Maintaining Verified Software

Joe Leslie-Hurd
Intel Corp.
joe@gilith.com

Abstract

Maintaining software in the face of evolving dependencies is a challenging problem, and in addition to good release practices there is a need for automatic dependency analysis tools to avoid errors creeping in. Verified software reveals more semantic information in the form of mechanized proofs of functional specifications, and this can be used for dependency analysis. In this paper we present a scheme for automatic dependency analysis of verified software, which for each program checks that the collection of installed libraries is sufficient to guarantee its functional correctness. We illustrate the scheme with a case study of Haskell packages verified in higher order logic. The dependency analysis reduces the burden of maintaining verified Haskell packages by automatically computing version ranges for the packages they depend on, such that any combination provides the functionality required for correct operation.

Categories and Subject Descriptors K.6.3 [*Software Management*]: Software Maintenance

Keywords Software Maintenance; Formal Verification; Dependency Analysis

1. Introduction

One feature of a healthy software ecosystem is the ability for developers to build upon the work of others, rather than creating everything from scratch. For example, a developer might link a program to a set of software libraries developed by others that implement useful functionality. However, the price for obtaining this useful functionality is that the correctness of the program now depends on the behaviour of the software libraries, and each new release of the libraries requires careful checking to make sure the program still functions correctly. Checking dependencies is such a common problem that the phrase *'dependency hell'* exists to describe various extreme forms of it.[1]

Software dependency management is an old problem (Lehman 1980), and over time there have emerged release practices and automatic tools to ease the maintenance burden on developers. For example, many software projects have an established practice that each new release is assigned a version number that reflects the scale of the change from the previous release. If the last release of the software was assigned version 1.3, then a new release assigned version 1.4 is expected to behave more like the last release than a new release assigned version 2.0. There is a language-independent *Semantic Versioning Specification* that formalizes this practice by setting conditions for incrementing the different components of a version number,[2] although realizing its promised benefit relies on library developers being willing and able to consistently apply the rules.

In addition to following good release practices, developers can use automatic tools to manage software dependencies. A general observation is that the more information about software behaviour is available to an automatic tool, the more precise it can be about deciding whether a program will function correctly using a particular set of libraries. For example, a linker tool such as `ld` is used to resolve library references in programs, binding function calls in the program to the corresponding object code in a library. However, since the linker only processes program symbols, it can only detect dependency problems where the names of library functions have changed, and is oblivious to changes in their behaviour. By contrast, a compiler tool such as `gcc` is able to detect some dependency problems stemming from changes in software behaviour, since it processes source code containing type information for functions and data.

One way of making more software behaviour information available to automatic tools is to formally verify it. A developer of verified software has the ability to write arbitrary functional specifications for a program, and then construct machine-checkable proofs that the program satisfies its specification. Though real-world verified software is still something of a rarity, developing reusable verified libraries is one way to speed up production of verified applications.

This paper presents a scheme that makes use of the mechanized proofs of verified software to decide which versions of available libraries are required to guarantee functional correctness of a program. It works by analyzing exactly how the environmental assumptions of a verified program are satisfied by the properties of the available libraries, and then using an incremental algorithm to track these dependencies through previous versions of each library until an incompatibility is found.

The scheme for analyzing dependencies relies on the existence of a formal semantic model for the environment of a program and also the code that it implements. Viewed in this way a program P is similar to a logical theory $\Gamma \rhd \Delta$, which uses a set Γ of assumptions to derive a set Δ of theorems. The set Γ of assumptions models the environment used by P: a collection of externally defined symbols with associated properties. The set Δ of theorems models the code implemented in P: a collection of defined symbols with associated properties that are expressed in terms of externally defined sym-

[1] `http://en.wikipedia.org/wiki/Dependency_hell`

[2] `http://semver.org/`

Haskell '13, September 23–24, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2383-3/13/09…$15.00.
http://dx.doi.org/10.1145/2503778.2503787

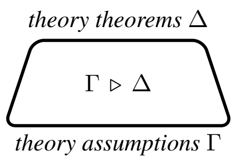

Figure 1. Visual representation of a *theory* $\Gamma \rhd \Delta$ as a labelled box that derives theorems Δ at its top from assumptions Γ at its base.

bols. Associating a verified program P with a logical theory $\Gamma \rhd \Delta$ in this way, it is possible to view the dependency analysis as a *module system* for verified programs.

For example, consider a Haskell program consisting of the following function definition:[3]

```
divides :: Natural -> Natural -> Bool
divides m n =
  if m == 0 then n == 0 else n `mod` m == 0
```

The environment model of this program contains the externally defined type operators $\{\mathsf{Natural}, \mathsf{Bool}, \rightarrow\}$ and constants $\{\mathsf{mod}, =, 0\}$, satisfying a set Γ of their regular mathematical properties.[4] The code model of the program consists of the defined constant $\mathsf{divides}$ satisfying a set Δ of properties such as $\vdash \forall m, n.\ \mathsf{divides}\ n\ (m{*}n)$. Note that the environment and code properties may refer to additional externally defined symbols not directly used in the program (like the $*$ constant in this example), and these are simply folded into the environment model.

The primary contribution of this paper is a language-independent method to reduce the maintenance burden for verified software by automatically computing version ranges of libraries for a program, with the property that selecting any combination of library versions respecting the ranges is sufficient to ensure that the program functions correctly. By modelling verified programs as logical theories this dependency analysis can be carried out by formal reasoning, providing a guarantee of soundness. This precise dependency analysis of verified software is not supported by any existing tool.

A secondary contribution is a concrete implementation of this dependency analysis for Haskell packages verified in higher order logic, both illustrating the approach and demonstrating its feasibility. An advantage of choosing Haskell as the target language for this case study is that it has enough in common with higher order logic that we can perform verifications using a shallow embedding of program behaviour, requiring less infrastructure than a deep embedding of program syntax.

The remainder of this paper is structured as follows: Section 2 presents language-independent techniques for checking dependencies between logical theories; Section 3 describes a form of theory packages that can be exported as verified Haskell packages; Section 4 puts these together to automatically check dependencies between verified Haskell packages; and lastly Sections 5 and 6 survey related work and summarize.

2. Theory Dependencies

Our method for dependency checking of verified software works by modelling verified software as logical theories, and then checking logical dependencies between theories. This section sets this up by focusing on logical theories (without considering their provenance) and presents different techniques for checking dependencies between them.

