## Bab 2

### Metode Formal [Hinchey1995](#fn-hinchey1995)

* Apa itu metode formal (Definisi, Penemu, Filosofi) [Monin2003](#fn-monin2003)

Baier  
ICT systems (Information and Communication Technology)

formal methods can be considered as “the applied mathematics for modeling and analyzing ICT systems”.

Holloway

formal methods are defined as the mathematics of computer software and hardware systems

Menurut Monin, sebenarnya istilah Teknik Formal lebih pantas untuk digunakan dibandingkan Metode Formal karena teknik ini belum memiliki metodologi yang baku. Namun karena istilah Metode Formal lebih populer maka untuk selanjutnya dalam tulisan ini istilah Metode Formal lah yang akn digunakan.

* Kenapa metode formal [Holloway1997](#fn-holloway1997)

Baier

Besides, formal methods are one of the “highly recommended” verification techniques for software development of safetycritical systems according to, e.g., the best practices standard of the IEC (International Electrotechnical Commission) and standards of the ESA (European Space Agency). The resulting report of an investigation by the FAA (Federal Aviation Authority) and NASA (National Aeronautics and Space Administration) about the use of formal methods concludes that Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied maths is a necessary part of the education of all other engineers

* The bug in Intel’s Pentium II floating-point division unit in the early nineties caused a loss of about 475 million US dollars to replace faulty processors, and severely damaged Intel’s reputation as a reliable chip manufacturer. The software error in a baggage handling system postponed the opening of Denver’s airport for 9 months, at a loss of 1.1 million US dollar per day. Twenty-four hours of failure of the worldwide online ticket reservation system of a large airplane company will cause its bankruptcy because of missed orders. It is all about safety: errors can be catastrophic too. The fatal defects in the control software of the Ariane-5 missile (Figure 1.1), the Mars Pathfinder, and the airplanes of the Airbus family led to headlines in newspapers all over the world and are notorious by now. Similar software is used for the process control of safety-critical systems such as chemical plants, nuclear power plants, traffic control and alert systems, and storm surge barriers. Clearly, bugs in such software can have disastrous consequences. For example, a software flaw in the control part of the radiation therapy machine Therac-25 caused the death of six cancer patients between 1985 and 1987 as they were exposed to an overdose of radiation. The increasing reliance of critical applications on information processing leads us to state: The reliability of ICT systems is a key issue in the system design process.
* Software Engineering jauh lebih tidak reliable dari other eng

Holloway

Software is notorious for being late in delivery and unpredictable and unreliable in operation. According to a 1994 article by Wayt Gibbs, “Studies have shown that for every six new large-scale software systems that are put into operation, two others are cancelled. The average software development project overshoots its schedule by half; larger projects generally do worse. And three quarters of all large systems are operating failures that either do not function as intended or are not used at all.” [11] When compared to other engineering disciplines, software engineering does not come out looking good. But this should not be surprising, because in at least two respects, software is different from the physical objects, materials, and systems with which traditional engineers work

First, in physical systems smooth changes in inputs usually produce smooth changes in outputs. That is, most physical systems are continuous. This allows the behavior of the system to be determined by testing only certain inputs, and using extrapolation and interpolation to determine the behaviors for untested inputs. Software systems are, by their very nature, discontinuous. A small change in input may change the outcomes at several decision points within the software, causing very different execution paths and major changes in output behavior. As a result, using extrapolation and interpolation to estimate output behaviors for untested inputs is risky at best, and exceedingly dangerous at worst. Software differs from physical systems in another way: its complexity. Much of the functionality of modern systems is provided by software; therefore, much of the complexity of these systems is expressed in the software, also. The greater the complexity, the more likely design flaws — flaws in the intellectual construction of the system that cause it to do the wrong thing under some conditions — are to occur. Design flaws are the only way that software can go wrong; software does not wear out like physical components. Thus, to ensure that a software system does what it is intended to do, design flaws must be handled in some way

* Kenapa metode formal jarang dipake

Pena

In spite of such a big effort, and of the fact that many universities include formal program verification in their curricula, forty years after we should admit that formal verification is far from being part of dayto-day programming. There are a number of reasons for this situation: • It takes some effort to formalise the specification of methods by writing a precondition and a postcondition for each of them. • It takes much more effort to guess the loop invariants, and other critical intermediate assertions of programs. • Even having written all the critical assertions, writing and proving by hand all the verification conditions, need writing a text between 5 to 10 times the volume of the code being verified. The general impression is then than formal verification gives us obvious benefits, but the effort investment needed to get them is too high. As a consequence, formal methods are barely used, and only in a few safety critical systems such an investment seems to be justified.

* Contoh metode formal yang sudah dipake

Baier

One of the oldest large-scale experiments is the CICS project undertaken at IBM (Huxley Park, United Kingdom), in collaboration with Oxford University. Its purpose was to perform a major restructuring of a large existing software system used for transaction management. The overall system was composed of about 800,000 lines of assembly language and of Plas, a high-level proprietary language. 268,000 lines were modified or rewritten, of which 37,000 made use of formal specification with the Z specification notation. Measurem ~nt procedures were introduced in order to evaluate the impact of a formal method on productivity and on quality. The quantitative results are detailed in [HK91]. They can be summarized as follows: - development costs decreased by 9 percent; - in the first eight months following the installation of the new version of CICS, in 1990, the clients reported 2.5 times fewer errors in the parts developed with Z than in the parts developed with non-formal techniques; moreover, these errors were perceived as being less serious. This experiment is interesting because of the large amount of code involved. In contrast, its technical goals were rather limited: the issue was to specify software with the Z formal notation, and then to develop the code from the documents resulting from this phase; proof techniques were not taken into account.

* Kelemahan Metode Formal

Monin

First, we have to keep in mind that there always remains a distance between a formal specification, and the object it is supposed to represent. A similar wellknown situation is true of the laws of physics: we cannot prove that they govern the real world, but it is quite reasonable to be confident that this is the case. The certainty of the correctness or appropriateness of a specification can be accepted as relevant only if it has been validated by a process composed of careful reading, reformulation, and confrontation. When a new formal method is considered, the first obstacle to be overcome is to become fully acquainted with the notation. Beyond this stage, formal methods require an appropriate application, which includes pragmatic aspects - manipulation of tools - and theoretical aspects. Note, in passing, that the mathematical culture developed in traditional scholarly programs often favorl'l analysis to the detriment of discrete mathematics. The situation is improving nowadays, but it is symptomatic that we still feel the need to inform about formal methods for software, whereas in other engineering disciplines, such as electronics or aircraft engineering, mathematical models are natumlly applied. This acknowledges the rather experimental light in which programming is still commonly perceived. Finally, let us note that with formal approaches, much more time is devoted to the initial phases of a development (specification, design) than in common processes. However, experiments show that this investment is (partly) compensated in later phases (tests, integration). Indeed, formalization reveals delicate issues very early, whereas, in a conventional lifecycle, these would have to be solved during debugging, or later. Many difficulties that are met when using a formal method are actually a reflection of difficulties that are inherent in the problem at hand. For example, modeling problems will occur just because the situation is intrinsically more complicated than it may appear at first sight. The introduction of complex or abstract concepts - often denoted by mathematical symbols - is then not that surprising. We will see that actual formal techniques offer various degrees of abstraction level and mathematical complexity. But to reassure the reader: basic concepts in logic and set theory, understandable to high-school students, are sufficient for a working knowledge of techniques such as B. O

#### Justifikasi Logis [Huth2004](#fn-huth2004)

* Aturan Logis
* Logika Propositional
* Logika Predikat
* SAT and SMT Solver

#### Spesifikasi Program

Huth

The task of specifying and verifying code is often perceived as an unwelcome addition to the programmer’s job and a dispensable one. Arguments in favour of verification include the following: Documentation: The specification of a program is an important component in its documentation and the process of documenting a program may raise or resolve important issues. The logical structure of the formal specification, written as a formula in a suitable logic, typically serves as a guiding principle in trying to write an implementation in which it holds. Time-to-market: Debugging big systems during the testing phase is costly and time-consuming and local ‘fixes’ often introduce new bugs at other places. Experience has shown that verifying programs with respect to formal specifications can significantly cut down the duration of software development and maintenance by eliminating most errors in the planning phase and helping in the clarification of the roles and structural aspects of system components. Refactoring: Properly specified and verified software is easier to reuse, since we have a clear specification of what it is meant to do. Certification audits: Safety-critical computer systems – such as the control of cooling systems in nuclear power stations, or cockpits of modern aircrafts – demand that their software be specified and verified with as much rigour and formality as possible. Other programs may be commercially critical, such as accountancy software used by banks, and they should be delivered with a warranty: a guarantee for correct performance within proper use. The proof that a program meets its specifications is indeed such a warranty

