



Chapter 1

Background

1.1 Haskell

We mostly focus our attention on the Haskell programming language, both as an implementation vehicle and as our representation of functions, properties, etc. to explore. We choose Haskell since it combines formal, logical underpinnings which aid reasoning (compared to more popular languages like Java or C), yet it is still popular enough to sustain a rich ecosystem of tooling and a large corpus of existing code (compared to more formal languages like Coq or Isabelle). Haskell is well-suited to programming language research; indeed, this was a goal of the language’s creators [10]. An example of Haskell code is given in Figure 1.1.

Like most *functional* programming languages, Haskell builds upon λ -calculus, with extra features such as a strong type system and “syntactic sugar” to improve readability. The following features make Haskell especially useful for our purposes, although many are also present in other languages such as StandardML and Coq (which we also use, but only when needed for compatibility with prior work):

Functional: Most control flow in Haskell (all except *pattern-matching*, described below) is performed by function abstraction and application, which we can reason about using standard rules of inference such as *modus ponens*. For example, reading from external sources (like environment variables) is *impure*, so the only way to parameterise a value of type **B** with a value of type **A** is using a function of type **A** \rightarrow **B**. Conversely, applying a function of type **A** \rightarrow **B** to a value of type **A** can only ever produce a value of type **B** (it may instead crash the program or loop forever, but neither of those are *observable*, i.e. we can’t branch on them).

Pure: Execution of actions (e.g. reading or writing files) is separate to evaluation of expressions; hence our reasoning can safely ignore complicated external and non-local interactions. Purity ensures *referential transparency*: references can be substituted for their referent with no change in semantics (as in standard mathematical practice). This implies that calling

```

-- A datatype with two constructors
data Bool = True | False

-- A recursive datatype (S requires a Nat as argument)
data Nat = Z | S Nat

-- A polymorphic (AKA "generic") datatype
data List t = Nil | Cons t (List t)

-- Arithmetic functions

plus :: Nat -> Nat -> Nat
plus  Z  y = y
plus (S x) y = S (plus x y)

mult :: Nat -> Nat -> Nat
mult  Z  y = Z
mult (S x) y = plus y (mult x y)

-- Mutually-recursive functions

odd :: Nat -> Bool
odd  Z  = False
odd (S n) = even n

even :: Nat -> Bool
even  Z  = True
even (S n) = odd n

```

Figure 1.1: Haskell code, defining datatypes and functions involving them (note that `--` introduces a comment). `Bool` is the Booleans, `Nat` is a Peano encoding of the natural numbers and `List t` are polymorphic lists with elements of type `t`. Juxtaposition denotes a function call, e.g. `f x`, and functions may be defined by pattern-matching (case analysis) on their arguments. `A :: B` is an optional type annotation, stating that value `A` has type `B`. `A -> B` is the type of functions from `A` to `B`.

a function twice with the same argument must produce the same result (modulo crashes or infinite loops). For example, consider applying the function `pair x = (x, x)` to some arbitrary function call `foo y`. The references `x` in the resulting pair `(x, x)` both refer to `foo y` so, by referential transparency, can be substituted to get `(foo y, foo y)`. Both members of `(x, x)` are identical, by definition; hence, to preserve the semantics, both (and indeed *all*) calls to `foo y` must also be identical. This holds regardless of what we choose for `foo` and `y`, and implies that function behaviour cannot depend on external interactions or non-deterministic processes (which may change between calls).