[3] The Haskell `Natural` type is defined in the `opentheory-primitive` package introduced in Section 3.2.

[4] Typeface is used to distinguish types and values used in Haskell code (e.g., `Natural` and `==`) from their mathematical models (e.g., $\mathsf{Natural}$ and $=$).

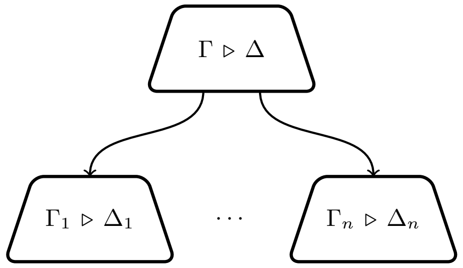

Figure 2. Visual representation of theories $\Gamma_1 \rhd \Delta_1, \ldots, \Gamma_n \rhd \Delta_n$ *required* by theory $\Gamma \rhd \Delta$.

To fix terminology, in this paper a *theory* $\Gamma \rhd \Delta$ consists of:

1. A set Γ of assumptions.
2. A set Δ of theorems.
3. A proof that the theorems in Δ logically derive from the assumptions in Γ.

Figure 1 introduces a visual representation of the theory $\Gamma \rhd \Delta$ as a labelled box that derives theorems Δ at its top from assumptions Γ at its base.

A theory $\Gamma \rhd \Delta$ may define new symbols in its proof, and also refer to external symbols. Given a *context* theorem set Θ we can construct a *binding* σ of external symbols in $\Gamma \rhd \Delta$ to defined symbols in Θ with the same name. Before importing $\Gamma \rhd \Delta$ into the context Θ we construct and apply such a binding σ resulting in $\Gamma[\sigma] \rhd \Delta[\sigma]$. During the import we *satisfy* assumptions in $\Gamma[\sigma]$ with matching theorems in Θ, and if every assumption is satisfied the import is successful and results in the theorem set $\Delta[\sigma]$.

The approach described in this paper is illustrated by experiments carried out using theories of higher order logic (Farmer 2008), where the symbols are either type operators (e.g., $\rightarrow$) or constants (e.g., 0). The theories were extracted from the higher order logic theorem prover in which they were developed and stored as *theory packages* in OpenTheory format (Hurd 2011). The OpenTheory project[5] defines a standard package format which contains a representation of a higher order logic theory in a standard syntax, and also meta-data such as the name, version and author of the theory. The goal of the project is to build up a collection of logical theories that are portable across higher order logic theorem provers, and to this end theory packages in OpenTheory format can be imported into different theorem prover contexts or analyzed directly.

2.1 Theory Dependency Checking

As part of its package meta-data, a theory $\Gamma \rhd \Delta$ can specify a set

$$\{\Gamma_1 \rhd \Delta_1,\ \ldots,\ \Gamma_n \rhd \Delta_n\}$$

of theories that it *requires*, which intuitively means that the theorem sets Δ_i of the required theories should collectively satisfy the assumption set Γ of the theory. Figure 2 gives a visual representation of required theories, and this section will formalize this notion of theory dependency.

An OpenTheory *directory* stores a set of theory packages, which can be thought of as the nodes of a directed *dependency graph*. The arcs of the graph represent the dependency between theories as follows: the graph contains the arc $(\Gamma \rhd \Delta) \rightarrow (\Gamma' \rhd \Delta')$ iff the

[5] `http://www.gilith.com/research/opentheory/`

theory $\Gamma \rhd \Delta$ specifies that it requires theory $\Gamma' \rhd \Delta'$. New versions of theory packages can be added to directories at any time, which can lead to broken dependencies. For example, a new version of a theory might change the set of theorems it proves, resulting in unsatisfied assumptions in theories that require it. Alternatively, a new version of a theory package might require an additional theory, causing a cycle to appear in the dependency graph.

We say that a theory package $\Gamma \rhd \Delta$ is *up-to-date* in a directory if it is possible to prove all of its theorems 'from scratch'. This boils down to the following two conditions:

1. Every theory package $\Gamma_i \rhd \Delta_i$ that theory $\Gamma \rhd \Delta$ requires is up-to-date in the directory and proves the theorem set Θ_i.
2. It is possible to successfully import the theory $\Gamma \rhd \Delta$ into the context $\bigcup_i \Theta_i$, resulting in the theorem set Θ.

For Condition 2 to hold, it must be possible to construct a binding σ of external symbols in $\Gamma \rhd \Delta$ to defined symbols in $\bigcup_i \Theta_i$ with the same name. The only way this can fail is if there are defined symbols in Θ_i and Θ_j with the same name and different definitions. Enforcing consistency of definitions in contexts is good practice anyway, since it rules out soundness bugs such as (i) define $\vdash \mathsf{c} = 0$; (ii) define $\vdash \mathsf{c} = 1$; (iii) prove $\vdash 0 = 1$. Once the binding σ has been successfully constructed, the import will be successful if every assumption in $\Gamma[\sigma]$ is satisfied by a matching theorem in $\bigcup_i \Theta_i$.

Note that Condition 1 implies that there are no cycles in the dependency graph reachable from up-to-date theory packages. We can thus automatically check whether theories are up to date in a directory by treating the above conditions as a recursive algorithm that explicitly computes the theorem sets for every up-to-date package (reporting *"not up-to-date"* if a cycle is detected). In addition, if the algorithm is implemented by a theorem prover in the LCF design (Gordon et al. 1979), the soundness guarantee of the logical kernel provides high assurance that the up-to-date check will have no false positives.

2.2 Local Dependency Checking

As the previous section shows, checking that a theory $\Gamma \rhd \Delta$ is up-to-date, in the sense that all its theorems are provable 'from scratch', is a property of the whole dependency graph reachable from $\Gamma \rhd \Delta$. However, during theory development it is also interesting to know whether $\Gamma \rhd \Delta$ contains any inherent problems that would prevent it from being up-to-date in *any possible* dependency graph. We call this *local dependency checking*, and it consists of checking four properties that are necessary for theories to be up-to-date.

The first local dependency check examines a single theory in isolation.

Check 1: No Axioms A theory $\Gamma \rhd \Delta$ may define new symbols and prove properties of them, so it is common for defined symbols to appear in the theorem set Δ. However, assumptions in Γ that contain defined symbols can never be satisfied by required theories; we call such assumptions *axioms* of the theory. This check flags all axioms as dependency problems.

