From Higher-Order Logic to Haskell: There and Back Again

Florian Haftmann *

Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany
http://www.in.tum.de/~haftmann/

Abstract

We present two tools which together allow reasoning about (a substantial subset of) Haskell programs. One is the code generator of the proof assistant Isabelle, which turns specifications formulated in Isabelle's higher-order logic into executable Haskell source text; the other is Haskabelle, a tool to translate programs written in Haskell into Isabelle specifications. The translation from Isabelle to Haskell directly benefits from the rigorous correctness approach of a proof assistant: generated Haskell programs are always partially correct w.r.t. to the specification from which they are generated.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.1 [*Software Engineering*]: Requirements/Specifications Tools; D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs; D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods; D.3.4 [*Programming languages*]: Processors—Code generation

General Terms Design, Languages, Verification

Keywords theorem proving, code generation, higher-order logic, Isabelle, Haskell

1. Motivation and overview

Formal verification forms an increasingly vital part in development of high-assurance software; typically, an implementation in a suitable programming language is verified against a formal specification in a proof assistant. We introduce two tools which provide previously non-existing support for this activity: Isabelle [Nipkow et al. 2002] with its built-in code generator transforms suitable specifications in higher-order logic (HOL) to corresponding Haskell programs.[1] Haskell programs can be made accessible to Isabelle by turning them into corresponding Isabelle specification text, which is accomplished by a separate tool called Haskabelle [Rittweiler and Haftmann].[2]

* Supported by DFG project NI 491/10-1.

[1] The code generator may also target other languages; also Isabelle provides further logics other than HOL. However this is not relevant for our focus here: Isabelle and HOL are used synonymously.

[2] The tools are available from http://isabelle.in.tum.de.

PEPM'10, January 18–19, 2010, Madrid, Spain.

In the following we examine and compare the essentials of Isabelle and Haskell and explain how they can be related to each other (§2). Next follows a description of the architecture of the tool suite (§3); its application is demonstrated with a few examples (§4), before we describe some generic methodology aspects (§5). A short survey of related work (§6) leads to a conclusion which provides clues to where to continue with further reading (§7).

2. The relationship between Isabelle and Haskell

2.1 A cursory glance at Isabelle

Some familiarity with Haskell is assumed. To approach Isabelle, for the scope of this paper it is sufficient to understand it as a kind of functional programming language; the following example specifies amortized queues represented by two lists:[3]

datatype α *queue* $=$ *Queue* $(\alpha$ *list*$)$ $(\alpha$ *list*$)$

definition *empty* $::$ α *queue* **where**
empty $=$ *Queue* [] []

primrec *enqueue* $:: \alpha \Rightarrow \alpha$ *queue* $\Rightarrow \alpha$ *queue* **where**
enqueue x (*Queue* *xs* *ys*) $=$ *Queue* (x # *xs*) *ys*

fun *dequeue* $:: \alpha$ *queue* $\Rightarrow \alpha$ *option* $\times\ \alpha$ *queue* **where**
dequeue (*Queue* [] []) $=$ (*None*, *Queue* [] [])
| *dequeue* (*Queue* *xs* (y # *ys*)) $=$ (*Some* y, *Queue* *xs* *ys*)
| *dequeue* (*Queue* (x # *xs*) []) $=$
(*case rev* (x # *xs*) *of* y # *ys* $\Rightarrow$ (*Some* y, *Queue* [] *ys*))

The innermost entities here are HOL expressions: types like α *queue* and terms like *empty* $=$ *Queue* [] []; these form part of statements in Isabelle's specification and proof language *Isar*, whose basic specification toolbox for functional programs consists of **datatype** for inductive datatypes, **definition** for simple (i.e. non-recursive) definitions, **primrec** and **fun** for different kinds of recursive definitions with pattern matching. The reason why there is a whole menagerie of statements for specifying constants[4] is that in a logic like HOL only simple definitions are permitted, to guarantee consistency; recursion must be defined using a suitable combinator or function graph representation, from which the user-supplied specification can be derived [Krauss 2006]. So each of these statements (excluding **definition**, but including **datatype**) internally

[3] $\Rightarrow$ and $\times$ denote function space and product type respectively; lists with # as infix cons operator and usual bracket syntax are built-in in Isabelle – but note that in Isabelle almost everything including syntax is definable by the user, so built-in is rather a matter of policy than mechanism.