Monin

In line 3 we have the unknown: the expected program. In lines 1 and 2 we have two assertions stating what we know before the execution of the program: they are called the preconditions. In line 4 we have another assertion, the postcondition, to describe the result. The desired program is then specified by a pair (precondition, postcondition). This is one of the basic principles of formal specification. What does it meaning? In a real-life (and complete) formal specification, assertions would be logical formulas, that can be assigned a mathematical meaning - a semantics. For the moment let us content ourselves with their intuitive meaning, as we stated previously. This specification is concerned with the state of the world, or merely that tiny part of it we are interested in here. In concrete terms, it is just computer memory, or at least an abstract version of it. The precondition2 states here that the state has two components, a set T and a predicate P, whereas the postcondition states that it contains an additional component, the element Xj moreover, T, P and x must satisfy the aforementioned conditions. The meaning of a specification expressed in this form (precondition, postcondition) is then: If the program is executed from a state satisfying the precondition, then, after execution, the state reached satisfies the postcondition.

#### Verifikasi Program

Baier

System verification techniques are being applied to the design of ICT systems in a more reliable way. Briefly, system verification is used to establish that the design or product under consideration possesses certain properties. The properties to be validated can be quite elementary, e.g., a system should never be able to reach a situation in which no progress can be made (a deadlock scenario), and are mostly obtained from the system’s specification. This specification prescribes what the system has to do and what not, and thus constitutes the basis for any verification activity. A defect is found once the system does not fulfill one of the specification’s properties. The system is considered to be “correct” whenever it satisfies all properties obtained from its specification. So correctness is always relative to a specification, and is not an absolute property of a system.

Huth

Formal verification techniques can be thought of as comprising three parts: a framework for modelling systems, typically a description language of some sort; a specification language for describing the properties to be verified; a verification method to establish whether the description of a system satisfies the specification.

Approaches to Verification

Huth

Approaches to verification can be classified according to the following criteria:

Proof-based vs. model-based. In a proof-based approach, the system description is a set of formulas Γ (in a suitable logic) and the specification is another formula φ. The verification method consists of trying to find a proof that Γ |− φ. This typically requires guidance and expertise from the user. In a model-based approach, the system is represented by a model M for an appropriate logic. The specification is again represented by a formula φ and the verification method consists of computing whether a model M satisfies φ (written M φ). This computation is usually automatic for finite models. In Chapters 1 and 2, we could see that logical proof systems are often sound and complete, meaning that Γ |− φ (provability) holds if, and only if, Γ φ (semantic entailment) holds, where the latter is defined as follows: for all models M, if for all ψ ∈ Γ we have M ψ, then M φ. Thus, we see that the model-based approach is potentially simpler than the proof-based approach, for it is based on a single model M rather than a possibly infinite class of them.

Degree of automation. Approaches differ on how automatic the method is; the extremes are fully automatic and fully manual. Many of the computer-assisted techniques are somewhere in the middle.

Full- vs. property-verification. The specification may describe a single property of the system, or it may describe its full behaviour. The latter is typically expensive to verify.

Intended domain of application, which may be hardware or software; sequential or concurrent; reactive or terminating; etc. A reactive system is one which reacts to its environment and is not meant to terminate (e.g., operating systems, embedded systems and computer hardware).

Pre- vs. post-development. Verification is of greater advantage if introduced early in the course of system development, because errors caught earlier in the production cycle are less costly to rectify. (It is alleged that Intel lost millions of dollars by releasing their Pentium chip with the FDIV error.)

4.1 Intrinsic vs. Extrinsic Verication

Vazou

e translation of the chunkable monoid specication of § 3.1 in Coq is a characteristic instance of how Liquid Haskell and Coq naturally favor intrinsic and extrinsic verication respectively. e (intrinsic) Liquid Haskell pre- and post-conditions of the take and drop functions are not embedded in the Coq types, but are independently, i.e., extrinsically, encoded as specication terms in the extra drop\_spec and take\_spec methods. (We use the double-lined code to frame Coq code.) Liquid Haskell favors intrinsic verication, as the shallow speci- cations of take and drop are embedded into the functions and automatically proved by the SMT solver. On the contrary, Coq users can (and usually) take the extrinsic verication approach, where the specications of take and drop are encoded as independent specication terms. Since, unlike Liquid Haskell’s implicit and SMT-automatic proofs, the Coq specication terms should be explicitly proved by the user, the extrinsic approach signicantly improves readability and ease-of-use of Coq code, as the function implementations are not liered by the specications’ proofs

Software and Hardware Verification

Baier

Software Verification Peer reviewing and testing are the major software verification techniques used in practice. A peer review amounts to a software inspection carried out by a team of software engineers that preferably has not been involved in the development of the software under review. The uncompiled code is not executed, but analyzed completely statically. Empirical studies indicate that peer review provides an effective technique that catches between 31 % and 93 % of the defects with a median around 60%. While mostly applied in a rather ad hoc manner, more dedicated types of peer review procedures, e.g., those that are focused at specific error-detection goals, are even more effective. Despite its almost complete manual nature, peer review is thus a rather useful technique. It is therefore not surprising that some form of peer review is used in almost 80% of all software engineering projects. Due to its static nature, experience has shown that subtle errors such as concurrency and algorithm defects are hard to catch using peer review. Software testing constitutes a significant part of any software engineering project. Between 30% and 50% of the total software project costs are devoted to testing. As opposed to peer review, which analyzes code statically without executing it, testing is a dynamic technique that actually runs the software. Testing takes the piece of software under consideration and provides its compiled code with inputs, called tests. Correctness is thus determined by forcing the software to traverse a set of execution paths, sequences of code statements representing a run of the software. Based on the observations during test execution, the actual output of the software is compared to the output as documented in the system specification. Although test generation and test execution can partly be automated, the comparison is usually performed by human beings. The main advantage of testing is that it can be applied to all sorts of software, ranging from application software (e.g., e-business software) to compilers and operating systems. As exhaustive testing of all execution paths is practically infeasible; in practice only a small subset of these paths is treated. Testing ccan thus never be complete. That is to say, testing can only show the presence of errors, not their absence. Another problem with testing is to determine when to stop. Practically, it is hard, and mostly impossible, to indicate the intensity of testing to reach a certain defect density – the fraction of defects per number of uncommented code lines. Studies have provided evidence that peer review and testing catch different classes of defects at different stages in the development cycle. They are therefore often used together. To increase the reliability of software, these software verification approaches are complemented with software process improvement techniques, structured design and specification methods (such as the Unified Modeling Language), and the use of version and configuration management control systems. Formal techniques are used, in one form or another, in about 10 % to 15% of all software projects. These techniques are discussed later in this chapter. Catching software errors: the sooner the better. It is of great importance to locate software bugs. The slogan is: the sooner the better. The costs of repairing a software flaw during maintenance are roughly 500 times higher than a fix in an early design phase (see Figure 1.3). System verification should thus take place early stage in the design process

About 50% of all defects are introduced during programming, the phase in which actual coding takes place. Whereas just 15% of all errors are detected in the initial design stages, most errors are found during testing. At the start of unit testing, which is oriented to discovering defects in the individual software modules that make up the system, a defect density of about 20 defects per 1000 lines of (uncommented) code is typical. This has been reduced to about 6 defects per 1000 code lines at the start of system testing, where a collection of such modules that constitutes a real product is tested. On launching a new software release, the typical accepted software defect density is about one defect per 1000 lines of code lines1. Errors are typically concentrated in a few software modules – about half of the modules are defect free, and about 80% of the defects arise in a small fraction (about 20%) of the modules – and often occur when interfacing modules. The repair of errors that are detected prior to testing can be done rather economically. The repair cost significantly increases from about $ 1000 (per error repair) in unit testing to a maximum of about $ 12,500 when the defect is demonstrated during system operation only. It is of vital importance to seek techniques that find defects as early as possible in the software design process: the costs to repair them are substantially lower, and their influence on the rest of the design is less substantial.