Statically Typed: Expressions are constrained by *types*, which can be used to eliminate unwanted combinations of values (for example `plus True` is not a valid expression), and hence reduce search spaces; *static* types can be deduced syntactically, without having to execute the code. The type of Haskell expressions can also be *inferred* automatically.¹

Curried: All functions in Haskell take a single argument (as in λ -calculus), which makes them easier to manipulate programatically. Currying allows multi-argument functions to be simulated, by accepting one argument and returning a function to accept the rest. The `mult` function in Figure 1.1 has type `Nat -> Nat -> Nat` meaning it takes a `Nat` as argument and returns a function of type `Nat -> Nat`. Function calls are written with whitespace, so `mult x y` calls the `mult` function with the argument `x`, then calls the result of that with the argument `y`. This allows *partial application* such as `double = mult (S (S Z))`, but for our purposes it is important for unifying calling conventions. For example, in Javascript `mult x y` would be either `mult(x, y)` or `mult(x)(y)`, and depends on the definition of `mult` (the problem compounds with more arguments). In Haskell there is no distinction between these forms.

Non-strict: “Strict” evaluation strategies evaluate the arguments of a function call before the function body; non-strict does the opposite. The Haskell standards do not specify a particular evaluation strategy, but they do require that it be non-strict (for efficiency, most implementations use *lazy* evaluation to avoid duplicating work). Strictness can result in infinite loops and other errors which may be avoided by non-strictness, making the latter more useful for reasoning in the face of such errors. For example, given a pair of values `(x, y)` and a projection function `fst`, we might make the “obvious” conjecture that `fst (x, y) = x`.² This statement is true for non-strict languages, but *not* for strict languages. Crucially, a strict language will attempt to calculate the value of `y`, which may cause an infinite loop or other error; a non-strict language like Haskell will ignore `y` since it isn’t used in the body of the `fst` function.

¹Except for certain cases, such as those caused by the *monomorphism restriction* [10][§4.5.5]

²Incidentally, this is also a valid definition of the `fst` function

Algebraic Data Types: These provide a rich grammar for building up user-defined data representations, and an inverse mechanism to inspect these data by *pattern-matching* (Haskell’s other form of control flow). The `Bool`, `Nat` and `List t` definitions in Figure 1.1 are ADTs; whilst the functions use pattern-matching to branch on their first argument. For our purposes, the useful consequences of ADTs and pattern-matching include their amenability for inductive proofs and the fact they are *closed*; i.e. an ADT’s declaration specifies all of the normal forms for that type. This makes exhaustive case analysis trivial, which would be impossible for *open* types (for example, consider classes in an object oriented language, where new subclasses can be introduced at any time).

Parametricity: This allows Haskell *values* to be parameterised over *type-level* objects; provided those objects are never inspected. This enables *polymorphism*. The `List t` type in Figure 1.1 is an example: there are many useful functions involving lists which work in the same way regardless of the element type (e.g. getting the length, appending, reversing, etc.). An even simpler example is the polymorphic identity function `id x = x`. The type of `id` is `forall t. t -> t`³, which we can view as taking *two* parameters: a type `t` and a value of type `t`. Only the latter argument appears in the definition (as `x`), meaning that we can’t use the type `t` to determine the function’s behaviour. Indeed, in the case of `id` we can’t branch on the value of `x` either, since we don’t know what type it might have (our definition must work *for all* types `t`); the only functions we can call on `x` must also be polymorphic, and hence also incapable of branching. The type of `id` states that it returns a value of type `t`; without knowing what that type is, the only type-correct value it can return is the argument `x`. Hence the *type* of `id` tells us everything about its behaviour, with this style of reasoning known as *theorems for free* [14]. Haskell definitions are commonly made polymorphic like this, to prevent incorrect implementations passing the type checker, e.g. `fst :: (Nat, Nat) -> Nat` might return the wrong element, but `fst :: (t1, t2) -> t1` can’t.

Type classes: Along with their various extensions, type classes are interfaces which specify a set of operations over a type or other type-level object (like a *type constructor*, e.g. `List`). Many type classes also specify a set of *laws* which their operations should obey but, lacking a simple mechanism to enforce this, laws are usually considered as documentation. As a simple example, we can define a type class `Semigroup` with the following operation and an associativity law:

```
op :: forall t. Semigroup t => t -> t -> t
```

$$\forall x\ y\ z. op\ x\ (op\ y\ z) = op\ (op\ x\ y)\ z$$

³The `forall t.` is optional; type-level identifiers beginning with a lowercase letter are assumed to be universally quantified variables.