[4] In Isabelle, constants are globally defined term symbols; in Haskell parlance, these would be called function symbols, function variables or operators.

issues a complex series of logic deductions to finally yield what the user actually wants. Sometimes constructing specifications needs guidance by the user; the neat thing is that automation behind **fun** is powerful enough to cope with most naively terminating recursive specifications, so in cases of doubt **fun** is the default choice.

Isabelle is not only a simple functional programming language – after things have been specified, proofs can be conducted, for example:

lemma *dequeue-enqueue-empty*:
dequeue (*enqueue x empty*) = (*Some x*, *empty*)
by (*simp add*: *empty-def*)

It is not our intention to give an introduction how to write Isar proofs; to understand the examples it is usually sufficient to use equational logic. Further Isar snippets can be found in §4.

2.2 Isabelle vs. Haskell

On the concrete source level, Isabelle and Haskell already appear quite similar: term and type expressions form part of *statements* which issue declarations (e.g. classes, datatypes or functions in Haskell). A series of statements is collected in a module, called *theory* in Isabelle and contained in a particular file.

This correspondence breaks down when considering abstract properties of the two systems: whereas Haskell's λ-calculus is very rich (F_ω [Pierce 2002] with corresponding type classes), Isabelle is much more conservative with its schematic polymorphism and simple type classes [Haftmann and Wenzel 2007].

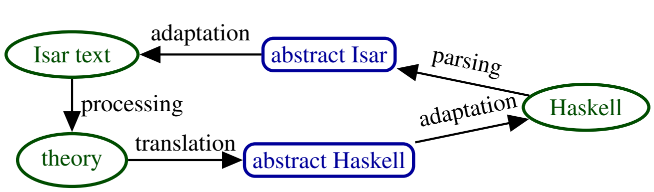
Also the purpose of each calculus is different: Haskell describes operational evaluation of terms in the internal GHC core λ-calculus F_c [Sulzmann et al. 2007] [5]; Isabelle formulates statements about constructibility and derivability of logical entities and propositions relative to a theory.

2.3 Bridging both worlds: shallow embedding

To bring both worlds together, a well-established intuition is *shallow embedding*: classes, type constructors and constants in the logic are *identified* with corresponding counterparts in the target language. How is then evaluation logically represented in the logic? In essence, the evaluation of a term in Haskell is the consecutive application of suitable equations as rewrite rules to a term; these equations stem from declarations in the program (typically function declarations or `instance` statements). Thus evaluation can be simulated in the logic by a consecutive application of equational theorems. When in the logic a term t inside a theory is evaluated to t' by means of applying equational theorems, it can be deduced that $t = t'$ holds. From this follows that if a set of equational theorems in a theory is translated to a Haskell program with the *same* equational semantics, this program is partially correct w.r.t. the original equational theorems.

3. Conceptional architecture

The following diagram illustrates how Isabelle and Haskabelle use shallow embedding:



[5] We use the Glasgow Haskell Compiler here for reference.

Isabelle produces Haskell source text in two steps: first, equational theorems from an abstract theory are *translated* into an abstract representation of Haskell programs which covers the typical structure of Haskell statements (`data`, function binding, `class` and `instance`); this abstract Haskell program has the same equational semantics as the original equational theorems, guaranteeing partial correctness. Before the final Haskell source text is written, an *adaptation* step allows to accomplish specific syntax (e.g. using built-in lists and common list syntax instead of providing a distinct list type).

The opposite direction starts with plain Haskell source text and *parses* it into an abstract representation of Isar text. At this stage unavoidably restrictions apply: the whole procedure must ignore (or fail for) artifacts which are not representable in Isabelle, e.g. due to its much more restricted type system. Before generating the final Isar text, customary syntax *adaptation* can be applied. This final Isar text can be processed by Isabelle again, resulting in an abstract theory.

4. Some examples

4.1 From Isabelle to Haskell

4.1.1 Amortized queues

The code corresponding to the queue example from §2.1 is straightforward:[6]

```
data Queue a = Queue [a] [a];

empty :: forall a. Queue a;
empty = Queue [] [];

dequeue :: forall a. Queue a -> (Maybe a, Queue a);
dequeue (Queue [] []) = (Nothing, Queue [] []);
dequeue (Queue xs (y : ys)) = (Just y, Queue xs ys);
dequeue (Queue (x : xs) []) =
  let {
    (y : ys) = reverse (x : xs);
  } in (Just y, Queue [] ys);

enqueue :: forall a. a -> Queue a -> Queue a;
enqueue x (Queue xs ys) = Queue (x : xs) ys;
```

The adaption stage as mentioned in §3 makes the code using the built-in list and maybe types and reverse operation; this is only a convenience – those could be generated also.

