listable, if it can be completely enumerated in a list. We begin by showing that listable sets have decidable equality. This result allows us to conclude that certain basic variations of listability are logically equivalent to each other. We also develop a library of combinators to ease programming with finite sets. The library includes combinators for concise definition of functions on listable sets and a prover for quantified formulas over decidable properties on listable sets. Additionally, we propose that new listable sets can be defined by listing subsets of a base set with decidable equality.

The second part of the thesis is devoted to parsing. We implement and certify the Cocke-Younger-Kasami algorithm, as it has a simple and elegant structure. Moreover, the algorithm allows one to parse general context-free languages. The relative simplicity greatly contributes to certifying the implementation. The naive recursive encoding leads to excessive recomputations, but we recover the efficient algorithm by introducing memoization. The refinement to the memoized version is done in a provably correctness-preserving manner.

The downside of the CYK parsing algorithm is that it requires context-free grammars in Chomsky normal form. The last part of the thesis focuses on normalization of context-free grammars. We divide normalization into four independent transformations. For each transformation, we prove that it achieves progress towards normality and also preserves the language of the grammar. We also show that the

composition of transformations in the appropriate order converts any context-free grammar into its Chomsky normal form. Moreover, the proof of soundness of the normalization procedure is a function for converting any parse tree for the normalized grammar back into a parse tree for the original grammar.

---

https://doi.org/10.1017/S0956796816000290 Published online by Cambridge University Press

The Scalability of Reliable Computation in Erlang

AMIR GHAFFARI
University of Glasgow, UK

Date: October 2015; Advisor: Phil Trinder
URL: http://tinyurl.com/hvtff6a

With the advent of many-core architectures, scalability is a key property for programming languages. Actor-based frameworks like Erlang are fundamentally scalable, but in practice they have some scalability limitations.

The RELEASE project aims to scale the Erlang's radical concurrency-oriented programming paradigm to build reliable general-purpose software, such as server-based systems, on emergent commodity architectures with 104 cores. The RELEASE consortium works to scale Erlang at the virtual machine, language level, infrastructure levels, and to supply profiling and refactoring tools.

This research contributes to the RELEASE project at the language level. First, we study the provision of scalable persistent storage options for Erlang. We articulate the requirements for scalable and available persistent storage, and evaluate four popular Erlang DBMSs against these requirements. We investigate the scalability limits of the Riak NoSQL DBMS using Basho Bench up to 100 nodes on the Kalkyl cluster and establish for the first time scientifically the scalability limit of Riak as 60 nodes, thereby confirming developer folklore.

We design and implement DE-Bench, a scalable fault-tolerant peer-to-peer benchmarking tool that measures the throughput and latency of distributed Erlang commands on a cluster of Erlang nodes. We employ DE-Bench to investigate the scalability limits of distributed Erlang on up to 150 nodes and 1,200 cores. Our results demonstrate that the frequency of global commands limits the scalability of distributed Erlang. We also show that distributed Erlang scales linearly up to 150 nodes and 1,200 cores with relatively heavy data and computation loads when no global commands are used.

As part of the RELEASE project, the Glasgow University team has developed Scalable Distributed Erlang (SD Erlang) to address the scalability limits of distributed Erlang. We evaluate SD Erlang by designing and implementing the first ever demonstrators for SD Erlang, i.e. DE-Bench, Orbit and Ant Colony Optimisation(ACO). We employ DE-Bench to evaluate the performance and scalability of group operations in SD-Erlang up to 100 nodes. Our results show that the alternatives SD-Erlang offers for global commands (i.e. group commands) scale linearly up to 100 nodes. We also develop and evaluate an SD-Erlang implementation of Orbit, a symbolic computing kernel and a generalization of a transitive closure computation. Our evaluation results show that SD Erlang Orbit outperforms the distributed Erlang Orbit on 160 nodes and 1,280 cores. Moreover, we develop a reliable distributed version of ACO and show that the reliability of ACO limits its scalability in traditional distributed Erlang. We use SD-Erlang to improve the

scalability of the reliable ACO by eliminating global commands and avoiding full mesh connectivity between nodes. We show that SD Erlang reduces the network traffic between nodes in an Erlang cluster effectively.

---

https://doi.org/10.1017/S0956796816000290 Published online by Cambridge University Press

Unfolding Semantics of the Untyped λ-Calculus with Letrec

JAN ROCHEL
Utrecht University, The Netherlands

Date: June 2016; Advisor: Doaitse Swierstra, Vincent van Oostrom and Clemens Grabmayer
URL: http://tinyurl.com/jrmdxes

We investigate the relationship between finite terms in λ-letrec, the λ-calculus with letrec, and the infinite λ-terms they express. We say that a λ-letrec term expresses a λ-term if the latter can be obtained as an infinite unfolding of the former. Unfolding is the process of substituting occurrences of function variables by the right-hand side of their definition. We consider the following questions:

- How can we characterise those infinite λ-terms that are λ-letrec-expressible?
- Given two λ-letrec terms, how can we determine whether they have the same unfolding?
- Given a λ-letrec term, can we find a more compact version of the term with the same unfolding?