The remaining local dependency checks examine a single theory $\Gamma \rhd \Delta$ in the context of the theorem sets Δ_i of the theories that $\Gamma \rhd \Delta$ requires.

Check 2: Definitional Consistency This check flags all instances of defined symbols with the same name that come from different theorem sets Δ_i and Δ_j, since multiple definitions of the same symbol can result in an inconsistent context.

Check 3: No Unsatisfied Assumptions Assuming the theorem sets Δ_i satisfy definitional consistency, we can construct a binding σ of external symbols in $\Gamma \rhd \Delta$ to defined symbols in Δ_i with the same name. This check flags any unsatisfied assumptions in $\Gamma[\sigma]$ that are not matched by any theorems in $\bigcup_i \Delta_i[\sigma]$. Note that we need to apply the binding σ to the theorem sets Δ_i, since they may also contain instances of external symbols. Note also that this check subsumes *Check 1: No Axioms*, but we keep both because the special case is a useful check that is independent of other theories.

Check 4: No Ungrounded Symbols This final check flags all instances of ungrounded symbols in the theorem set Δ. An *ungrounded* symbol is an external symbol that does not appear in Γ or any of the Δ_i, and thus will not be bound to a defined symbol during import. This causes problems because, although this theory import might succeed, the resulting theorem set will contain an external symbol that fails to match assumptions made by later theories that require $\Gamma \rhd \Delta$. Ungrounded symbols are a relatively rare problem in practice, but can appear in a theory such as

$$\Delta \equiv \{ \vdash \forall m, n.\ \mathsf{divides}\ m\ n \iff \exists k.\ k * m = n \}$$

which defines the divides constant and then derives a single theorem from the raw definition. The theory requires only Boolean theory to satisfy all the assumptions made in its short proof, but this will result in the $*$ constant being an ungrounded symbol. The solution is for the theory to require both Boolean and natural number theory; this will not only satisfy all assumptions but will also ground $*$ to a defined constant.

Local dependency checking offers the theory developer a fast way to discover dependency problems in a theory independent of a particular dependency graph. In addition, there is a way to efficiently discover which versions of its required theories allow a theory to pass local dependency checking, which we describe in the next section.

2.3 Incremental Dependency Checking

Suppose we are given a theory $\Gamma \rhd \Delta$ that passes local dependency checking for its required theories. It is natural to ask whether we could replace one of the required theories with another version of the same theory and still pass local dependency checking. The general problem is to find version ranges for each required theory, such that theory $\Gamma \rhd \Delta$ passes local dependency checking for any combination of required theory versions respecting the ranges. A naive way to compute these version ranges is to add required theory versions one at a time and carry out local dependency checking for all new combinations, but this would result in an exponential algorithm. In this section we present an efficient solution to this problem in the form of an incremental algorithm.

The input to the algorithm is a theory $\Gamma \rhd \Delta$ that requires theories with names n_i, and for now we fix on a particular version of each required theory. We will illustrate the concepts using the `natural-divides` theory, which defines the divides relation on natural numbers (the definition appears in the previous section) and proves properties that follow from the definition. The `natural-divides` theory requires theories with names `bool` (Boolean) and `natural` (natural number), and we fix on their latest versions (1.29 and 1.78 respectively). We create finite map data structures to represent the following three functions:

$$\begin{aligned}
\mathsf{defines} &: \text{symbol names} \to \text{theory name sets} \\
&\quad s \mapsto \{n_i \mid \text{required theory } n_i \text{ defines symbol } s\} \\
\mathsf{satisfies} &: \text{assumptions} \to \text{theory name sets} \\
&\quad \gamma \mapsto \{n_i \mid \text{required theory } n_i \text{ satisfies assumption } \gamma\} \\
\mathsf{grounds} &: \text{symbol names} \to \text{theory name sets} \\
&\quad s \mapsto \{n_i \mid \text{required theory } n_i \text{ grounds symbol } s\}
\end{aligned}$$

For the `natural-divides` theory, the following are true:

- $\mathsf{defines}(\wedge) = \{\texttt{bool}\}$
  $\mathsf{defines}(*) = \{\texttt{natural}\}$
  $\mathsf{defines}(\mathsf{divides}) = \{\}$
- $\mathsf{satisfies}(\vdash\ \top) = \{\texttt{bool}\}$
  $\mathsf{satisfies}(\vdash\ \forall n.\, n \leq n) = \{\texttt{natural}\}$
- $\mathsf{grounds}(\wedge) = \{\texttt{bool}, \texttt{natural}\}$
  $\mathsf{grounds}(*) = \{\texttt{natural}\}$

The local dependency checks can be expressed as properties of these functions:

1. *No Axioms* is a special case of *No Unsatisfied Assumptions*.
2. *Definitional Consistency* $\iff$ For every symbol s, the set $\mathsf{defines}(s)$ must contain at most one required theory name.
3. *No Unsatisfied Assumptions* $\iff$ For every assumption $\gamma \in \Gamma$, the set $\mathsf{satisfies}(\gamma)$ must contain at least one required theory name.
4. *No Ungrounded Symbols* $\iff$ For every external symbol s that appears in Δ but does not appear in Γ, the set $\mathsf{grounds}(s)$ must contain at least one required theory name.

Note that each of these properties can be efficiently computed over the finite map representation of the functions.

Up to this point the introduction of the above functions and properties might just seem like a complicated way of performing local dependency checking for a theory. The benefit of expressing local dependency checking in this way is that the triple $(\mathsf{defines}, \mathsf{satisfies}, \mathsf{grounds})$ is capable of representing dependency information for a version set of each required theory, such that the above properties hold iff local dependency checking would pass regardless which version of each required theory was selected. The key to making this work is an $\otimes$ operation that combines triples like so:

$$\begin{pmatrix} \mathsf{defines}, \\ \mathsf{satisfies}, \\ \mathsf{grounds} \end{pmatrix} \otimes \begin{pmatrix} \mathsf{defines}', \\ \mathsf{satisfies}', \\ \mathsf{grounds}' \end{pmatrix}$$
$$= \begin{pmatrix} \lambda s.\ \mathsf{defines}(s) \cup \mathsf{defines}'(s), \\ \lambda \gamma.\ \mathsf{satisfies}(\gamma) \cap \mathsf{satisfies}'(\gamma), \\ \lambda s.\ \mathsf{grounds}(s) \cap \mathsf{grounds}'(s) \end{pmatrix}$$

The intuition behind this definition of the $\otimes$ operation is as follows. *Definitional Consistency* holds if there is no symbol overlap between the definitions made by required theories, so once some version of required theory n_i has defined the symbol s we consider s to be owned by n_i, and no version of another required theory n_j is permitted to define s. *No Unsatisfied Assumptions* is more complicated, because for a given assumption γ a required theory might satisfy in some versions and not in others. However, it must be the case that there is some required theory that satisfies γ in every version. If there was not then we could select a version of each required theory that did not satisfy γ, and it would remain unsatisfied by the combination. *No Ungrounded Symbols* follows the same reasoning as *No Unsatisfied Assumptions*.