It is also possible to replace equations forming the program:

lemma [*code*]:
dequeue (*Queue xs* []) = (*if null xs then* (*None*, *Queue* [] [])
else dequeue (*Queue* [] (*rev xs*)))
dequeue (*Queue xs* (*y* # *ys*)) = (*Some y*, *Queue xs ys*)
by (*cases xs*, *simp-all*) (*cases rev xs*, *simp-all*)

The annotation *code* is an Isar *attribute* which states that the given theorems should be considered as code equations for the corresponding constant:

```
dequeue :: forall a. Queue a -> (Maybe a, Queue a);
dequeue (Queue xs (y : ys)) = (Just y, Queue xs ys);
dequeue (Queue xs []) =
  (if null xs then (Nothing, Queue [] [])
    else dequeue (Queue [] (reverse xs)));
```

[6] For brevity we always concentrate on the statements proper and omit the surrounding module statements, although these are also generated.

4.1.2 Rational numbers

In the examples so far, the specification text was always close to a functional program: in particular, pattern matching only occurred on constants which have been introduced as datatype constructors. But this is no requirement – the logical nature of Isabelle also permits more abstract specifications to be executed. A prominent example are rational numbers: logically, they are constructed as a quotient of pairs of integer numbers; concrete rational values are built using constant $Fract :: int \Rightarrow int \Rightarrow rat$, where $Fract\ p\ q$ is the value $\frac{p}{q}$ for $q \neq 0$ and 0 otherwise. Due to the syntactic structure of its type, *Fract* can serve as datatype constructor:[7]

```
data Rat = Fract Integer Integer;
```

Appropriate equations, e.g. for multiplication, are easily proved in Isabelle:

$$Fract\ a\ b * Fract\ c\ d = Fract\ (a * c)\ (b * d)$$

resulting in the following Haskell code:

```
times_rat :: Rat -> Rat -> Rat;
times_rat (Fract a b) (Fract c d) = Fract (a * c) (b * d);
```

This is a convenient place to present the proper treatment of equality. A suitable provable equational theorem is:

$$Fract\ a\ b = Fract\ c\ d \longleftrightarrow (\mathit{if}\ \mathit{is\text{-}zero}\ b\ \mathit{then}\ \mathit{is\text{-}zero}\ c \vee \mathit{is\text{-}zero}\ d\ \mathit{else}\ \mathit{if}\ \mathit{is\text{-}zero}\ d\ \mathit{then}\ \mathit{is\text{-}zero}\ a \vee \mathit{is\text{-}zero}\ b\ \mathit{else}\ a * d = b * c)$$[8]

Internally, an explicit type class *eq* is used to describe equality in a similar way as Haskell does; this class can be identified with the Haskell Eq class, resulting in the following statements:

```
eq_rat :: Rat -> Rat -> Bool;
eq_rat (Fract a b) (Fract c d) =
  (if is_zero b then is_zero c || is_zero d
    else (if is_zero d then is_zero a || is_zero b
           else a * d == b * c));

instance Eq Rat where {
  a == b = eq_rat a b;
};
```

4.2 From Haskell to Isabelle

4.2.1 Radix representations

Haskabelle is conveniently demonstrated by example:[9]

```
separate :: a -> [a] -> [a]
slice :: [Int] -> [a] -> [[a]]

radix :: (Int -> a) -> Int -> Int -> [a]
radix ch r n = if n <= 0 then [ch 0]
  else reverse (radx ch r n)

{-# HASKABELLE permissive radx #-}
radx :: (Int -> a) -> Int -> Int -> [a]
radx ch r n = if n <= 0 then []
  else let (m, d) = divMod n r in ch d : radx ch r m

hex :: Int -> String -> String
hex n sep = bunch $ radix ((!!) hexdigits) hexradix n where
  hexradix = 16
  hexdigits = "0123456789ABCDEF"
  bunch xs = concat (separate sep (slice ks xs)) where
    (q, r) = divMod (length xs) 4
    ks = if r == 0 then replicate q 4 else r : replicate q 4
```

[7] For convenience *int* values are mapped to `Integer`.

[8] Infix $(\longleftrightarrow)$ is syntactic sugar for $(=)$ on *bool* (equivalence).

