Prover Programming

Compiling a Minimal Theorem Prover from Isabelle to Prolog

Thomas Pethick (s144448)

***Abstract*—We present a Prolog code generation of the *Simple Prover* program by Villadsen et al. which has been certified in the Isabelle theorem prover. The parsing of the Isabelle inner syntax and the subsequent Prolog compilation step is implemented in Haskell using a series of catamorphisms.**

The source code is accessible at https://github.com/tmpethick/simple-prover-pl/tree/paper.

I. Introduction

Isabelle is a theorem prover in which mathematical formulas are specified in higher-order logics (HOL) and subsequently proven. If the specification is a program it is desirable to be able to export it to other languages in which it could be executed. Isabelle already supports code generation of the HOL specification to the functional languages SML, OCaml, Haskell and Scala for which partial correctness is guaranteed [4, p. 1].

This paper aims to investigate Prolog code generation since the declarative style of both HOL and Prolog makes it an obvious target. It does so by considering a specific instance of a formally verified Isabelle program called Simple Prover [8]. Simple Prover is a theorem prover for first-order logic. It is a good case to examine since it aims to depend on as few language features as possible. This allows us to focus on the semantics of the structure being equivalent without having to define a mapping between built-in functions.

Whereas Simple Prover is certified using Isabelle, the compiled Prolog executable deviates from the original idea behind Simple Prover by not being provably correct. Instead, the objective is to provide preliminary work for code generation to Prolog guaranteeing partial correctness by proposing a series of transformations, yielding a Prolog version of Simple Prover that parses several end-to-end tests. The intention is to achieve partial correctness by eventually implementing the compiler in Isabelle but that is outside the scope of this paper.

The Isabelle parser and the Prolog compiler are implemented in Haskell with the abstract syntax trees (ASTs) represented as a parameterized abstract data types (ADTs). This allows us to split the transformation in the implementation into several catamorphisms, so each part of the transformation can be reasoned about separately. Concerning the parsing of Isabelle's inner syntax, details has been left out for brevity. Instead we refer to the implementation for elaboration on the lexer and the monadic parser.

Many of the observations made concerning Simple Prover are applicable to the HOL syntax independent of the specific program, thus providing preliminary work for compiling Prolog from Isabelle's HOL syntax in the more general case.

The paper is structured as follows: First the necessary Prolog terminology is introduced in §II. Then catamorphisms are introduced in §III with emphasis on tree structures including an additional note on how an AST can be annotated using the cofree comonad. With the necessary tools in place, we consider the implementation by first making precise what is being parsed in §IV. The transformation from Isabelle's HOL syntax to Prolog is then covered in §V. Subsequently, §VI evaluates the resulting Prolog program and discusses where it deviates from the original specification. Finally, §VII concludes and describes directions for further development.

II. Prolog Terminology

This section will cover essential terminology in Prolog used to described the transformations using the official documentation of SWI Prolog implementation [7]. It is not intended as an overview of how Prolog operates but rather focuses on the structure.

A *term* is the only data type and is either a *constant*, a *variable* or a *compound term*. Compound term consist of a *functor* (which is a name with an associated *arity*) followed by a number of *arguments* that are themselves terms.

The logic of a Prolog program is described using *clauses* which can be either a *rule* or a *fact*. A rule consists of a *head* and a *body* of the following form:

```
Head :- Body.
```

If the clause contains no body it is a fact which means that no conditions need to be satisfied for the head to be true. A collection of clauses with the same functor is called a *predicate*. Note that the word functor refers to different concepts in functional programming. Which one that is being used will be clear from the particular context.

III. Catamorphisms

Catamorphism, popularized by Meijer, Fokkinga and Paterson [5], generalizes the notion of a fold, which consumes a list to build up a return value. However, it works on an abstract data type instead of being restricted to lists.

To read this paper an operational approach to catamorphisms is sufficient, so we do not provide a category theoretic introduction. A catamorphism for an ADT defines a replacement function for each of the data constructors. Applying the catamorphism will deconstruct the input recursively and, in a bottom-up fashion, apply the replacement function and combine the intermediate return values to build the final return value [2] [1].

As a concrete example, consider a parameterized binary tree defined in Haskell:

```
data Tree = Leaf | Node Tree Tree
```

If we wanted to count all nodes, we could do so with the following recursive definition.

```
count Leaf       = 1
count (Node a b) = (count a) + (count b)
```

To generalize the recursion, `Tree` is parameterized to `TreeF a` so that it can contain the transformed children. This is problematic, however, since depending on the depth of the trees `a` and `b` will have varying type. So we take the *fixed point of the `TreeF` functor*.

```
data TreeF a = Leaf | Node a a deriving Functor

newtype Fix f = Fix (f (Fix f))
unfix (Fix f) = f

type Tree = Fix TreeF
```

Substituting `f` with `TreeF`, the usefulness becomes clear, `Fix (TreeF (Fix TreeF))`. Unfixing that constructor we obtain a value of type `TreeF` with children of type `Fix TreeF`, so that they are of the same fixed type independent of the depth.

This fixed point of the functor is sufficient to apply a catamorphism which for this particular instance is of type `(TreeF a -> a) -> Tree -> a`. Applying this to the counting example is done using the following algebra:

```
count = cata alg where
    alg Leaf     = 1
    alg Node a b = a + b
```

It is clear from the type signature of the algebra that the algebra has already been applied to the children and thus has the same type as the return type for the catamorphism. More precisely the ADT is traversed in a bottom-up approach in post-order so that the children have already been transformed using the algebra.

This allows us to implement the recursive transformation without expressing the recursive call explicitly. So the catamorphism captures this specific recursive pattern and for that reason is called a *recursive scheme*.