Hardware Verification Preventing errors in hardware design is vital. Hardware is subject to high fabrication costs; fixing defects after delivery to customers is difficult, and quality expectations are high. Whereas software defects can be repaired by providing users with patches or updates – nowadays users even tend to anticipate and accept this – hardware bug fixes after delivery to customers are very difficult and mostly require refabrication and redistribution. This has immense economic consequences. The replacement of the faulty Pentium II processors caused Intel a loss of about $ 475 million. Moore’s law – the number of logical gates in a circuit doubles every 18 months – has proven to be true in practice and is a major obstacle to producing correct hardware. Empirical studies have indicated that more than 50% of all ASICs (Application-Specific Integrated Circuits) do not work properly after initial design and fabrication. It is not surprising that chip manufacturers invest a lot in getting their designs right. Hardware verification is a well-established part of the design process. The design effort in a typical hardware design amounts to only 27% of the total time spent on the chip; the rest is devoted to error detection and prevention. Hardware verification techniques. Emulation, simulation, and structural analysis are the major techniques used in hardware verification. Structural analysis comprises several specific techniques such as synthesis, timing analysis, and equivalence checking that are not described in further detail here. Emulation is a kind of testing. A reconfigurable generic hardware system (the emulator) is configured such that it behaves like the circuit under consideration and is then extensively tested. As with software testing, emulation amounts to providing a set of stimuli to the circuit and comparing the generated output with the expected output as laid down in the chip specification. To fully test the circuit, all possible input combinations in every possible system state should be examined. This is impractical and the number of tests needs to be reduced significantly, yielding potential undiscovered errors. With simulation, a model of the circuit at hand is constructed and simulated. Models are typically provided using hardware description languages such as Verilog or VHDL that are both standardized by IEEE. Based on stimuli, execution paths of the chip model are examined using a simulator. These stimuli may be provided by a user, or by automated means such as a random generator. A mismatch between the simulator’s output and the output described in the specification determines the presence of errors. Simulation is like testing, but is applied to models. It suffers from the same limitations, though: the number of scenarios to be checked in a model to get full confidence goes beyond any reasonable subset of scenarios that can be examined in practice. Simulation is the most popular hardware verification technique and is used in various design stages, e.g., at register-transfer level, gate and transistor level. Besides these error detection techniques, hardware testing is needed to find fabrication faults resulting from layout defects in the fabrication process.

There are basically two complementary ways of writing a specification: - describing the properties of a system; - providing a model of the system by means of built-in constructs. One sometimes uses the terminology property oriented and model oriented formal specification. Properties are expressed by logical axioms whereas models are derived with the help of set-theoretic operations. This duality is already present in mathematical logic, where we have a syntax for expressing logical properties and a semantics describing what we are talking about. This aspect oflogic is called model theory. One distinguishes, on the one hand, the concept of a logical statement built upon a formal language, for example: Vx3y(y > x) , (3.1) and on the other hand the concept of a model satisfying this statement; for instance, (3.1) admits, among other models, N endowed with the relation "greaterthan", lR endowed with the relation "less-than" and N endowed with the relation . ''is-a-multiple-of''.

* Verifikasi dengan Model Checking (Model Based)

Monin

A fundamental concept of model theory is the relation called logical consequence or semantic consequence. A sentence E is a semantic consequence of the sentences A, B, C ... if every model having the properties A, B, C ... has also the property E. This is a very concrete relation. Let us consider, for instance, the three properties "every terminal is a piece of equipment", "every piece of equipment possesses a registration number" and ''this phone is a terminal". A practical consequence, of interest to the department in charge of inventories, is that in any situation where the above three properties hold true, we have, systematically, ''this phone possesses a registration number". The concept of model is represented here by what we just called a situation.

Baier

Model-based verification techniques are based on models describing the possible system behavior in a mathematically precise and unambiguous manner. It turns out that – prior to any form of verification – the accurate modeling of systems often leads to the discovery of incompleteness, ambiguities, and inconsistencies in informal system specifications. Such problems are usually only discovered at a much later stage of the design. The system models are accompanied by algorithms that systematically explore all states of the system model. This provides the basis for a whole range of verification techniques ranging from an exhaustive exploration (model checking) to experiments with a restrictive set of scenarios in the model (simulation), or in reality (testing). Due to unremitting improvements of un-derlying algorithms and data structures, together with the availability of faster computers and larger computer memories, model-based techniques that a decade ago only worked for very simple examples are nowadays applicable to realistic designs. As the startingpoint of these techniques is a model of the system under consideration, we have as a given fact that Any verification using model-based techniques is only as good as the model of the system. Model checking is a verification technique that explores all possible system states in a brute-force manner. Similar to a computer chess program that checks possible moves, a model checker, the software tool that performs the model checking, examines all possible system scenarios in a systematic manner. In this way, it can be shown that a given system model truly satisfies a certain property. It is a real challenge to examine the largest possible state spaces that can be treated with current means, i.e., processors and memories. Stateof-the-art model checkers can handle state spaces of about 108 to 109 states with explicit state-space enumeration. Using clever algorithms and tailored data structures, larger state spaces (1020 up to even 10476 states) can be handled for specific problems. Even the subtle errors that remain undiscovered using emulation, testing and simulation can potentially be revealed using model checking.

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

In applying model checking to a design the following different phases can be distinguished: Modeling phase: – model the system under consideration using the model description language of the model checker at hand; – as a first sanity check and quick assessment of the model perform some simulations; – formalize the property to be checked using the property specification language. • Running phase: run the model checker to check the validity of the property in the system model. • Analysis phase: – property satisfied? → check next property (if any); – property violated? → 1. analyze generated counterexample by simulation; 2. refine the model, design, or property; 3. repeat the entire procedure. – out of memory? → try to reduce the model and try again. In addition to these steps, the entire verification should be planned, administered, and organized. This is called verification organization.

The strengths of model checking: • It is a general verification approach that is applicable to a wide range of applications such as embedded systems, software engineering, and hardware design. • It supports partial verification, i.e., properties can be checked individually, thus allowing focus on the essential properties first. No complete requirement specification is needed. • It is not vulnerable to the likelihood that an error is exposed; this contrasts with testing and simulation that are aimed at tracing the most probable defects. • It provides diagnostic information in case a property is invalidated; this is very useful for debugging purposes. It is a potential “push-button” technology; the use of model checking requires neither a high degree of user interaction nor a high degree of expertise. • It enjoys a rapidly increasing interest by industry; several hardware companies have started their in-house verification labs, job offers with required skills in model checking frequently appear, and commercial model checkers have become available. • It can be easily integrated in existing development cycles; its learning curve is not very steep, and empirical studies indicate that it may lead to shorter development times. • It has a sound and mathematical underpinning; it is based on theory of graph algorithms, data structures, and logic.

The weaknesses of model checking: • It is mainly appropriate to control-intensive applications and less suited for dataintensive applications as data typically ranges over infinite domains. • Its applicability is subject to decidability issues; for infinite-state systems, or reasoning about abstract data types (which requires undecidable or semi-decidable logics), model checking is in general not effectively computable. • It verifies a system model, and not the actual system (product or prototype) itself; any obtained result is thus as good as the system model. Complementary techniques, such as testing, are needed to find fabrication faults (for hardware) or coding errors (for software). • It checks only stated requirements, i.e., there is no guarantee of completeness. The validity of properties that are not checked cannot be judged. • It suffers from the state-space explosion problem, i.e., the number of states needed to model the system accurately may easily exceed the amount of available computer memory. Despite the development of several very effective methods to combat this problem (see Chapters 7 and 8), models of realistic systems may still be too large to fit in memory. • Its usage requires some expertise in finding appropriate abstractions to obtain smaller system models and to state properties in the logical formalism used. • It is not guaranteed to yield correct results: as with any tool, a model checker may contain software defects. It does not allow checking generalizations: in general, checking systems with an arbitrary number of components, or parameterized systems, cannot be treated. Model checking can, however, suggest results for arbitrary parameters that may be verified using proof assistants.