We now have all the tools we need to present the incremental algorithm for local dependency checking:

1. Take as input a theory $\Gamma \rhd \Delta$ that requires theories with theory names n_i.
2. Create new $\mathsf{defines}$, $\mathsf{satisfies}$ and $\mathsf{grounds}$ functions from the latest versions of the required theories. Check the local dependency properties of $\mathsf{defines}$, $\mathsf{satisfies}$ and $\mathsf{grounds}$, and if any fail then exit with the error *"not up-to-date"*.
3. Pick an arbitrary required theory n_j where there is a previous untested version available. If there is no such n_j then go to Step 6.
4. Temporarily replace required theory n_j with its previous version. Create new $\mathsf{defines}'$, $\mathsf{satisfies}'$ and $\mathsf{grounds}'$ functions from the required theories, and temporarily replace
$$\begin{pmatrix} \mathsf{defines}, \\ \mathsf{satisfies}, \\ \mathsf{grounds} \end{pmatrix} := \begin{pmatrix} \mathsf{defines}, \\ \mathsf{satisfies}, \\ \mathsf{grounds} \end{pmatrix} \otimes \begin{pmatrix} \mathsf{defines}', \\ \mathsf{satisfies}', \\ \mathsf{grounds}' \end{pmatrix}$$
5. Check the local dependency properties of $\mathsf{defines}$, $\mathsf{satisfies}$ and $\mathsf{grounds}$. If they all hold then commit the temporary changes, and if any fail then roll back the temporary changes. Go back to Step 3.
6. For each required theory n_i, output the up-to-date range to be everything between the current version and the latest version.

Running this algorithm on the `natural-divides` theory generates the following version range:

$$\begin{array}{r} \texttt{bool}\ \geq 1.25\ \wedge\ \leq 1.29 \\ \texttt{natural}\ \geq 1.55\ \wedge\ \leq 1.78 \end{array}$$

This syntax means that combining any `bool` theory package in the version range 1.25–1.29 with any `natural` theory package in the version range 1.55–1.78 will be sufficient to pass local dependency checking for the `natural-divides` theory package.

The algorithm will always terminate, because every selection of n_j in Step 3 will result in either moving to a previous version of n_j (if the check in Step 5 succeeds) or removing n_j from further consideration (if the check fails).

In general, the problem of finding maximal version ranges of required theories that satisfy local dependency checking does not have a unique solution, and when there are multiple solutions the particular version range generated by the algorithm will depend on the theory choices made in Step 3. For example, suppose the `natural-divides` theory made an assumption that was satisfied by a theorem proved only in the latest version of required theories `bool` and `natural`. The generated version range cannot contain prior versions of both `bool` and `natural` (selecting prior versions of both would leave the assumption unsatisfied), but whether the version range contains prior versions of `bool` or prior versions of `natural` depends on which required theory was considered first in Step 3.

3. Verified Haskell Packages

Haskell (O'Sullivan et al. 2008) is a functional programming language that is rapidly growing in popularity, in part due to the Cabal package system that simplifies the process of reusing code libraries developed by other Haskellers (Coutts et al. 2008). For our case study of verified software dependencies we will target Cabal packages containing Haskell code verified in higher order logic.

It is well-known that there is a correspondence between a pure subset of Haskell and a subset of higher order logic (Haftmann 2010). For our case study we choose to develop the code in a higher order logic theorem prover (Section 3.1) and then generate Haskell from theory packages (Section 3.2).

3.1 Theories of Haskell Packages

The OpenTheory `haskell-prime` package uses the sieve of Eratosthenes to compute the prime numbers, and we will use this

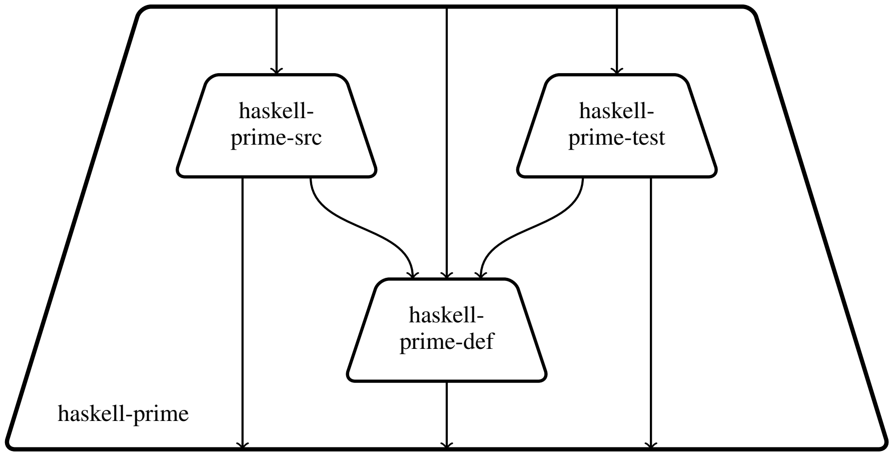

Figure 3. Theory structure of the OpenTheory `haskell-prime` package.

theory as a running example to illustrate the method of exporting OpenTheory packages as Haskell packages.[6]

Figure 3 shows the internal structure of the `haskell-prime` theory package (recall that theories are represented by labelled boxes that derive theorems at their top from assumptions at their base). The top-level `haskell-prime` theory package includes three nested theory packages, which have dependencies both between themselves and between the assumptions and theorems of `haskell-prime`. For example, the two arrows down from `haskell-prime-src` show that it satisfies its assumptions using the theorem set of `haskell-prime-def` and the theorem sets of the packages that `haskell-prime` depends on (not shown). The arrows down from the top of `haskell-prime` show that its theorem set consists of the union of the theorem sets of its included theories.