The notation `Semigroup t =>` is a *type class constraint*, which restricts the possible types `t` to only those which implement `Semigroup`.⁴ There are many *instances* of `Semigroup` (types which may be substituted for `t`), e.g. `Integer` with `op` performing addition. Many more examples can be found in the *typeclassopedia* [15]. This ability to constrain types, and the existence of laws, helps us reason about code generically, rather than repeating the same arguments for each particular pair of `t` and `op`.

Equational: Haskell uses equations at the value level, for definitions; at the type level, for coercions; at the documentation level, for typeclass laws; and at the compiler level, for ad-hoc rewrite rules. This provides us with many *sources* of equations, as well as many possible *uses* for any equations we might discover. Along with their support in existing tools such as SMT solvers, this makes equational conjectures a natural target for theory exploration.

Modularity: Haskell’s module system allows definitions to be kept private. This mechanism allows modules to provide more guarantees than are available just in their types, by constraining the ways that values can be constructed. For example, the following module represents email addresses as a pair of `Strings`, one for the user part and one for the host:

```
-- Exports appear between the parentheses
module Email (Email(), at, render) where

data Email = E String String

render :: Email -> String
render (E user host) = user ++ "@" ++ host

-- if/then/else is sugar for pattern-matching a Bool
at :: String -> String -> Maybe Email
at user host = if user == "" || host == ""
               then Nothing
               else Just (E user host)
```

An `Email` value can be constructed by passing any two `Strings` to `E`, but `E` is private (not exported). The `at` function is exported, but only passes its arguments to `E` iff they are not empty.⁵ Since this module never calls `E` with

⁴Alternatively, we can consider `Semigroup t` as the type of “implementations of `Semigroup` for `t`”, in which case `=>` has a similar role to `->` and we can consider `op` to take *four* parameters: a type `t`, an implementation of `Semigroup t` and two values of type `t`. As with parameteric polymorphism, this extra `Semigroup t` parameter is not available at the value level. Even if it were, we could not alter our behaviour by inspecting it, since Haskell only allows types to implement each type class in at most one way, so there would be no information to branch on.

⁵`Maybe t` is a safer alternative to the NULL construct of other languages [5]. It is defined as `data Maybe t = Nothing | Just t` and can be understood as an optional value, or a com-

empty `Strings`, and other modules must use `at`, we’re guaranteed that *all* `Email` values will have non-empty `Strings`. Such “smart constructors” can guarantee *any* decidable property, at the cost of performing run-time checks on each invocation.⁶

Together, these features make Haskell code highly structured, amenable to logical analysis and subject to many algebraic laws. However, as mentioned with regards to type classes, Haskell itself is incapable of expressing or enforcing these laws (at least, without difficulty [9]). This reduces the incentive to manually discover, state and prove theorems about Haskell code, e.g. in the style of interactive theorem proving, as these results may be invalidated by seemingly innocuous code changes. This puts Haskell in a rather special position with regards to the discovery of interesting theorems; namely that many discoveries may be available with very little work, simply because the code’s authors are focused on *software* development rather than *proof* development. The same cannot be said, for example, of ITP systems; although our reasoning capabilities may be stronger in an ITP setting, much of the “low hanging fruit” will have already been found through the user’s dedicated efforts, and hence theory exploration would be less likely to discover unexpected properties.

Other empirical advantages to studying Haskell, compared to other programming languages or theorem proving systems, include:

- The large amount of Haskell code which is freely available online, e.g. in repositories like Hackage, with which we can experiment.
- The existence of theory exploration systems such as HIPSPEC, QUICKSPEC and SPECULATE.
- Related tooling we can re-use such as counterexample finders (QUICKCHECK, SMALLCHECK, SMARTCHECK, LEANCHECK, HEDGEHOG, etc.), theorem provers (e.g. HIP [11]), and other testing and term-generating systems like MUCHECK [8], MAGICHASKELLER [7] and DJINN [1].
- The remarkable amount of infrastructure which exists for working with Haskell code, including package managers, compilers, interpreters, parsers, static analysers, etc.