Model checking is an effective technique to expose potential design errors.

LTL

Linear-time temporal logic, or LTL for short, is a temporal logic, with connectives that allow us to refer to the future. It models time as a sequence of states, extending infinitely into the future. This sequence of states is sometimes called a computation path, or simply a path. In general, the future is not determined, so we consider several paths, representing different possible futures, any one of which might be the ‘actual’ path that is realised.

CTL

In our analysis of LTL (linear-time temporal logic) in the preceding sections, we noted that LTL formulas are evaluated on paths. We defined that a state of a system satisfies an LTL formula if all paths from the given state satisfy it. Thus, LTL implicitly quantifies universally over paths. Therefore, properties which assert the existence of a path cannot be expressed in LTL. This problem can partly be alleviated by considering the negation of the property in question, and interpreting the result accordingly. To check whether there exists a path from s satisfying the LTL formula φ, we check whether all paths satisfy ¬φ;apositive answer to this is a negative answer to our original question, and vice versa. We used this approach when analysing the ferryman puzzle in the previous section. However, as already noted, properties which mix universal and existential path quantifiers cannot in general be model checked using this approach, because the complement formula still has a mix. Branching-time logics solve this problem by allowing us to quantify explicitly over paths. We will examine a logic known as Computation Tree Logic, or CTL. In CTL, as well as the temporal operators U, F, G and X of LTL we also have quantifiers A and E which express ‘all paths’ and ‘exists a path’, respectively. For example, we can write: There is a reachable state satisfying q: this is written EF q. From all reachable states satisfying p, it is possible to maintain p continuously until reaching a state satisfying q: this is written AG (p → E[p U q]). Whenever a state satisfying p is reached, the system can exhibit q continuously forevermore: AG (p → EG q). There is a reachable state from which all reachable states satisfy p: EF AG p

CTL\*

CTL\* is a logic which combines the expressive powers of LTL and CTL, by dropping the CTL constraint that every temporal operator (X, U, F, G) has to be associated with a unique path quantifier (A, E). It allows us to write formulas such as A[(p U r) ∨ (q U r)]: along all paths, either p is true until r, or q is true until r. A[X p ∨ X X p]: along all paths, p is true in the next state, or the next but one. E[G F p]: there is a path along which p is infinitely often true.

* Verifikasi dengan Semantic Entailment (Proof Based)

Huth

The methods of the previous chapter are suitable for verifying systems of communicating processes, where control is the main issue, but there are no complex data. We relied on the fact that those (abstracted) systems are in a finite state. These assumptions are not valid for sequential programs running on a single processor, the topic of this chapter. In those cases, the programs may manipulate non-trivial data and – once we admit variables of type integer, list, or tree – we are in the domain of machines with infinite state space.

In terms of the classification of verification methods given at the beginning of the last chapter, the methods of this chapter are Proof-based. We do not exhaustively check every state that the system can get in to, as one does with model checking; this would be impossible, given that program variables can have infinitely many interacting values. Instead, we construct a proof that the system satisfies the property at hand, using a proof calculus. This is analogous to the situation in Chapter 2, where using a suitable proof calculus avoided the problem of having to check infinitely many models of a set of predicate logic formulas in order to establish the validity of a sequent. Semi-automatic. Although many of the steps involved in proving that a program satisfies its specification are mechanical, there are some steps that involve some intelligence and that cannot be carried out algorithmically by a computer. As we will see, there are often good heuristics to help the programmer complete these tasks. This contrasts with the situation of the last chapter, which was fully automatic. Property-oriented. Just like in the previous chapter, we verify properties of a program rather than a full specification of its behaviour. Application domain. The domain of application in this chapter is sequential transformational programs. ‘Sequential’ means that we assume the program runs on a single processor and that there are no concurrency issues. ‘Transformational’ means that the program takes an input and, after some computation, is expected to terminate with an output. For example, methods of objects in Java are often programmed in this style. This contrasts with the previous chapter which focuses on reactive systems that are not intended to terminate and that react continually with their environment. Pre/post-development. The techniques of this chapter should be used during the coding process for small fragments of program that perform an identifiable (and hence, specifiable) task and hence should be used during the development process in order to avoid functional bugs.

Monin 3.3.2

Of course, the logician must ensure that those formal manipulations respect the semantics, hence the concept of soundness. The converse property (every semantic consequence is provable) is a form of completeness. Another kind of completeness relates a collection of formulas r with one intended model M, stating that the latter is completely characterized by r, i.e. every true (respectively false) formula in M is provable (respectively refutable) from r.

Huth

A framework for software verification

4.2.1 A core programming language

4.2.2 Hoare triples

4.2.3 Partial and total correctness

4.2.4 Program variables and logical variables

### Pemrograman Fungsional

* Paradigma Pemrograman [Harper2017](#fn-harper2017)

Van Roy

A programming paradigm is an approach to programming a computer based on a mathematical theory or a coherent set of principles. Each paradigm supports a set of concepts that makes it the best for a certain kind of problem. For example, object-oriented programming is best for problems with a large number of related data abstractions organized in a hierarchy. Logic programming is best for transforming or navigating complex symbolic structures according to logical rules. Discrete synchronous programming is best for reactive problems, i.e., problems that consist of reactions to sequences of external events. Languages that support these three paradigms are respectively Java, Prolog, and Esterel.

Michaelson

The contemporary idea of a programming paradigm may well have originated with Floyd’s 1978 Turing Award Lecture[6] “The Paradigms of Programming”. Floyd’s notion draws explicitly on Kuhn’s. However, his focus is on programming as a problem solving methodology supported by, rather than characterised by, a language. Thus, he contrasts structured programming, based on stepwise refinement followed by information hiding, with branch-and-bound, divide-andconquer and state-machine paradigms. Floyd also emphasises that support environments are as important as languages in supporting paradigms.

Tedre[25] questions whether Computer Science is mature enough to even speak of paradigms. We certainly can’t identify disjunctions in programming practice comparable to those from, say, Newtonian to relativistic mechanics. Rather, we see an evolution, where older programming practices build on and are absorbed into newer ones. For example, 70 years on from ENIAC, flowcharts are still used, and 1990s UML includes 1960s state machines. We also can’t identify disjunctions in language use; rather, new languages coexist with the old, which slowly fade away or morph to more closely resemble the new. Thus, C and ML now both have OO descendents, in OCaml , and C++ and C#, respectively. Despite the best endeavours of language warriors, programming paradigms are much more akin to Kuhn’s co-extant traditions rooted within a common paradigm.. This is hardly surprising. As Turing and Church hypothesised well before digital computers[3], all known models of computability may be shown to be Turing complete, that is equivalent to Turing machines, through sound schema for translating instances of any one model into instances of any other. Similarly, all “full strength” programming languages may be shown to be equivalent, through schema for translating a program in any one language into any other. Indeed, it would be most disconcerting if programs that supposedly solved the same problem in different languages gave different outputs from the same inputs. Ultimately, just as all flesh is grass, all executable code is machine code. Thus, I strongly contest Tedre. As we saw, for Kuhn, one paradigm is dominant in any given epoch, until it is displaced by another paradigm with greater explanatory power. Thus, in Kuhnian terms, Turing complete finite computation, now over 80 years old, is patently the dominant paradigm for Computer Science, providing a substantial body of mathematical theory underpinning the world changing practice of constructing and programming computers. Arguably, Turing complete computation is the first as well as the only Computier Science paradigm. It was thought that analogue computing, based on differential equations reified as differential analysers and then amplifiers, constituted a prior paradigm, but analogue computing has also been shown to be Turing complete[11]. Similarly, quantum computing is widely heralded as disruptive but Deutch[5] long established that it is Turing complete. Trans-finite hypercomputation would constitute a new paradigm by definition, as hypercomputers can allegedly solve decidability problems that Turing machines can’t, as, for example, Hogarth argues[14]. However, the possibility of hypercomputation is hotly contested[4]

* Paradigma Pemrograman Fungsional [Sabry1998](#fn-sabry1998)

[Hudak, Paul](https://en.wikipedia.org/wiki/Paul_Hudak) (September 1989). ["Conception, evolution, and application of functional programming languages"](http://www.dbnet.ece.ntua.gr/~adamo/languages/books/p359-hudak.pdf) (PDF). *ACM Computing Surveys*. **21** (3): 359–411. [doi](https://en.wikipedia.org/wiki/Doi_(identifier)):[10.1145/72551.72554](https://doi.org/10.1145%2F72551.72554).