A Haskell package defines new programming objects and exports names for referring to them. In the subset of Haskell that is the target of our case study, the new programming objects are types (algebraic datatypes and newtypes) and values (functions and data). The close correspondence between our Haskell subset and higher order logic allows us to use a shallow embedding to model types with types and values with terms, but this is not essential. For a different target language we might model all programming objects with terms (e.g., using a deep embedding). There is only one feature of the model that is essential for our dependency analysis to succeed: exported names in the software must be formalized by defining new symbols in the theory. In our example the `haskell-prime-def` package formalizes exported Haskell names by defining new type operators and constants in terms of symbols from mathematics and other verified Haskell packages. For example, the definition of the stream (i.e., infinite list) of prime numbers is

$$\vdash \mathsf{Haskell.Prime.all} = \mathsf{Prime.all}$$

which just defines a new constant in the Haskell namespace that is an alias for the mathematical definition.

We now consider the `haskell-prime-src` package, which uses the definitions made in the `haskell-prime-def` package to derive the 'computational forms' that can appear in Haskell source files. For example, the computational form for the stream of prime numbers is the theorem

$$\begin{array}{ll}\vdash & \mathsf{Haskell.Prime.all} = \\ & \mathsf{Haskell.Stream.unfold\ Haskell.Prime.next} \\ & \quad \mathsf{Haskell.Prime.initial}\end{array}$$

which 'unfolds' a stream from an initial state and a next state function, and looks much more like a Haskell value declaration than its higher order logic definition that we saw before. Rather than assigning a mathematical meaning to a description of a computation, here we are deriving a description of a computation from a mathematical definition. An advantage of this way of describing computations is that it side-steps the thorny problem of defining functions by general recursion in higher order logic at the same time as proving that they are total (Slind 1999). In this scheme we can define a function in whichever form is most convenient, and then derive the recursive equations as a logical consequence.

The final nested package in our example is `haskell-prime-test`, which contains a collection of executable properties we expect the Haskell sources to satisfy. Executable properties must either be an *assertion* that is quantifier free, or a *proposition* with a single universally quantified variable of type random: these can be automatically tested using the Haskell's QuickCheck (Claessen and Hughes 2000). Both assertions and propositions are used to check that the export step has been carried out correctly. The `haskell-prime-test` package contains one assertion and three propositions. The assertion is

$$\vdash \mathsf{Haskell.Stream.nth\ Haskell.Prime.all}\ 0 \neq 0$$

which ensures the first element in the stream of prime numbers is not zero. The first of the three propositions is

$$\begin{array}{ll}\vdash & \forall r. \\ & \quad \mathsf{let}\ (i, r') \leftarrow \mathsf{Haskell.Natural.fromRandom}\ r\ \mathsf{in} \\ & \quad \mathsf{let}\ (j, r'') \leftarrow \mathsf{Haskell.Natural.fromRandom}\ r'\ \mathsf{in} \\ & \quad (\mathsf{Haskell.Stream.nth\ Haskell.Prime.all}\ i \leq \\ & \quad\ \ \mathsf{Haskell.Stream.nth\ Haskell.Prime.all}\ j) \iff i \leq j\end{array}$$

which ensures the stream of prime numbers is strictly increasing. Note the use of fromRandom conversion functions to manufacture values of arbitrary types from the single permitted quantified variable of type random.

3.2 Exporting Theories as Haskell Packages

The previous section described how to construct an OpenTheory package in a form that is ready for export as a Haskell pack-

[6] It is perhaps confusing that the names of OpenTheory packages and symbols that correspond to Haskell entities are identified by a "Haskell" prefix (and vice versa).

age. The export procedure is completely automatic: the command `opentheory export haskell-prime` exports the example `haskell-prime` OpenTheory package to a Haskell package called `opentheory-prime`.[7] In this section we continue with the `haskell-prime` example to illustrate how the export procedure generates the contents of a Haskell package: source code; a test suite; and package meta-data.

The Haskell source code is generated from the theorem set of the included `haskell-prime-src` theory package. There are three different standard forms of theorems that generate datatype, newtype and value declarations. For example, the Haskell value declaration that is generated for the stream of prime numbers in the `OpenTheory.Prime` module is

```
all :: [OpenTheory.Primitive.Natural.Natural]
all = OpenTheory.Stream.unfold next initial
```

As this example shows, some higher order logic type operators and constants are mapped to primitive Haskell types and values, and the rest are mapped by simply renaming them (the Haskell namespace is replaced with OpenTheory). The primitive Haskell types and values provide the computational platform for all verified Haskell packages, and its implementation is split between standard Haskell (e.g., lists) and a special `opentheory-primitive` package (e.g., natural numbers).

The mapping to primitive Haskell types and values does not have to be reversible, and indeed we map both finite and infinite list types in higher order logic to standard Haskell list types. Though there is potential for clients of a verified Haskell package to be confused about whether a function expects finite or infinite lists as its argument, other verified Haskell packages that depend on it are checked in higher order logic where using the wrong kind of list is a type error.

Our approach of interpreting equality theorems as Haskell equations offers no guarantees of termination (we could just have easily exported the equation `all = all` above), so the generated Haskell packages are only verified to be partially correct despite higher order logic being a logic of total functions. A nice feature of Haskell code is that all declarations are interpreted as being mutually recursive, so we do not need any special handling for generating these computational forms. Appendix A lists the full Haskell source code generated from the `haskell-prime-src` theory package.

The theorem set of the nested `haskell-prime-test` theory package is used to generate a QuickCheck test suite for the Haskell package: assertions and propositions are exported as value declarations together with a `main` function that tests them in sequence. For example, here is the Haskell value declaration for the proposition that the stream of prime numbers is strictly increasing:

```
proposition0 ::
  OpenTheory.Primitive.Random.Random -> Bool
proposition0 r =
  let (i,r') = OpenTheory.Natural.fromRandom r in
  let (j,_) = OpenTheory.Natural.fromRandom r' in
  (OpenTheory.Stream.nth OpenTheory.Prime.all i <=
   OpenTheory.Stream.nth OpenTheory.Prime.all j) ==
  (i <= j)
```

This proposition can be tested using QuickCheck, which checks that calling the `proposition0` function with randomly generated values of the `Random` type always returns `True`. It might seem strange for a Haskell package to contain a test suite of formally verified properties, but exporting OpenTheory packages as Haskell packages is an informal operation, and the test suite provides an extra check that the target has the expected behaviour.