To tackle these questions we introduce and study the following formalisms:

- a rewriting system for unfolding λ-letrec terms into λ-terms
- a rewriting system for 'observing' λ-terms by dissecting their term structure
- higher-order and first-order graph formalisms together with translations between them as well as translations from and to λ-letrec

We identify a first-order term graph formalism on which bisimulation preserves and reflects the unfolding semantics of λ-letrec and which is closed under functional bisimulation. From this we derive efficient methods to determine whether two terms are equivalent under infinite unfolding and to compute the maximally shared form of a given λ-letrec term.

Parallel Evaluation Strategies for Lazy Data Structures in Haskell

PRABHAT TOTOO
Heriot-Watt University, UK

Date: May 2016; Advisor: Hans-Wolfgang Loidl
URL: http://tinyurl.com/zc8myr3

Conventional parallel programming is complex and error prone. To improve programmer productivity, we need to raise the level of abstraction with a higher-level programming model that hides many parallel coordination aspects. Evaluation strategies use non-strictness to separate the coordination and computation aspects of a Glasgow parallel Haskell (GpH) program. This allows the specification of high level parallel programs, eliminating the low-level complexity of synchronisation and communication associated with parallel programming.

This thesis employs a data-structure-driven approach for parallelism derived through generic parallel traversal and evaluation of sub-components of data structures. We focus on evaluation strategies over list, tree and graph data structures, allowing re-use across applications with minimal changes to the sequential algorithm.

In particular, we develop novel evaluation strategies for tree data structures, using core functional programming techniques for coordination control, achieving more flexible parallelism. We use non-strictness to control parallelism more flexibly. We apply the notion of fuel as a resource that dictates parallelism generation, in particular, the bi-directional flow of fuel, implemented using a circular program definition, in a tree structure as a novel way of controlling parallel evaluation. This is the first use of circular programming in evaluation strategies and is complemented by a lazy function for bounding the size of sub-trees.

We extend these control mechanisms to graph structures and demonstrate performance improvements on several parallel graph traversals. We combine circularity for control for improved performance of strategies with circularity for computation using circular data structures. In particular, we develop a hybrid traversal strategy for graphs, exploiting breadth-first order for exposing parallelism initially, and then proceeding with a depth-first order to minimise overhead associated with a full parallel breadth-first traversal.

The efficiency of the tree strategies is evaluated on a benchmark program, and two non-trivial case studies: a Barnes-Hut algorithm for the n-body problem and sparse matrix multiplication, both using quad-trees. We also evaluate a graph search algorithm implemented using the various traversal strategies.

We demonstrate improved performance on a server-class multicore machine with up to 48 cores, with the advanced fuel splitting mechanisms proving to be more flexible in throttling parallelism. To guide the behaviour of the strategies, we develop heuristics-based parameter selection to select their specific control parameters.

Formalizing Symbolic Decision Procedures for Regular Languages

DMITRIY TRAYTEL
Technische Universität München, Germany

Date: October 2015; Advisor: Tobias Nipkow
URL: http://tinyurl.com/hfan33k

This thesis studies decision procedures for the equivalence of regular languages represented symbolically as regular expressions or logical formulas.

Traditional decision procedures in this context rush to dispose of the concise symbolic representation by translating it into finite automata, which then are efficiently minimized and checked for structural equality.

We develop procedures that avoid this explicit translation by working with the symbolic structures directly. This results in concise functional algorithms that are easy to reason about, even formally. Indeed, the presented decision procedures are specified and proved correct in the proof assistant Isabelle.

The main motivation for working with symbolic representations is simplicity. Regular expressions and formulas are free datatypes equipped with induction and recursion principles and suitable for equational reasoning—the core competence of proof assistants and functional programming languages. In contrast, automata are arbitrary graphs and therefore not as easy to reason about in a structural fashion.

The core idea, shared by all procedures under consideration, is the usage of a symbolic derivative operation that replaces the global transition table of the automaton. For regular expressions those are the increasingly popular Brzozowski derivatives and their cousins. For formulas, the development of such operations is the main theoretical contribution of this thesis.

The main technical contribution is the formalization of a uniform framework for deciding equivalence of regular languages and the instantiation of this framework by various symbolic representations. Overall, this yields formally verified decision procedures for the equivalence of various kinds of regular expressions, Presburger arithmetic formulas, and formulas of monadic second-order logic on finite words under two different existing semantics (WS1S and M2L(Str)). Using Isabelle's code generator, we extract certified algorithms from the formalization in conventional functional programming languages.