The class of functional, or applicative, programming languages, in which computation is carried out entirely through the evaluation of expressions, is one such family of languages,

Among the claims made by functional language advocates are that programs can be written quicker, are more concise, are higher level (resembling more closely traditional mathematical notation), are more amenable to formal reasoning and analysis, and can be executed more easily on parallel architectures. Of course, many of these features touch on rather subjective issues, which is one reason why the debates can be so lively.

Imperative languages are characterized as having an implicit state that is modified (i.e., side effected) by constructs (i.e., commands) in the source language. As a result, such languages generally have a notion of sequencing (of the commands) to permit precise and deterministic control over the state. Most, including the most popular, languages in existence today are imperative.

In contrast, declarative languages are characterized as having no implicit state, and thus the emphasis is placed entirely on programming with expressions (or terms). In particular, functional languages are declarative languages whose underlying model of computation is the function (in contrast to, for example, the relation that forms the basis for logic programming languages).

* Functional (in general, declarative) programming is often described as expressing what is being computed rather than how, although this is really a matter of degree.
* Paradigram Pemrograman Fungsional Murni

[Sabry1998](#fn-sabry1998)

In essence, the new definition asserts that a language is purely functional if it can be implemented using either call-by-value, call-by-need, or call-by-name, with no observable difference between the different strategies – other than termination properties.

Definition 4.7 (Purely Functional Language) A language is purely functional if: 1. It is a conservative extension of the simply typed λ-calculusDefinition 4.7 (Purely Functional Language) A language is purely functional if: 1. It is a conservative extension of the simply typed λ-calculus 2. It has well-defined call-by-value, call-by-need and call-by-name evaluation functions (implementations). 3. All three evaluation functions (implementations) are weakly equivalent.

There are several important points to note: • The first condition in the definition requires that the language be a conservative extension of the simply typed λ-calculus. This condition guards against languages with no functions, and hence that would vacuously satisfy the second and third conditions. • Among the many parameter-passing mechanisms we have selected call-byvalue, call-by-name, and call-by-need as the relevant ones for the thesis. This choice appears to work well as it allows us to verify that the subset of SML (a call-by-value language) without assignments and exceptions is pure, and also that Haskell (a language with a call-by-name denotational semantics and a call-by-need implementation) is pure. It may be the case that the thesis could be formulated with only two of the parameter-passing mechanisms, for example by omitting call-by-value entirely. This new thesis would essentially be about sharing of computations since this is the fundamental difference between call-by-name and call-by-need. We leave this point as an open problem. • A drawback of this definition is that it requires the existence of several evaluation functions (implementations) for the same syntax. Starting from a call-by-value language like Scheme, it is straightforward to devise a callby-need or call-by-name evaluator. However, starting from a call-by-name language like Λ! or Idealized Algol (Reynolds, 1991; Reynolds, 1981; Reynolds, 1988), the design of the call-by-value or call-by-need variant first requires setting a notion of syntactic value. This latter decision affects the purity of the language. Indeed, as we will see in the next section, by varying the notion of value in Λ!, we can design a new variant of the language that is purely functional. • The thesis follows the convention that non-termination and errors are special kinds of computation whose effects are not observable. Hence expressions that diverge, or evaluate to a black hole, or an error are all considered equivalent. If errors become observable, then not even PCF would be pure (Cartwright and Felleisen, 1991; Cartwright et al., 1993)

The paper proposes a framework for reasoning about purely functional languages and their extensions with computational effects. We have put forward the thesis that purity can be determined by the (weak) equivalence of call-by-name, call-byvalue, and call-by-need. This definition of purity naturally motivates and explains the various strategies used to integrate computational effects with purely functional languages.

* Kenapa Fungsional Murni itu excellent buat Metode Formal [Turner1985](#fn-turner1985) [Hughes1989](#fn-hughes1989) [Butler1995](#fn-butler1995)

Butler

Functional programming lauguages are declarative, and are fi”entally Merent fiom the more conventid procedural languages such as Pascal and ‘C’. A functiooal programming language has only one basic operation, the application of a function to arguments. Functional programming languages have properties which make them suitable for formal specification. They consist only of function calls, or expressions, and include no assignment statements. A variable, once defined to denote a given expression, never changes value within its scope and thus no side effects are admitted. This means that functional programs have the property of referential transparency - that an expression and its value may be fieely interchanged. Such programs are thus amenable to formal reasoning. Functional programming languages have a sound mathematical basis in 5- calculus. The perceived advantage of using a functional language for fomd specification is that the system can subsequently be implemented in the same language, removing the need for a paradigm change and thus reducing the effort required to move from specihition to implementation.

We found that many of the features offered by Gofer/Haskeil are amducive to writing clear and concise specifications. Its succinct and flexible semantics provide list comprehension, Aqressions, infix operators and pattern matching. User defined data types including sets, bags and trees can be added to those provided within the language. With minor extensions the language can be used to express requirements implicitly in terms of predicates as well as constructively, thus allowing abstmction and loose specification. Polymorphic strong typing allows functions to be defined over parametensed sets of tupes, aiding function reuse and ensuring that many classes of error are detected by the type system. Type classes are also provided, giving some of the benefits of object oriented approaches. The Gofer interpreter provides rapid type and syntax checking allowing early elimination of many types of error, and provides fkiities for locating defhitions across a system consisting of a number of modules. A literate script style is supported, enabling integration of the formal statements with formatted explanatory text, diagrams and tables. After development of an animation of the de”&& ’on system (a My hctid but inefficient implementation), some formal verification of the animation against the specification was carried out. This was done on a samplmg basis due to limited resource. We found that the functional programming language approach was well suited to this task. Two related approaches were taken. Firstly, proof of the equivalence of the specification and the implementation is possible. A modular approach can be taken since referential transparency allows individual functons to be considered independentl y of the contexts in which they are used. Secondly, theorem about the specification and the animation can be expressed in Gofer and their truth or falsity determined by mathematical reasoning. The main drawback to this approach to specification is that functional programming Iauguages are still largely a research topic. There is as yet no published formal semantics for Haskell, and automated proof tools or proof assistants are not available. In conclusion, the SADLI project has demonstrakd that there is no barrier to using a hctional programming language for both formal specification and implementahon. The use of a single mathematically formal paradigm across successive phases of the software Mqcle is likely to minimise transformation errors whilst providing potential for formal proof that the implementation satisfies the specification. A further advantage of this approach is that the project team needs to leam only one language, which could result in reduced training time and costs. Durbg the project we found the learning curve for the hctiod programming language used to be short. However, the use of a single paradigm means that progression from specification to implementation becomes a continuous evolutionary process. This presents problems for project management in that firm project “es become difficult to define. The major shortcoming of the approach taken is that functional programming languages are still largely a research area and there are a number of technical advances which are required before our approach becomes practical. Most importantly for formal specification purposes, a complete definition of the formal semantics for Haskell has not yet been published, although this area is currently beii addressed by the Glasgow Haskell group at the University of Glasgow, and tool support for formal proof is not yet available.