[7] The `opentheory` tool that implements this export procedure is available for download at `http://www.gilith.com/software/opentheory/`

One limitation of this approach is that every assertion and proposition in the test suite must be expressed as executable code. However, many interesting properties fall into this subset: the expressive power of higher order logic even supports a proof of the meta-property that the stream of prime numbers is the unique stream that will always pass the assertion and three proposition tests exported from the `haskell-prime-test` package. Appendix B lists the full QuickCheck test suite generated from the `haskell-prime-test` theory package.

The last element of a verified Haskell package is the meta-data that contains the package information and instructions for building the library and test-suite. The package information comes directly from the meta-data of the OpenTheory package, either copied exactly (like package version and author) or modified according to a fixed scheme (the `haskell` prefix in the package name is replaced with `opentheory`). The build instructions are a combination of standard boilerplate that is the same for all verified Haskell packages (such as `ghc-options`), and information derived from the export (such as `exposed-modules`). The only exception is the list of build dependencies on other Haskell packages—generating this information will be covered in the next section.

4. Verified Haskell Package Dependencies

The Hackage repository is the central hub for Haskell developers to upload their packages for others to use, and at the time of writing it contains 5,260 unique packages spanning 31,099 package versions.[8] A combination of Haskell language features and the Cabal package system has made it particularly easy to reuse code from the Hackage repository, but this has creating difficulties with managing the dependencies of large Haskell projects. The previous sections showed how to check dependencies between logical theories, and how to generate verified Haskell packages from OpenTheory packages. We now put these together to automatically generate dependencies between verified Haskell packages.

4.1 Checking Package Build Dependencies

Section 3 described a form of OpenTheory packages that can be exported as verified Haskell packages. We call these exportable OpenTheory packages *Haskell theories*, and by convention their names have the prefix `haskell`. Like any OpenTheory package, a Haskell theory $\Gamma \rhd \Delta$ specifies a list of required theory packages, some of which are Haskell theories and the rest provide mathematical support for the verification. The required Haskell theories represent build dependencies of the verified Haskell package generated from $\Gamma \rhd \Delta$. We can thus use our theory dependency checking techniques from Section 2 to manage verified Haskell package dependencies.

To check the build dependencies between a collection of verified Haskell packages, one possible scheme is to maintain a parallel OpenTheory directory alongside a Cabal directory of Haskell packages. Whenever a verified Haskell package was installed or upgraded in the Cabal directory, the corresponding OpenTheory package would be installed or upgraded in the OpenTheory directory. We can ensure that there are no broken dependencies between the current versions of the verified Haskell packages by checking that the corresponding theories in the OpenTheory directory are up-to-date (in the sense of Section 2.1).

In fact this scheme works for maintaining the build dependencies between versions of verified software in any language, because all the checking is carried out by OpenTheory at install time using the corresponding theory packages. However, in the case of verified Haskell packages there is a way to ensure correct build dependencies using the standard Cabal build infrastructure, avoiding the need

[8] `http://hackage.haskell.org/`

for users of verified Haskell packages to process their corresponding theories.

The Cabal tool is used to build Haskell packages that depend on specified version ranges of other packages. Given a top-level package version as build target, it employs a solver to construct a set S of package versions that satisfy the following conditions:

- **Targeted:** The build target should be in S.
- **Closed:** Every package version dependency in S is satisfied by exactly one package version in S.
- **Acyclic:** There are no cycles in the package version dependency graph for S.
- **Consistency:** There are no name clashes between symbols exported by package versions in S.

This is not a complete list of conditions on Cabal package version sets (e.g., S should be minimal), but it is sufficient for our purposes.

Given a set of verified Haskell package versions, the above Cabal requirements ensure that the corresponding theories will be up-to-date, *so long as every assumption made by a package version is satisfied by a theorem in a dependent package version.* Therefore, every verified Haskell package must specify version ranges for dependent packages such that every combination guarantees this extra condition. This is precisely the notion of local dependency checking from Section 2.2, and so we can use the incremental algorithm described in Section 2.3 to generate version ranges for required Haskell packages.

This scheme allows the developer of verified Haskell packages to automatically generate version ranges of dependent packages at release time, and the user of the packages will be guaranteed correct behaviour by simply building them using the standard Cabal infrastructure. In addition, this scheme reduces the effort required for the developer to maintain verified Haskell packages. As new releases of dependent packages are made, it is trivial to bump the version, recompute the dependency analysis and make a new release with updated version ranges (this could even be carried out by the repository either completely automatically or with minimal human involvement).

4.2 Verified Haskell Package Examples

To test our approach to dependency analysis, we created a toy library of verified Haskell packages generated from OpenTheory packages extracted from the HOL Light theorem prover (Harrison 1996).[9] The packages and their dependencies are illustrated in Figure 4: an arrow from package H to package H' means that the list of build dependencies for H contains H'. The dotted lines indicate the manual set-up of the computational platform, and the solid lines indicate verified Haskell packages and build dependencies automatically generated from OpenTheory packages.

The manual set-up, including the implementation of primitives and the random testing using QuickCheck, is described in Section 3.2. These Haskell packages are added to the build dependencies of every verified Haskell package.

The `opentheory` Haskell package implements a library of basic utility functions in terms of the primitives, but does so as a verified Haskell package that is automatically generated from the `haskell` OpenTheory package. This design helps to keep the manually written code small and static, and allows us to use the standard dependency analysis to discover which versions of the `opentheory` package are required by other verified Haskell packages.

The `opentheory-prime` package is the result of exporting the `haskell-prime` OpenTheory package. To close the running example, here are the automatically generated build dependencies that are inserted in the Haskell package meta-data:

```
build-depends:
  base >= 4.0 && < 5.0,
  random >= 1.0.1.1 && < 2.0,
  QuickCheck >= 2.4.0.1 && < 3.0,
  opentheory-primitive >= 1.0 && < 2.0,
  opentheory >= 1.73 && <= 1.74
```

This `build-depends` meta-data uses similar syntax to the example at the end of Section 2.3, and lists the acceptable version ranges of Haskell packages that the `opentheory-prime` package depends on. As a result of inserting this meta-data the Haskell package manager will only attempt to build the `opentheory-prime` package with dependent verified packages that pass local dependency checking.