[9] `separate` and `slice` bear no surprise and are only hinted at.

which results in the following Isar text:

function $(\mathit{sequential})\ \mathit{radx} :: (\mathit{int} \Rightarrow \alpha) \Rightarrow \mathit{int} \Rightarrow \mathit{int} \Rightarrow \alpha\ \mathit{list}$
where
$\mathit{radx}\ \mathit{ch}\ r\ n = (\mathit{if}\ n <= 0\ \mathit{then}\ \mathit{Nil}$
$\quad \mathit{else}\ \mathit{let}\ (m, d) = \mathit{divmod}\ n\ r$
$\quad\quad \mathit{in}\ \mathit{ch}\ d\ \#\ \mathit{radx}\ \mathit{ch}\ r\ m)$
sorry termination sorry

fun $\mathit{radix} :: (\mathit{int} \Rightarrow \alpha) \Rightarrow \mathit{int} \Rightarrow \mathit{int} \Rightarrow \alpha\ \mathit{list}$
where
$\mathit{radix}\ \mathit{ch}\ r\ n = (\mathit{if}\ n <= 0\ \mathit{then}\ [\mathit{ch}\ 0]\ \mathit{else}\ \mathit{rev}\ (\mathit{radx}\ \mathit{ch}\ r\ n))$

fun $\mathit{bunch}0$
where
$\mathit{bunch}0\ \mathit{sep}\ \mathit{xs} = (\mathit{let}\ (q, r) = \mathit{divmod}\ (\mathit{length}\ \mathit{xs})\ 4;$
$\quad \mathit{ks} = \mathit{if}\ \mathit{equal}\ r\ 0\ \mathit{then}\ \mathit{replicate}\ q\ 4$
$\quad \mathit{else}\ r\ \#\ \mathit{replicate}\ q\ 4$
$\quad \mathit{in}\ \mathit{concat}\ (\mathit{separate}\ \mathit{sep}\ (\mathit{slice}\ \mathit{ks}\ \mathit{xs})))$

fun $\mathit{hex} :: \mathit{int} \Rightarrow \mathit{string} \Rightarrow \mathit{string}$
where
$\mathit{hex}\ n\ \mathit{sep} = (\mathit{let}\ \mathit{bunch} = \mathit{bunch}0\ \mathit{sep};$
$\quad \mathit{hexradix} = 16;$
$\quad \mathit{hexdigits} = ''0123456789ABCDEF''$
$\quad \mathit{in}\ \mathit{bunch}\ \$\ \mathit{radix}\ (\mathit{nth}\ \mathit{hexdigits})\ \mathit{hexradix}\ n)$

Haskabelle tries pragmatically to follow the Haskell source text as close as possible: as a rule of thumb, one Haskell declaration maps to one Isar statement. Haskell permits an arbitrary order of declarations; if possible this is sequentialized for Isabelle, as in the case of *radix* and *radx*. Types and terms are translated literally, failing if their expressions exceed the bounds of the Isabelle type system or term expressions; for terms, these expressions include numerals, strings, list comprehensions, case expressions, and tuple bindings, but e.g. no guards or arbitrary bindings.

Local function definitions (which occur quite often in practice) are dealt with by decomposing them into global functions with suitable abstractions (constants *hex* and *bunch*0).

The adaptation of Haskabelle allows mapping operations directly on Isabelle counterparts (e.g. *concat*, *replicate*, `@`). Also a dedicated Isabelle theory *Prelude.thy* enriches the context in which the generated Isar specification is checked. Both can be customized by the user.

To preserve logical consistency, termination must be proved for each function definition. The automation of **fun** succeeds in lots of cases. Otherwise the function can be annotated with a pragma `{-# HASKABELLE permissive ...#}` in Haskell source; then it gets translated to a **function** where the user is supposed to replace the "holes" **sorry** with appropriate proof text.

Haskabelle also supports simultaneous import of several modules, where each module is mapped onto one theory.

4.2.2 Case study: finite maps

We have applied Haskabelle to a self-contained non-trivial Haskell module [Adams 1993] implementing finite maps as balanced trees. Encouragingly, only five modifications had to be made to the original code to let Haskabelle produce a working Isabelle theory; first, the literate Haskell source file had to be stripped of all non-code, which can be easily scripted. The substantial changes were:

line 63: `import Maybe ( isJust )` It would have been possible to provide the source code for the Maybe module or define an appropriate counterpart in the *Prelude* theory; for simplicity the definition of `isJust` was inlined here.

lines 228ff: `delFromFM` This function contains guards, which had to be desugared into *if*-cascades.

line 301: `Just elt1 = maybe_elt1` This non-tuple pattern bind had to be rewritten as partial case expression.

line 458: `instance Eq` This crude instance had been dropped.