Further evidence of Haskell’s suitability for theory exploration is given by the fact that the state-of-the-art implementation for Isabelle/HOL, the HIPSTER [6] system, is actually implemented by translating to Haskell and invoking HIPSPEC [3].

putation which may fail, or as a list with at most one element, or as a degenerate search tree with no backtracking [13]

⁶We can guarantee non-emptiness “by construction”, without run-time checks or `Maybe` wrappers, by changing the type to require at least one element. `String` is equivalent to `List Char`, with `"` represented as `Nil`. Changing `Nil` to require a `Char` would eliminate empty `Strings`, e.g. `data NonEmpty t = Nil t | Cons t (NonEmpty t)`. We could also use a pair like `data NonEmpty t = NE t (List t)` instead. Such precise types are often more desirable than smart constructors, but are less general since ad-hoc representations need to be invented to enforce each guarantee.

$$\begin{aligned}
\textit{expr} &\rightarrow \text{Var } id \\
&\quad | \text{Lit } \textit{literal} \\
&\quad | \text{App } \textit{expr} \textit{expr} \\
&\quad | \text{Lam } \mathcal{L} \textit{expr} \\
&\quad | \text{Let } \textit{bind} \textit{expr} \\
&\quad | \text{Case } \textit{expr} \mathcal{L} [\textit{alt}] \\
&\quad | \text{Type} \\
\textit{id} &\rightarrow \text{Local } \mathcal{L} \\
&\quad | \text{Global } \mathcal{G} \\
&\quad | \text{Constructor } \mathcal{D} \\
\textit{literal} &\rightarrow \text{LitNum } \mathcal{N} \\
&\quad | \text{LitStr } \mathcal{S} \\
\textit{alt} &\rightarrow \text{Alt } \textit{altcon} \textit{expr} [\mathcal{L}] \\
\textit{altcon} &\rightarrow \text{DataAlt } \mathcal{D} \\
&\quad | \text{LitAlt } \textit{literal} \\
&\quad | \text{Default} \\
\textit{bind} &\rightarrow \text{NonRec } \textit{binder} \\
&\quad | \text{Rec } [\textit{binder}] \\
\textit{binder} &\rightarrow \text{Bind } \mathcal{L} \textit{expr}
\end{aligned}$$

Where: \mathcal{S} = string literals
 \mathcal{N} = numeric literals
 \mathcal{L} = local identifiers
 \mathcal{G} = global identifiers
 \mathcal{D} = constructor identifiers

Figure 1.2: Simplified syntax of GHC Core in BNF style. $[]$ and $(,)$ denote repetition and grouping, respectively.

1.1.1 GHC Core

Whilst our systems and experiments use normal Haskell code, for simplicity we perform some of our analyses on an intermediate representation of the GHC compiler, known as *GHC Core*, rather than the relatively large and complex syntax of Haskell proper. Core is based on System F_C , which is described in detail in [12, Appendix C].

Core contains explicit type annotations and coercions, which we omit as they have no effect on runtime behaviour. The resulting sub-set of Core⁷ is shown in Figure 1.2; for brevity, we also omit several other forms of literal (machine words of various sizes, individual characters, etc.), since their treatment is similar to those of strings and numerals. We use quoted strings to denote names and literals, e.g. `Local "foo"`, `Global "bar"`, `Constructor "Baz"`, `LitStr "quux"` and `LitNum "42"`, and require only that they can be compared for equality.

GHC's translation from Haskell to Core is routine: Figure 1.3 shows the Core representation of functions from Figure 1.1. Although the Core is more verbose, we can see that similar structure in the Haskell definitions gives rise to similar structure in the Core; for example, the definitions of `odd` and `even` are identical in both languages, except for the particular identifiers used. It is this close correspondence which allows us to analyse Core expressions in place of their more complicated Haskell source.