* Perbandingan dengan Paradigma lain

#### Haskell [O'Sullivan2008](#fn-o'sullivan2008)

* Definisi, Penemu, Filosofi [Hudak2007](#fn-hudak2007)

Haskell is a general-purpose, purely functional programming language exhibiting many of the recent innovations in functional (as well as other) programming language research, including higher order functions, lazy evaluation, static polymorphic typing, user-defined datatypes, pattern matching, and list comprehensions. It is also a very complete language in that it has a module facility, a well-defined functional I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers. in this sense Haskell represents both the culmination and solidification of many years of research on functional languages-the design was influenced by languages as old as Iswim and as new as Miranda. Haskell also has several interesting new features; most notably, a systematic treatment of overloading, an orthogonal abstract datatype facility, a universal and purely functional I/O system, and, by analogy to list comprehensions, a notion of array comprehensions. Haskell is not a small language. The decision to emphasize certain features such as pattern matching and user-defined datatypes and the desire for a complete and practical language that includes such things as I/O and modules necessitates a somewhat large design. The Haskell Report also provides a denotational semantics for both the static and dynamic behavior of the language; it is considerably more complex than the simple semantics defined in Section 2.5 for the lambda calculus, but then again one wouldn’t really want to program in as sparse a language as the lambda calculus.

* Fungsional Murni

Hudak2007 Lazy

An immediate consequence of laziness is that evaluation order is demand-driven. As a result, it becomes more or less impossible to reliably perform input/output or other side effects as the result of a function call. Haskell is, therefore, a pure language. For example, if a function f has type Int -> Int you can be sure that f will not read or write any mutable variables, nor will it perform any input/output. In short, f really is a function in the mathematical sense: every call (f 3) will return the same value. Once we were committed to a lazy language, a pure one was inescapable. The converse is not true, but it is notable that in practice most pure programming languages are also lazy. Why? Because in a call-by-value language, whether functional or not, the temptation to allow unrestricted side effects inside a “function” is almost irresistible. Purity is a big bet, with pervasive consequences. Unrestricted side effects are undoubtedly very convenient. Lacking side effects, Haskell’s input/output was initially painfully clumsy, which was a source of considerable embarrassment. Necessity being the mother of invention, this embarrassment ultimately led to the invention of monadic I/O, which we now regard as one of Haskell’s main contributions to the world, as we discuss in more detail in Section 7. Whether a pure language (with monadic effects) is ultimately the best way to write programs is still an open question, but it certainly is a radical and elegant attack on the challenge of programming, and it was that combination of power and beauty that motivated the designers. In retrospect, therefore, perhaps the biggest single benefit of laziness is not laziness per se, but rather that laziness kept us pure, and thereby motivated a great deal of productive work on monads and encapsulated state.

* Lazy Computing
  + Hudak2007 Lazy

Laziness was undoubtedly the single theme that united the various groups that contributed to Haskell’s design. Technically, Haskell is a language with a non-strict semantics; lazy evaluation is simply one implementation technique for a non-strict language. Nevertheless the term “laziness” is more pungent and evocative than “non-strict,” so we follow popular usage by describing Haskell as lazy. When referring specifically to implementation techniques we will use the term “call-by-need,” in contrast with the call-by-value mechanism of languages like Lisp and ML. By the mid-eighties, there was almost a decade of experience of lazy functional programming in practice, and its attractions were becoming better understood. Hughes’s paper “Why functional programming matters” captured these in an influential manifesto for lazy programming, and coincided with the early stages of Haskell’s design. (Hughes first presented it as his interview talk when applying for a position at Oxford in 1984, and it circulated informally before finally being published in 1989 (Hughes, 1989).) Laziness has its costs. Call-by-need is usually less efficient than call-by-value, because of the extra bookkeeping required to delay evaluation until a term is required, so that some terms may not be evaluated, and to overwrite a term with its value, so that no term is evaluated twice. This cost is a significant but constant factor, and was understood at the time Haskell was designed. A much more important problem is this: it is very hard for even experienced programmers to predict the space behaviour of lazy programs, and there can be much more than a constant factor at stake. As we discuss in Section 10.2, the prevalence of these space leaks led us to add some strict features to Haskell, such as seq and strict data types (as had been done in SASL and Miranda earlier). Dually, strict languages have dabbled with laziness (Wadler et al., 1988). As a result, the strict/lazy divide has become much less an all-or-nothing decision, and the practitioners of each recognise the value of the other.

* Hard Typing

Hudak2007 Lazy

Although laziness was what brought Haskell’s designers together, it is perhaps type classes that are now regarded as Haskell’s most distinctive characteristic. Type classes were introduced to the Haskell Committee by Wadler in a message sent to the fplangc mailing list dated 24 February 1988. Initially, type classes were motivated by the narrow problem of overloading of numeric operators and equality. These problems had been solved in completely different ways in Miranda and SML. SML used overloading for the built-in numeric operators, resolved at the point of call. This made it hard to define new numeric operations in terms of old. If one wanted to define, say, square in terms of multiplication, then one had to define a different version for each numeric type, say integers and floats. Miranda avoided this problem by having only a single numeric type, called num, which was a union of unbounded-size integers and double-precision floats, with automatic conversion of int to float when required. This is convenient and flexible but sacrifices some of the advantages of static typing – for example, in Miranda the expression (mod 8 3.4) is type-correct, even though in most languages the modulus operator mod only makes sense for integer moduli.

SML also originally used overloading for equality, so one could not define the polymorphic function that took a list and a value and returned true if the value was equal to some element of the list. (To define this function, one would have to pass in an equality-testing function as an extra argument.) Miranda simply gave equality a polymorphic type, but this made equality well defined on function types (it raised an error at run time) and on abstract types (it compared their underlying representation for equality, a violation of the abstraction barrier). A later version of SML included polymorphic equality, but introduced special “equality type variables” (written ’’a instead of ’a) that ranged only over types for which equality was defined (that is, not function types or abstract types). Type classes provided a uniform solution to both of these problems. They generalised the notion of equality type variables from SML, introducing a notion of a “class” of types that possessed a given set of operations (such as numeric operations or equality). The type-class solution was attractive to us because it seemed more principled, systematic and modular than any of the alternatives; so, despite its rather radical and unproven nature, it was adopted by acclamation. Little did we know what we were letting ourselves in for! Wadler conceived of type classes in a conversation with Joe Fasel after one of the Haskell meetings. Fasel had in mind a different idea, but it was he who had the key insight that overloading should be reflected in the type of the function. Wadler misunderstood what Fasel had in mind, and type classes were born! Wadler’s student Steven Blott helped to formulate the type rules, and proved the system sound, complete, and coherent for his doctoral dissertation (Wadler and Blott, 1989; Blott, 1991). A similar idea was formulated independently by Stefan Kaes (Kaes, 1988). We elaborate on some of the details and consequences of the typeclass approach in Section 6. Meanwhile, it is instructive to reflect on the somewhat accidental nature of such a fundamental and farreaching aspect of the Haskell language. It was a happy coincidence of timing that Wadler and Blott happened to produce this key idea at just the moment when the language design was still in flux. It was adopted, with little debate, in direct contradiction to our implicit goal of embodying a tried-and-tested consensus. It had far-reaching consequences that dramatically exceeded our initial reason for adopting it in the first place.

Haskell’s type system has developed extremely anarchically. Many of the new features described above were sketched, implemented, and applied well before they were formalised. This anarchy, which would be unthinkable in the Standard ML community, has both strengths and weaknesses. The strength is that the design space is explored much more quickly, and tricky corners are often (but not always!) exposed. The weakness is that the end result is extremely complex, and programs are sometimes reduced to experiments to see what will and will not be acceptable to the compiler.

Meanwhile, it is worth asking why Haskell has proved so friendly a host language for type-system innovation. The following reasons seem to us to have been important. On the technical side: • The purity of the language removed a significant technical obstacle to many type-system innovations, namely dealing with mutable state. • Type classes, and their generalisation to qualified types (Jones, 1994), provided a rich (albeit rather complex) framework into which a number of innovations fitted neatly; examples include extensible records and implicit parameters. • Polymorphic recursion was in the language, so the idea that every legal program should typecheck without type annotations (a tenet of ML) had already been abandoned. This opens the door to features for which unaided inference is infeasible. But there were also nontechnical factors at work: • The Haskell Committee encouraged innovation right from the beginning and, far from exercising control over the language, disbanded itself in 1999 (Section 3.7). • The two most widely used implementations (GHC, Hugs) both had teams that encouraged experimentation. • Haskell has a smallish, and rather geeky, user base. New features are welcomed, and even breaking changes are accepted

* Program yang dibuat menggunakan Haskell (Hackage)
  + Hudak2007 Lazy

Some of the most important applications of Haskell were originally developed as libraries. The Haskell standard includes a modest selection of libraries, but many more are available. The Haskell web site (haskell.org) lists more than a score of categories, with the average category itself containing a score of entries. For example, the Edison library of efficient data structures, originated by Okasaki (Okasaki, 1998a) and maintained by Robert Dockins, provides multiple implementations of sequences and collections, organised using type classes. The HSQL library interfaces to a variety of databases, including MySQL, Postgres, ODBC, SQLite, and Oracle; it is maintained by Angelov. Haskell also has the usual complement of parser and lexer generators. Marlow’s Happy was designed to be similar to yacc and generated LALR parsers. (“Happy” is a “dyslexic acronym” for Yet Another Haskell Parser.) Paul Callaghan recently extended Happy to produce Generalised LR parsers, which work with ambiguous grammars, returning all possible parses. Parser combinator libraries are discussed later in this section. Documentation of Haskell programs is supported by several systems, including Marlow’s Haddock tool.