The `opentheory-parser` and `opentheory-char` packages are more realistic examples with non-trivial correctness proofs. The `opentheory-parser` package implements a simple set of parser combinators operating on polymorphic streams; the `opentheory-char` package defines a Haskell representation of Unicode characters and uses the parser combinators to implement the UTF-8 encoding of Unicode characters as byte streams.

The approach taken to develop all these verified Haskell packages was first to prototype an unverified version in Haskell, and once that was passing some basic tests to port the implementation to higher order logic and prove that it satisfied Unicode and UTF-8 expected properties. Finally the verified Haskell packages were generated, together with a test suite of executable properties such as *round-trip*: UTF-8 encoding followed by decoding returns the original string of Unicode characters.

The effort of maintaining even this toy collection of verified Haskell packages is reduced using the scheme presented in Section 4.1. All theories are free to evolve as they will, and each time a new release is made the Haskell packages are regenerated together with version ranges of dependent packages that guarantee their verified properties. Finding the widest version ranges manually would be tedious work, but the alternative of artificially narrowing them would unnecessarily constrain users of the verified Haskell packages.

5. Related Work

The Cabal/Hackage system (Coutts et al. 2008) for distributing Haskell packages is integrated with the GHC compiler, which checks dependencies between packages as part of building them. The Haskell type system can express many useful properties of programs, so type checking a package with respect to its build dependencies has much in common with theory dependency checking. However, the precise properties required of dependent packages are limited by the expressivity of the type system and thus can't be used to search for compatible versions of packages. Much still relies on developers following good release practices such as the Cabal *Package Versioning Policy*.[10]

Another approach to specifying behaviour more precisely is to use a module system. The signature of a higher order logic theory $\Gamma \rhd \Delta$ mapping one set of type operators and constants with properties Γ to another set with properties Δ is similar in concept to a functor in the Standard ML module system (Milner et al. 1997). However, like the expressive Haskell type system and unlike logical theories, module systems are designed to help programmers avoid bugs and not to capture all the interesting behaviour of a module. This means that they can only be used to show that a version of

[9] The HOL Light source files used to generate these OpenTheory packages are available at `https://github.com/gilith/hol-light/`

[10] `http://www.haskell.org/haskellwiki/Package_versioning_policy`

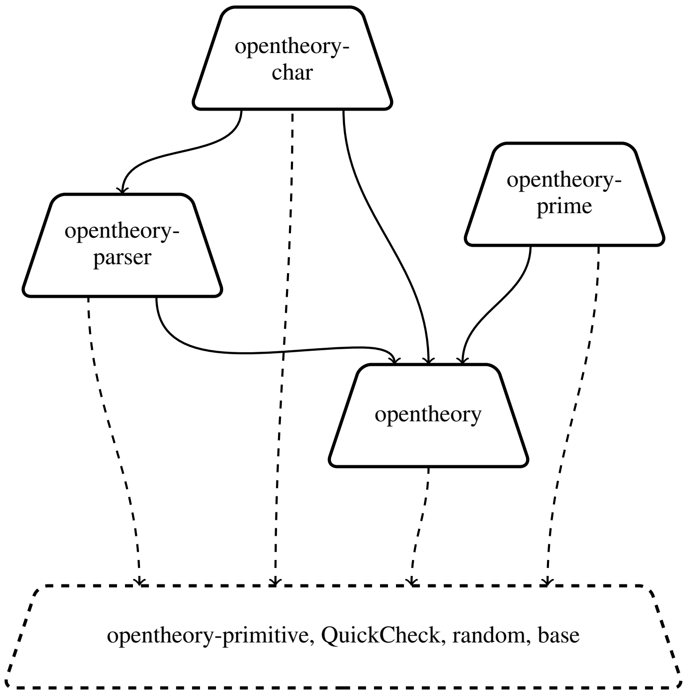

Figure 4. Build dependencies between the Haskell computational platform and a collection of verified packages.

a dependent package is incompatible with the current module, and never to show that a collection of dependent package versions are sufficient to enable the desired interesting behaviour.

Many operating system distributions rely on package managers to keep track of the logical dependencies between different versions of applications and ensure consistency of the platform. The Nix package manager (Dolstra and Löh 2008) adopts a purely functional approach to package management, supporting rollbacks and multiple installed versions of packages. Although highly scalable and robust, most package managers are not equipped to automatically check dependencies between packages. It would be interesting to integrate the techniques in this paper into a package manager to create a high assurance distribution mechanism for verified software.

There has been a great deal of research into generating code using higher order logic theories; the Haskell case study in this paper is most similar to the work of Haftmann converting specifications formulated in Isabelle's higher-order logic into executable Haskell code (Haftmann 2010). The present work differs from previous efforts by its focus on modelling the logical dependencies between units of generated code, so that the build dependencies can be automatically generated in addition to the code. To improve the fidelity and increase the scope of our Haskell theories, we could use Huffman's formalization of Haskell (Huffman 2012), which can distinguishes between strict and lazy code, and also includes the program structuring constructs of monads and transformers.

6. Summary

This paper presented a scheme for analyzing the dependencies of verified software and automatically generating library version ranges that guarantee functional correctness. The key idea underlying our approach is that *we can perform verified software dependency checking by formal reasoning on logical theories.* The aim of this research is to reduce the maintenance burden on verified software without stifling its evolution.

The dependency analysis is illustrated by an in-depth case study of verified Haskell packages generated by higher order logic theory packages. A by-product of this work is the beginning of a verified Haskell package library, which can be extended using the recipe of the `opentheory-char` package for creating verified Haskell packages from unverified prototypes. All of the verified Haskell packages in this paper have been uploaded to the Hackage repository, and there is now an opportunity to build on the verified Unicode package to develop a suite of verified text-processing tools (from `wc` to `grep`).

An interesting avenue of future work is to expand the use of verified properties of Haskell programs beyond their inclusion in a package test suite. Perhaps a Haskell compiler could perform better optimizations if it knew extra properties of the code such as the possible range of a numeric variable or that a user-defined function is commutative.

Acknowledgements

This paper was greatly improved by comments from Flemming Andersen, Iavor Diatchki, Rebekah Leslie-Hurd, Lee Pike and the anonymous referees.