Note that we exclude representations for type-level entities, including datatype definitions like that of `Nat`. GHC can represent these, but we are mostly concerned with dynamic behaviour, which excludes types (they are *erased* during compilation). For simplicity we also avoid using constructors directly, choosing instead to wrap them with normal functions (e.g. `s x = S x`), since this makes our domain more symmetric and reduces cross-language differences.

1.2 Property Checking

Although unit testing is the de facto industry standard for quality assurance in non-critical systems, the level of confidence it provides is rather low, and totally inadequate for many (e.g. life-) critical systems. To see why, consider the following Haskell function, along with some unit tests⁸:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)

fact_base      = factorial 0 == factorial 1
fact_increases = factorial 3 <= factorial 4
fact_div       = factorial 4 == factorial 5 `div` 5
```

⁷As of GHC version 7.10.2.

⁸Haskell functions can be prefix or infix: prefix functions have alphanumeric names, like `div` and `factorial`, and appear in function calls before their arguments, e.g. `div 10 5`; infix functions have non-alphanumeric names, like `+` and `==`, and appear in function calls between their first two arguments, e.g. `4 + 3`. Prefix functions can appear infix using backticks, like `10 `div` 5`, and infix functions can appear prefix using parentheses, like `(+) 4 3`.

plus

```

Lam "a" (Lam "y" (Case (Var (Local "a"))
  "b"
  (Alt (DataAlt "Z") (Var (Local "y")))
  (Alt (DataAlt "S") (App (Var (Constructor "S"))
    (App (App (Var (Global "plus"))
      (Var (Local "x")))
      (Var (Local "y"))))
    "x"))))

```

mult

```

Lam "a" (Lam "y" (Case (Var (Local "a"))
  "b"
  (Alt (DataAlt "Z") (Var (Constructor "Z")))
  (Alt (DataAlt "S") (App (App (Var (Global "plus"))
    (Var (Local "y")))
    (App (App (Var (Global "mult"))
      (Var (Local "x")))
      (Var (Local "y"))))
    "x"))))

```

odd

```

Lam "a" (Case (Var (Local "a"))
  "b"
  (Alt (DataAlt "Z") (Var (Constructor "False")))
  (Alt (DataAlt "S") (App (Var (Global "even"))
    (Var (Local "n")))
    "n"))

```

even

```

Lam "a" (Case (Var (Local "a"))
  "b"
  (Alt (DataAlt "Z") (Var (Constructor "True")))
  (Alt (DataAlt "S") (App (Var (Global "odd"))
    (Var (Local "n")))
    "n"))

```

Figure 1.3: Translations of functions in Figure 1.1 into the Core syntax of Figure 1.2. Notice the introduction of explicit λ abstractions (**Lam**) and the use of **Case** to represent piecewise definitions. Fresh variables are chosen arbitrarily as "a", "b", etc.

The intent of the function is to map an input n to an output $n!$. The tests check a few properties of the implementation, including the base case, that the function is monotonically increasing, and a relationship between adjacent outputs. However, these tests will *not* expose a serious problem with the implementation: it diverges on half of its possible inputs.

All of Haskell’s built-in numeric types allow negative numbers, which this implementation doesn’t take into account. Whilst this is a rather trivial example, it highlights a common problem: unit tests are insufficient to expose incorrect assumptions. In this case, our assumption that numbers are positive has caused a bug in the implementation *and* limited the tests we’ve written.

If we do manage to spot this error, we might capture it in a *regression test* and update the definition of `factorial` to handle negative numbers, e.g. by taking their absolute value:

```
factorial 0 = 1
factorial n = let nPos = abs n
              in nPos * factorial (nPos - 1)

fact_neg = factorial 1 == factorial (-1)
```

However, this is *still* not enough, since this function will also accept fractional values⁹, which will also cause it to diverge. Clearly, by choosing what to test we are biasing the test suite towards those cases we’ve already taken into account, whilst neglecting the problems we did not expect.

Haskell offers a partial solution to this problem in the form of *property checking*. Tools such as QUICKCHECK separate tests into three components: a *property* to check, which unlike a unit test may contain *free variables*; a source of values to instantiate these free variables; and a stopping criterion.

1.2.1 QuickCheck

QUICKCHECK [2] is the most widely used property checking library for Haskell¹⁰. Here is how we might restate our unit tests as QUICKCHECK properties:

```
fact_base      = factorial 0 == factorial 1
fact_increases n = factorial n <= factorial (n + 1)
fact_div      n = factorial n == factorial (n + 1) `div` (n + 1)
fact_neg      n = factorial n == factorial (-n)
```

The free variables (all called `n` in this case) are abstracted as function parameters; these parameters are implicitly *universally quantified*, i.e. we’ve gone

⁹Since we only use generic numeric operations, the function will be polymorphic with a type of the form `forall t. Num t => t -> t`, where `Num t` constrains the type variable `t` to be numeric. In fact, Haskell will infer extra constraints such as `Eq t` since we have used `==` in the unit tests.

¹⁰According to downloads counted at <http://hackage.haskell.org/packages/browse>, and packages depending on it at <https://packdeps.haskellers.com/reverse/QuickCheck>, accessed on 2019-05-21.

from a unit test asserting $\text{factorial}(3) \leq \text{factorial}(4)$ to a property asserting $\forall n, \text{factorial}(n) \leq \text{factorial}(n+1)$. Notice that unit tests like `fact_base` are valid properties; they just happen to assert rather weak statements, since they contain no free variables.



To check these properties, QUICKCHECK treats closed terms (like `fact_base`) just like unit tests: pass if they evaluate to `True`, fail otherwise. For open terms, a random selection of values are generated and passed in via the function parameter; the results are then treated in the same way as closed terms. The default stopping criterion for QUICKCHECK (for each test) is when a single generated test fails, or when 100 generated tests pass.

The ability to state *universal* properties in this way avoids some of the bias we encountered with unit tests. In the `factorial` example, this manifests in two ways:

- QUICKCHECK cannot test polymorphic functions; they must be *monomorphised* first (instantiated to a particular concrete type). This is a technical limitation, since QUICKCHECK must know which type of values to generate. In our example this would bring the generality of the type to our attention, which we could restrict to something non-fractional like `Int` (or the more general `Integral` type class, although we'd still need to pick a concrete type like `Int` for our tests).
- By default, QUICKCHECK picks a generator based on the *type* of value to be generated: since `Int` includes positive and negative values, the `Int` generator will output both. This will expose the problem with negative numbers, which we weren't expecting.

Whilst property checking generalises and improves upon unit testing, the problem of tests being biased towards expected cases remains, since the properties to be checked are still manually specified. Property checking can be complemented by *theory exploration* (TE), which avoids this bias by *discovering* such properties; through a combination of brute-force enumeration, random testing and (in the case of HIPSPEC and HIPSTER) automated theorem proving. Property checkers like QUICKCHECK are an important component of TE systems, since their data generators can the search process, and they can prevent “obvious” falsehoods from being output by checking them first.

1.3 Theory Exploration

(Automated) Theory Exploration (TE) is the task of taking gnature of definitions in some formal system (for example a programming language) and automatically generating a set of formal statements (properties) involving those definitions. These may be conjectures or theorems (proven either by sending conjectures to an automated theorem prover, or by having the generating procedure proceed in logically sound steps). These statements may also be “interesting” in some way, to rule out unhelpful trivialities such as iterating a pattern over and over (e.g. $x + 0 = x$, $(x + 0) + 0 = x$, ...).

The choice of appropriate methods can differ according to the underlying logic, etc. that is in use. In this work we ground ourselves by applying TE to discovering properties of pure functional software libraries, mostly in the Haskell programming language. This also gives our results immediate utility to software engineering. We ignore the issue of *proving* such conjectures, deferring that task to existing tools (such as HIPSPEC).

The method of conjecture generation is a key characteristic of any theory exploration system, although all existing implementations rely on brute force enumeration to some degree. We focus on QUICKSPEC [4], which conjectures equations about Haskell code. We have used version 1 of QUICKSPEC in our experiments, due to its availability, stability and tooling integration; hence references to QUICKSPEC are to version 1 unless stated otherwise. At the time of writing there is a QUICKSPEC version 2 available, which has a much improved generation procedure which can be significantly faster than its predecessor. Whilst QUICKSPEC 2 has advanced the state of the art in theory exploration, our preliminary experience has found that it still suffers the scaling issues we identify in this work¹¹. Since QUICKSPEC 2 has less integration with the tooling we have used, we leave a more thorough analysis of it (and related systems such as SPECULATE) for future work, pending the necessary infrastructure changes that would require.

QUICKSPEC (version 1) is written in the Haskell programming language, and works with definitions which are also written in Haskell. A thorough description of Haskell, including its suitability for theory exploration, can be found in § 1.1. For illustrative purposes, some simple Haskell definitions are shown in Figure 1.4. QUICKSPEC uses the following procedure to generate conjectures, which are both plausible (due to testing on many examples) and potentially interesting (due to being mutually irreducible):

1. Given a signature Σ of typed expressions and set of variables V , QUICKSPEC generates a list *terms* containing the expressions from Σ (including functions), the variables from V and type-correct function applications $f\ x$, where f and x are elements of *terms*. To ensure the list is finite, function applications are only nested up to a specified depth (by default, 3).
2. The elements of *terms* are grouped into equivalence classes, based on their type.
3. The equivalence of terms in each class is tested using QUICKCHECK: variables are instantiated to particular values, generated randomly, and the resulting closed expressions are evaluated and compared for equality.
4. If a class is found to have non-equal members, it is split up to separate those members.

¹¹We encountered memory exhaustion when trying to run some of the examples included with the QUICKSPEC 2 source code; `let alone` auto-generated signatures.