11.1 Combinator libraries One of the earliest success stories of Haskell was the development of so-called combinator libraries. What is a combinator library? The reader will search in vain for a definition of this heavily used term, but the key idea is this: a combinator library offers functions (the combinators) that combine functions together to make bigger functions.

11.2 Domain-specific embedded languages A common theme among many successful Haskell applications is the idea of writing a library that turns Haskell into a domainspecific embedded language (DSEL), a term first coined by Hudak (Hudak, 1996a; Hudak, 1998). Such DSELs have appeared in a diverse set of application areas, including graphics, animation, vision, control, GUIs, scripting, music, XML processing, robotics, hardware design, and more. By “embedded language” we mean that the domain-specific language is simply an extension of Haskell itself, sharing its syntax, function definition mechanism, type system, modules and so on. The “domain-specific” part is just the new data types and functions offered by a library. The phrase “embedded language” is commonly used in the Lisp community, where Lisp macros are used to design “new” languages; in Haskell, thanks to lazy evaluation, much (although emphatically not all) of the power of macros is available through ordinary function definitions. Typically, a data type is defined whose essential nature is often, at least conceptually, a function, and operators are defined that combine these abstract functions into larger ones of the same kind. The final program is then “executed” by decomposing these larger pieces and applying the embedded functions in a suitable manner

11.5 Natural language processing13 Haskell has been used successfully in the development of a variety of natural language processing systems and tools. Richard Frost (Frost, 2006) gives a comprehensive review of relevant work in Haskell and related languages, and discusses new tools and libraries that are emerging, written in Haskell and related languages.

* How to Specify and Verify (Haskell for Specification)

Hudak2007 Lazy

QuickCheck

While debugging tools have not yet really reached the Haskell mainstream, testing tools have been more successful. The most widely used is QuickCheck, developed by Koen Claessen and Hughes. QuickCheck is based on a cool idea that turned out to work very well in practice, namely that programs can be tested against specifications by formulating specifications as boolean functions that should always return True, and then invoking these functions on random data. For example, the function definition prop\_reverse :: [Integer] -> [Integer] -> Bool prop\_reverse xs ys = reverse (xs++ys) == reverse ys++reverse xs expresses a relationship between reverse and ++ that should always hold. The QuickCheck user can test that it does just by evaluating quickCheck prop\_reverse in a Haskell interpreter. In this case testing succeeds, but when properties fail then QuickCheck displays a counter example. Thus, for the effort of writing a simple property, programmers can test a very large number of cases, and find counter examples very quickly.

QuickCheck was first released in 1999 and was included in the GHC and Hugs distributions from July 2000, making it easily accessible to most users. A first paper appeared in 2000 (Claessen and Hughes, 2000), with a follow-up article on testing monadic code in 2002 (Claessen and Hughes, 2002). Some early success stories came from the annual ICFP programming contests: Tom Moertel (“Team Functional Beer”) wrote an account12 of his entry in 2001, with quotable quotes such as “QuickCheck to the rescue!” and “Not so fast, QuickCheck spotted a corner case. . . ,” concluding QuickCheck found these problems and more, many that I wouldn’t have found without a massive investment in test cases, and it did so quickly and easily. From now on, I’m a QuickCheck man!

Today, QuickCheck is widely used in the Haskell community and is one of the tools that has been adopted by Haskell programmers in industry, even appearing in job ads from Galois Connections and Aetion Technologies. Perhaps QuickCheck has succeeded in part because of who Haskell programmers are: given the question “What is more fun, testing code or writing formal specifications?” many Haskell users would choose the latter—if you can test code by writing formal specifications, then so much the better!

#### Liquid Haskell [Pena2017](#fn-pena2017)

* Definisi, Penemu, Filosofi

Anish

Liquid Haskell is a framework for annotating Haskell programs with refinement types,

which are types decorated with predicates. The predicates are in the language of a

decidable logic (quantifier-free logic of linear arithmetic and uninterpreted functions),

allowing the use of an SMT solver for decidable type checking. In this work, we will

write Liquid Haskell, in body text for the name of the programming language and

the system, but LiquidHaskell in small capitals for the typechecking algorithm.

Liquid Haskell comes equipped with a default abstract domain, predicate templates

that can be filled in with program variables. This default abstract domain is useful

for verifying common operations in practice — indeed it is determined empirically,

based on its ability to verify a suite of benchmark programs — but is not sufficient

to prove all programs correct.

Pena

Liquid Haskell (LH) was first introduced in [17, 18]. It represents the application of the Liquid Type theory to a full-fledged functional language like Haskell. It consists of a static type-checker for a big part of the Haskell language. The first phase of LH uses the Haskell compiler GHC [6] in order to solve the external references, to type-check the program in the Hindley-Milner sense, and to transform it to its internal Core representation. This transformation simplifies the work of LH, since it then only deals with a few syntactic constructions.

Vazou

Source L IQUID H ASKELL can be run from the command-line 1 or

within a web-browser 2 . It takes as input: (1) a single Haskell source

file with code and refinement type specifications including refined

datatype definitions, measures (§ 2.3), predicate and type aliases,

and function signatures; (2) a set of directories containing imported

modules (including the Prelude) which may themselves contain

specifications for exported types and functions; and (3) a set of

predicate fragments called qualifiers, which are used to infer refine-ment types. This set is typically empty as the default set of quali-

fiers extracted from the type specifications suffices for inference.

Core L IQUID H ASKELL uses GHC to reduce the source to the Core

IL [35], and, to facilitate source-level error reporting, creates a map

from Core expressions to locations in the Haskell source.

Constraints Then, it uses the abstract interpretation framework

of Liquid Typing [29], modified to ensure soundness under lazy

evaluation [39], to generate logical constraints from the Core IL.

Solution Next, it uses a fixpoint algorithm (from [29]) combined

with an SMT solver to solve the constraints, and hence infers a

valid refinement typing for the program. L IQUID H ASKELL can use

any solver that implements the SMT-LIB2 standard [2], including

Z3 [10], CVC4 [1], and MathSat [6].

Types & Errors If the set of constraints is satisfiable, then L IQ -

UID H ASKELL outputs S AFE , meaning the program is verified.

If instead, the set of constraints is not satisfiable, then L IQUID -

H ASKELL outputs U NSAFE , and uses the invalid constraints to

report refinement type errors at the source positions that created

the invalid constraints, using the location information to map the

invalid constraints to source positions. In either case, L IQUID -

H ASKELL produces as output a source map containing the inferred

types for each program expression, which, in our experience, is

crucial for debugging the code and the specifications.

L IQUID H ASKELL is best thought of as an optional type checker

for Haskell. By optional we mean that the refinements have no

influence on the dynamic semantics, which makes it easy to ap-

ply L IQUID H ASKELL to existing libraries. To emphasize the op-

tional nature of refinements and preserve compatibility with ex-

isting compilers, all specifications appear within comments of the

form {-@ ... @-}, which we omit below for brevity.