Interestingly, **fun** is able to prove termination on all functions without any need for pragma annotations or proof assistance.

5. *State of the art*

The Isabelle code generator is an established and mature tool: it is used pervasively in the Isabelle distribution itself; its application methodology is well-understood, numerous applications use it for their purpose. Due to its conceptual simplicity and field-tested implementation it gives a high assurance.

Haskabelle is still close to its beginnings. Its applicability in principle to larger projects like the seL4 operating system kernel [Klein et al. 2009] has been figured out in preliminary experiments which seem quite promising. With Haskabelle as a focal point, we hope that future projects dealing with conversions between Isabelle and Haskell will profit from a common conversion tool which is adapted, extended and improved to meet specific needs, instead of constantly developing ad-hoc conversion scripts only fitting to a very specific setting. To this end we greatly encourage feedback and contributions for Haskabelle.

For the moment, two methodologies can be envisaged how to use Haskabelle:

- Program in Haskell, import into Isar theories, on top of these prove the desired properties.
- Import an existing Haskell project once into Isar theories, continue there and on demand produce Haskell again using the code generator.

Although the first seems more natural, the second may perhaps be easier to accomplish: the transition from Isar to Haskell is less delicate than the opposite direction, the translation source is not surface syntax but a properly internalized representation, and working exclusively in a unified environment like Isabelle is usually more satisfactory. Experience will show which path is more appropriate to go.

6. *Related work*

Turning specifications into programs is a well-established topic in theorem provers, typically with a bias towards the underlying implementation language of the system: ACL2 [Greve et al. 2007] and PVS [Crow et al. 2001] provides an intimate connection to the underlying Lisp language; since ACL2 is a subset of Common Lisp, programs written in this subset can be "imported" seamlessly.

The Coq system features a slightly different approach: programs are extracted from constructive proofs in the spirit of the Curry-Howard isomorphism [Letouzey 2004]; Haskell is supported.

Another approach for connecting Haskell with rigorous reasoning can be found in the Agda language [Bove et al. 2009] which combines a Haskell-like core with a dependent type system. A somehow different focus to include Haskell programs into formal processes can be found in Hets [Mossakowski et al. 2007] which lets Haskell snippets interact with various specification tools.

7. *Further reading*

We have given a rough overview and some taste of the possibilities of the couple Isabelle and Haskabelle. Further literature is available: an introductory course on Isabelle/HOL [Nipkow et al. 2002], a tutorial on code generation [Haftmann], the Haskabelle manual [Rittweiler and Haftmann] and a submitted Ph.D. thesis [Haftmann 2009] covering code generation from Isabelle in depth.

References

Stephen Adams. Efficient sets – a balancing act. *J. Funct. Program.*, 3 (4):553–561, 1993. URL `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/FiniteMap-0.1`.

Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs '09: Proceedings of the 22th International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 67–72. Springer, 2009. ISBN 978-3-642-03358-2.

J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001.

D. A. Greve, M. Kaufmann, P. Manolios, J S. Moore, S. Ray, J. L. Ruiz-Reina, R. Sumners, D. Vroon, and M. Wilding. Efficient execution in an automated reasoning environment. *Journal of Functional Programming*, 18(1):15–46, January 2007.

Florian Haftmann. *Code Generation from Specifications in Higher Order Logic*. PhD thesis, Technische Universität München, 2009.

Florian Haftmann. *Code generation from Isabelle theories*. `http://isabelle.in.tum.de/doc/codegen.pdf`.

Florian Haftmann and Makarius Wenzel. Constructive type classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs (TYPES 2006)*, volume 4502 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, October 2009. ACM Press.

Alexander Krauss. Partial recursive functions in higher-order logic. In *International Joint Conference on Automated Reasoning*, pages 589–603, 2006.

P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

Till Mossakowski, Christian Maeder, and Klaus Lüttich. The heterogeneous tool set, hets. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer-Verlag, 2007.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

Tobias Rittweiler and Florian Haftmann. *Haskabelle – converting Haskell source files to Isabelle/HOL theories*. `http://isabelle.in.tum.de/haskabelle.html`.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM Press, 2007. ISBN 1-59593-393-X.