A. Generated Haskell Code Example

This appendix lists Haskell source code implementing the sieve of Eratosthenes to compute prime numbers. The code was developed as logical definitions in the HOL Light theorem prover, extracted into the OpenTheory `haskell-prime` package, and automatically exported to the Haskell `opentheory-prime` package.[11]

A.1 The Infinite List of Primes

```
module OpenTheory.Number.Natural.Prime
where
```

[11] `http://hackage.haskell.org/package/opentheory-prime`

```
import qualified OpenTheory.Data.Stream
  as Data.Stream
import qualified OpenTheory.Number.Natural.Prime.Sieve
  as Sieve
import qualified OpenTheory.Primitive.Natural
  as Primitive.Natural

all :: [Primitive.Natural.Natural]
all = Data.Stream.unfold Sieve.next Sieve.initial
```

A.2 The Underlying Sieve Technique

```
module OpenTheory.Number.Natural.Prime.Sieve
where

import qualified OpenTheory.Primitive.Natural
  as Primitive.Natural

newtype Sieve =
  Sieve {
    unSieve ::
      (Primitive.Natural.Natural,
       [(Primitive.Natural.Natural,
         (Primitive.Natural.Natural,
          Primitive.Natural.Natural))])
  }

initial :: Sieve
initial = Sieve (1, [])

increment :: Sieve -> (Bool, Sieve)
increment =
  \s ->
    let (n, ps) = unSieve s in
    let n' = n + 1 in
    let (b, ps') = inc n' 1 ps in
    (b, Sieve (n', ps'))
  where
    inc n _ [] = (True, (n, (0, 0)) : [])
    inc n i ((p, (k, j)) : ps) =
      let k' = (k + i) `mod` p in
      let j' = j + i in
      if k' == 0 then (False, (p, (0, j')) : ps)
      else let (b, ps') = inc n j' ps in
           (b, (p, (k', 0)) : ps')

perimeter :: Sieve -> Primitive.Natural.Natural
perimeter s = fst (unSieve s)

next :: Sieve -> (Primitive.Natural.Natural, Sieve)
next s =
  let (b, s') = increment s in
  if b then (perimeter s', s') else next s'
```

B. Generated QuickCheck Test Example

This appendix lists the QuickCheck tests contained in the Haskell `opentheory-prime` package. Like the Haskell source code, this test script was automatically generated from the OpenTheory `haskell-prime` package.

```
module Main
  ( main )
where

import qualified OpenTheory.Data.Stream
  as Data.Stream
import qualified OpenTheory.Number.Natural
  as Number.Natural
import qualified OpenTheory.Number.Natural.Geometric
  as Number.Natural.Geometric
import qualified OpenTheory.Number.Natural.Prime
  as Number.Natural.Prime
import qualified OpenTheory.Primitive.Random
  as Primitive.Random
import qualified OpenTheory.Primitive.Test
  as Primitive.Test

assertion0 :: Bool
assertion0 =
  not (Data.Stream.nth Number.Natural.Prime.all 0 == 0)

proposition0 :: Primitive.Random.Random -> Bool
proposition0 r =
  let (i, r') = Number.Natural.Geometric.fromRandom r in
  let (j, _) = Number.Natural.Geometric.fromRandom r' in
  (Data.Stream.nth Number.Natural.Prime.all i <=
   Data.Stream.nth Number.Natural.Prime.all j) ==
  (i <= j)

proposition1 :: Primitive.Random.Random -> Bool
proposition1 r =
  let (i, r') = Number.Natural.Geometric.fromRandom r in
  let (j, _) = Number.Natural.Geometric.fromRandom r' in
  not
    (Number.Natural.divides
       (Data.Stream.nth Number.Natural.Prime.all i)
       (Data.Stream.nth Number.Natural.Prime.all
          (i + j + 1)))

proposition2 :: Primitive.Random.Random -> Bool
proposition2 r =
  let (n, r') = Number.Natural.fromRandom r in
  let (i, _) = Number.Natural.Geometric.fromRandom r' in
  any (\p -> Number.Natural.divides p (n + 2))
    (Data.Stream.take' Number.Natural.Prime.all i) ||
  Data.Stream.nth Number.Natural.Prime.all i <= n + 2

main :: IO ()
main =
    do Primitive.Test.assert "Assertion 0" assertion0
       Primitive.Test.check "Proposition 0" proposition0
       Primitive.Test.check "Proposition 1" proposition1
       Primitive.Test.check "Proposition 2" proposition2
       return ()
```

References

K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, Sept. 2000. URL http://www.md.chalmers.se/~rjmh/QuickCheck/.

D. Coutts, I. Potoczny-Jones, and D. Stewart. Haskell: Batteries included. In A. Gill, editor, *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 125–126. ACM, Sept. 2008. URL http://www.cse.unsw.edu.au/~dons/papers/CPJS08.html.

E. Dolstra and A. Löh. NixOS: A purely functional Linux distribution. In J. Hook and P. Thiemann, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Functional programming (ICFP 2008)*, pages 367–378. ACM, Sept. 2008. URL http://doi.acm.org/10.1145/1411204.1411255.

W. M. Farmer. The seven virtues of simple type theory. *Journal of Applied Logic*, 6:267–286, 2008. URL http://imps.mcmaster.ca/wmfarmer/publications.html.

M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

F. Haftmann. From higher-order logic to Haskell: There and back again. In J. P. Gallagher and J. Voigtländer, editors, *Proceedings of the ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation (PEPM 2010)*, pages 155–158. ACM, Jan. 2010. URL http://www4.in.tum.de/~haftmann/pdf/from_hol_to_haskell_haftmann.pdf.

J. Harrison. HOL light: A tutorial introduction. In M. Srivas and A. Camilleri, editors, *Proceedings of the First International Confer-*

ence on Formal Methods in Computer-Aided Design (FMCAD '96), volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996. URL `http://www.cl.cam.ac.uk/users/jrh/papers/demo.html`.

B. Huffman. Formal verification of monad transformers. In P. Thiemann and R. B. Findler, editors, *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*. ACM, Sept. 2012. URL `http://web.cecs.pdx.edu/~brianh/icfp2012.html`.

J. Hurd. The OpenTheory standard theory library. In M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *Third International Symposium on NASA Formal Methods (NFM 2011)*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, Apr. 2011. URL `http://gilith.com/research/papers`.

M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. URL `https://cs.uwaterloo.ca/~a78khan/cs446/additional-material/scribe/27-refactoring/Lehman-LawsOfSoftwareEvolution.pdf`.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.

B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

K. Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Technical University of Munich, 1999.