* Liquid Types

Liquid Haskell uses the Liquid Types [26] framework to infer refinement types,

which greatly reduces the annotation burden for users. The syntax of Liquid Haskell

refinement type annotations is described in Figure 2.1, while simplified inference rules

for subtyping constraint generation is given in Figure 2.2. Liquid Types subtyping constraints have the general form

Γ ` {ν : B | e 1 } <: {ν : B | e 2 }

where Γ is an environment, and e 1 , e 2 are both expressions — either predicative

type variables or formulae from the refinement logic. An environment is a list of

bindings, which have the general form x : {ν : τ | e}, where τ is a base type, and e is

an expression. Here, τ 1 <: τ 2 is the subtyping relation, read “τ 1 is a subtype of τ 2 .”

Liquid types, an abbreviation of Logically Qualified Data Types, were first introduced in [14]. They were presented as a “combination of Hindley-Milner type inference with Predicate Abstraction to automatically infer dependent types precise enough to prove a variety of safety properties”. Behind this definition there are different techniques: • The Hindley-Milner type inference algorithm, usually associated to modern functional languages. This is not strictly essential to the approach. Liquid types could be equally applied to programming languages having a variety of type systems. • Predicate abstraction [5, 15]. This is a technique based on abstract interpretation which searches for the strongest predicate satisfying a set of constraints in a finite complete lattice of predicates related by an entailment relation. This is an essential part of the liquid type approach.

The Liquid type annotations are provided by the programmer in the input file as Haskell comments of the form {-@ annotation @-}. These, of course, are ignored by GHC and are instead processed by LH. As a result, a set of type constraints are generated in the second phase, which are solved in a third phase with the help of a SMT solver, such as Z3 [11] or CVC4 [2]. The input file also contains the set of qualifier fragments from which the inferred liquid types are to be built. Due to a judicious choice of defaults, by which the qualifier fragments are directly extracted from the type annotations, this set is most of the times empty.

Liquid Types is built atop a Hindley-Milner style typing system: after an Hindley-

Milner oracle determines the “shapes” of the types of program expressions, the Liquid

Types constraint solver attempts to find a solution to the fresh predicate variables (the

κ variables) introduced. A solution maps each variable to a conjunction of predicates.

These predicates come from a set of qualifiers — predicate templates that can be

filled in program variables — drawn from our abstract domain. The Liquid Types

solver finds the strongest — most specific — solution for the predicate variables by

starting with the conjunction of every possible instantiated qualifier (those filled in

with variables) and repeatedly weakening the solution until no constraints fail or no

solution is possible. If this is the case, the LiquidHaskell algorithm returns the

Horn queries (as defined in Chapter 5) that were unable to be satisfied.

* Kelemahan Liquid Haskell
* Pena
* The limitations of the Liquid Type approach are those derived of the undecidability of the formula satisfaction problem. If the property being specified needs complex formulas to be proved valid, then the system will give up. For instance, the validity of most universally quantified formulas is undecidable, and these are frequently needed in program verification.
* Vazou
* Our case studies also highlighted several limitations of L IQUID -
* H ASKELL that we will address in future work. In most cases, we
* could alter the code slightly to facilitate verification.
* Ghost parameters are sometimes needed in order to materialize
* values that are not needed for the computation, but are necessary to
* prove various specifications. For example, the piv parameter in the
* append function for red-black trees (§ 6.1).
* Fixed-width integer and floating-point numbers L IQUID H ASKELL
* uses the theories of linear arithmetic and real numbers to reason
* about numeric operations. In some cases this causes us to lose pre-
* cision, e.g. when we have to approximate the behavior of bitwise
* operations. We could address this shortcoming by using the theory
* of bit-vectors to model fixed-width integers, but we are unsure of
* the effect this would have on L IQUID H ASKELL ’s performance.
* Higher-order functions must sometimes be specialized because
* the original type is not precise enough. For example, the concat
* function that concatenates a list of input ByteString s pre-allocates
* the output region by computing the total size of the input.
* len = sum . map length $ xs
* Unfortunately, the type for map is not sufficiently precise to con-
* clude that the value len equals bLens xs , se we must manually
* specialize the above into a single recursive traversal that computes
* the lengths. Rather than complicating the type system with a very
* general higher-order type for map we suspect the best way forward
* will be to allow the user to specify inlining in a clean fashion.
* Functions as Data Several libraries like Text encode data struc-
* tures like (finite) streams using functions, in order to facilitate fu-
* sion. Currently, it is not possible to describe sizes of these structures
* using measures, as this requires describing the sizes of input-output
* chains starting at a given seed input for the function. In future work,
* it will be interesting to extend the measure mechanism to support
* multiple parameters (e.g. a stream and a seed) in order to reason
* about such structures.
* Lazy binders sometimes get in the way of verification. A common
* pattern in Haskell code is to define all local variables in a single
* where clause and use them only in a subset of all branches. L IQ -
* UID H ASKELL flags a few such definitions as unsafe, not realizing
* that the values will only be demanded in a specific branch. Cur-
* rently, we manually transform the code by pushing binders inwards
* to the usage site. This transformation could be easily automated.
* Assumes which can be thought of as “hybrid” run-time checks,
* had to be placed in a couple of cases where the verifier loses
* information. One source is the introduction of assumptions about
* mathematical operators that are currently conservatively modeled
* in the refinement logic (e.g. that multiplication is commutative and
* associative). These may be removed by using more advanced non-
* linear arithmetic decision procedures.
* Error messages are a crucial part of any type-checker. Currently,
* we report error locations in the provided source file and output the failed constraint(s). Unfortunately, the constraints often re-
* fer to intermediate values that have been introduced during the
* ANF-transformation, which obscures their relation to the program
* source. In future work, we may attempt to map these intermedi-
* ate values back to their source expressions, which should increase
* the comprehensibility of our error messages. Another interesting
* possibility would be to search for concrete counterexamples when
* L IQUID H ASKELL detects an invalid constraint.

Liquid haskell Examples

* Vazou Real World Haskell
* We verified totality of two libraries: HsColour and Data.Map,
* earlier versions of which had previously been proven total by
* catch [24].
* Data.Map is a widely used library for (immutable) key-value
* maps, implemented as balanced binary search trees. Totality verifi-
* cation of Data.Map was quite straightforward. We had previously
* verified termination and the crucial binary search invariant [38].
* To verify totality it sufficed to simply re-run verification with the
* --totality argument. All the important specifications were al-
* ready captured by the types, and no additional changes were needed
* to prove totality.
* This case study illustrates an advantage of L IQUID H ASKELL
* over specialized provers (e.g., catch [24]), namely it can be used
* to prove totality, termination and functional correctness at the same
* time, facilitating a nice reuse of specifications for multiple tasks.
* HsColour is a library for generating syntax-highlighted LATEX
* and HTML from Haskell source files. Checking HsColour was
* not so easy, as in some cases assumptions are used about the struc-
* ture of the input data: For example, ACSS.splitSrcAndAnnos
* handles an input list of String s and assumes that whenever a spe-
* cific String (say breakS ) appears then at least two String s (call
* them mname and annots ) follow it in the list. Thus, for a list ls that
* starts with breakS the irrefutable pattern (\_:mname:annots)=
* ls should be total. Currently it is somewhat cumbersome to specify
* such properties, and these are interesting avenues for future work.
* Thus to prove totality, we added a dynamic check that validates that
* the length of the input ls exceeds 2 .
* In other cases assertions were imposed via monadic checks, for
* example HsColour.hs reads the input arguments and checks their
* well-formedness using
* when (length f > 1) $ errorOut "..."
* Currently L IQUID H ASKELL does not support monadic reasoning
* that allows assuming that (length f <= 1) holds when execut-
* ing the action following the when check. Finally, code modifi-
* cations were required to capture properties that currently we do
* not know how to express with L IQUID H ASKELL . For example,
* trimContext checks if there is an element that satisfies p in the
* list xs ; if so it defines ys = dropWhile (not . p)xs and com-
* putes tail ys . By the check we know that ys has at least one
* element, the one that satisfies p , but this is a property that we could
* not express in L IQUID H ASKELL .
* On the whole, while proving totality can be cumbersome (as in
* HsColour) it is a nice side benefit of refinement type checking,
* and can sometimes be a fully automatic corollary of establishing
* more interesting safety properties (as in Data.Map)

# Case Study: Okasaki’s Lazy Queues

Case Study: Associative Maps

Case Study: Pointers & Bytes