```
-- A datatype with two constructors (Note that -- introduces a comment)
data Bool = True | False

-- A recursive datatype (S requires a Nat as argument)
data Nat = Z | S Nat

-- A function turning a Bool into a Bool
not :: Bool -> Bool
not True  = False
not False = True

-- A pair of mutually recursive functions

odd  :: Nat -> Bool
odd  Z  = False
odd (S n) = even n

even :: Nat -> Bool
even  Z  = True
even (S n) = odd n
```

Figure 1.4: Haskell datatypes for booleans and natural numbers, followed by some simple function definitions (with type annotations).

5. The previous steps of variable instantiation and comparison are repeated until the classes stabilise (i.e. no differences have been observed for some specified number of repetitions).
6. For each class, one member is selected and equations are conjectured that it is equal to each of the other members.
7. A congruence closure algorithm is applied to these equations, to discard any which are implied by the others.

Such conjectures can be used in several ways: they can be presented directly to the user, sent to a more rigorous system like HIPSPEC or HIPSTER for proving, or even serve as a background theory for an automated theorem prover [3].

As an example, we can consider a simple signature containing the expressions from Figure 1.4, and some suitable variables:

$$\begin{aligned}\Sigma_{\text{Nat}} &= \{\text{Z}, \text{S}, \text{plus}, \text{mult}, \text{odd}, \text{even}\} \\ V_{\text{Nat}} &= \{\text{a} :: \text{Nat}, \text{b} :: \text{Nat}, \text{c} :: \text{Nat}\}\end{aligned}$$

QUICKSPEC's enumeration of these terms will resemble the following:

$$\begin{aligned}terms_{\text{Nat}} &= [\text{Z}, \text{S}, \text{plus}, \text{mult}, \text{odd}, \text{even}, \text{a}, \text{b}, \text{c}, \text{S Z}, \text{S a}, \text{S b}, \\ &\quad \text{S c}, \text{plus Z}, \text{plus a}, \dots]\end{aligned}$$

Notice that functions such as `plus` and `mult` are valid terms, despite not being applied to any arguments. In addition, all Haskell functions are unary (due to currying), which makes it valid to apply them one argument at a time as we construct $terms_{\text{Nat}}$.

The elements of $terms_{\text{Nat}}$ will be grouped into five classes, one each for `Nat`, `Nat -> Nat`, `Nat -> Nat -> Nat`, `Nat -> Bool` and `Bool`. As the variables `a`, `b` and `c` are instantiated to various randomly-generated numbers, the differences between these terms will be discovered and the equivalence classes will be divided, until eventually the equations in Figure 1.5 are conjectured.

Although complete, this enumeration approach is wasteful: many terms are unlikely to appear in theorems, which requires careful choice by the user of what to include in the signature. Here we know that addition and multiplication are closely related, and hence obey many algebraic laws; arbitrary definitions from a typical software library or proof development are unlikely to have been related so strongly.

```

      plus a b = plus b a
      plus a Z = a
plus a (plus b c) = plus b (plus a c)
      mult a b = mult b a
      mult a Z = Z
mult a (mult b c) = mult b (mult a c)
      plus a (S b) = S (plus a b)
      mult a (S b) = plus a (mult a b)
mult a (plus b b) = mult b (plus a a)
      odd (S a) = even a
      odd (plus a a) = odd Z
      odd (times a a) = odd a
      even (S a) = odd a
      even (plus a a) = even Z
      even (times a a) = even a
plus (mult a b) (mult a c) = mult a (plus b c)

```

Figure 1.5: Equations conjectured by QUICKSPEC for the functions in Figure 1.4; after simplification.

Bibliography

- [1] Lennart Augustsson. Djinn, a theorem prover in haskell, for haskell, 2005.
- [2] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *ACM SIGPLAN Notices*, 46(4):53–64, 2011.
- [3] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *Automated Deduction–CADE-24*, pages 392–406. Springer, 2013.
- [4] Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing Formal Specifications Using Testing. In Gordon Fraser and Angelo Gargantini, editors, *Tests and Proofs*, volume 6143 of *Lecture Notes in Computer Science*, pages 6–21. Springer Berlin Heidelberg, 2010.
- [5] Tony Hoare. Null references: The billion dollar mistake. *Presentation at QCon London*, 298, 2009.
- [6] Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. Hipster: Integrating Theory Exploration in a Proof Assistant. In Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban, editors, *Intelligent Computer Mathematics*, volume 8543 of *Lecture Notes in Computer Science*, pages 108–122. Springer International Publishing, 2014.
- [7] Susumu Katayama. MagicHaskeller: System demonstration. In *Proceedings of AAIP 2011 4th International Workshop on Approaches and Applications of Inductive Programming*, page 63. Citeseer, 2011.
- [8] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. Mucheck: An extensible tool for mutation testing of haskell programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 429–432. ACM, 2014.
- [9] Sam Lindley and Conor McBride. Hasochism: the pleasure and pain of dependently typed Haskell programming. *ACM SIGPLAN Notices*, 48(12):81–92, 2014.

- [10] Simon Marlow et al. Haskell 2010 language report. *Available online <http://www.haskell.org/> (May 2011)*, 2010.
- [11] Dan Rosén. Proving equational Haskell properties using automated theorem provers. Master’s thesis, University of Gothenburg, Sweden, 2012.
- [12] Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM, 2007.
- [13] Philip Wadler. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Conference on Functional Programming Languages and Computer Architecture*, pages 113–128. Springer, 1985.
- [14] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM, 1989.
- [15] Brent Yorgey. The typeclassopedia. *The Monad. Reader Issue 13*, page 17, 2009.