A further generalization of catamorphism leads to another recursive scheme that will be convenient called paramorphism. In the specific case of the tree type it is of type `(TreeF (Tree, a) -> a) -> Tree -> a`, thus providing a way to access the untransformed children through the first element of the tuple.

These morphisms can transform the ADT to a modified version or an entirely different type. Both cases will prove useful when applying it to pretty printing, Isabelle ADT to Prolog ADT transformation and finally the transformations within the Prolog ADT to form a valid Prolog AST.

A. Annotating AST with Cofree

The cofree comonad is useful for annotating ASTs. In the Free library it is defined in the following way [3].

```
data Cofree f a = a :< (f (Cofree f a))
```

It is similar to `Fix` but adds extra structure with the additional parameter `a`. With that structure it is possible to annotate every constructor in the recursively defined type and still have a type on which recursive schemes can be applied.

As example consider the type `TreeAnnotated` which associates with every `Node` and `Leaf` a boolean value.

```
type TreeAnnotated = Cofree TreeF Bool
```

A catamorphism can be defined that takes a tree of type `Tree` and annotates it using the cofree structure.

```
annotate = cata alg where
    alg e@Leaf       = True :< e
    alg e@(Node _ _) = False :< e
```

As mentioned above, the returned value can be further subjected to recursive schemes since it can be similarly unwrapped as `Fix`. However, when unwrapping, decisions can now be made based on the annotation as well. In this example the annotation only adds redundant information since the boolean value is already captured by the data constructors. More sophisticated information will be annotated however, in which a separation is desirable for maintainability. Additionally, applications in which the annotation is used repeatedly will reduce the computational complexity by acting as a type of memoization. Even though we do not concern ourselves with the efficiency of the transformation note that repeated use of the annotating is indeed the case for the Prolog transformation, as described in §V-C1.

IV. Parsing Isabelle

A. Inner Syntax

Isabelle is divided into an inner syntax and an outer syntax. The inner syntax provides a way to specify the main entities of the program in HOL [9, p. 166] while the outer syntax is the enclosing theory language used to prove that program [6]. The separation is achieved by enclosing the inner syntax in quotes (“”) or cartouches (‹›).

The aim is to compile the verified program itself so parsing of the inner syntax is sufficient. However, the program can consist of several cartouches-enclosed inner syntax entities in the outer syntax. Therefore the parser is extended to allow for the small subset of the outer syntax which is the cartouche. In this way it is possible to parse multiple lines of inner syntax.

B. Pretty Print

A pretty printer of Isabelle is defined for testing purposes. This allows us to compose together the parser and printer and verify that it is indeed the identity function, since by definition it holds that

$$printer \circ parser = id.$$

The transformation from the AST uses a paramorphism so that parenthesis can be properly applied depending on the precedence. This is necessary since whether to wrap a given child in parenthesis is dependent on the structure of the AST. Using a catamorphism would only make the transformed child available so pattern matching on the AST type would not be possible.

For completeness, parsing of the quantifiers in the inner syntax have also been included. This is redundant for the Prolog compilation but it simplifies testing since the identity then holds.

V. Transforming into Prolog with Catamorphisms

The Isabelle AST is transformed into a Prolog AST by composing together several catamorphisms. The first catamorphism transforms the abstract data type (ADT) used to represent the Isabelle AST into one used for the Prolog AST. This initial transformation leaves an invalid Prolog AST but the weaker definition allows us to continuously modify it through catamorphisms, finally leaving us with a valid Prolog AST structure. The various transformations are covered in the order in which they are applied. To differentiate between constructors, we prefix the Isabelle AST and Prolog AST data constructors by `I` and `P` respectively.

The AST before and after substitution of equivalence will be referred to as $\mathcal{P}_{eq}$ and $\mathcal{P}_{clause}$ respectively.

A. Function Applications

Function application on multiple arguments are represented in their curried form in the Isabelle AST. The nested structure this leads to is an inconvenient representation for Prolog, since modifying the arguments will prove to be useful. An example of this transformation can be seen on fig. 1. This in turn also identifies the function name which should not be capitalized in contrast to the variable terms.

This will further be transformed by a subsequent catamorphism since function applications are not part of the Prolog syntax. This only provides a convenient intermediate step.

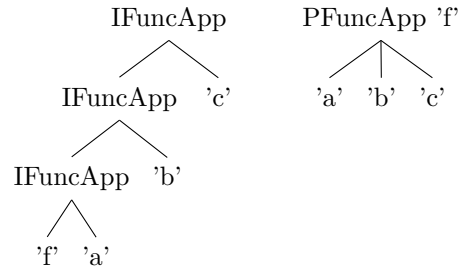

Fig. 1: Transformation for nested function application. The Isabelle AST and the Prolog AST are the left and right trees, respectively.

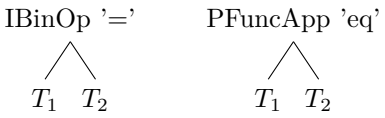

Fig. 2: Transformation for operators. The Isabelle AST and the Prolog AST are the left and right trees, respectively.

B. Operators

There are several infix operators (append, equivalence and conjunction), unary operator (not) and ternary operators (if-then-else) in Isabelle that have equivalents in Prolog. However, Simple Prover includes the definitions for these operators, so they are transformed into regular function application (see fig. 2 for an example). Thereby these operators can be handled identically to a regular function application.

C. Transforming Equivalence into Clauses

A specific application of 'check' in the Isabelle definition of Simple Prover can be run by repeatably applying the equivalences to rewrite it. Given a program P with an equivalence $A \equiv B$, denote by P' the program in which A is substituted with B in P. By definition P' is equivalent to P.

It is possible to mimic this rewriting using unification in Prolog, however we cannot simply substitute A with B in $\mathcal{P}_{eq}$. Instead transform $A \equiv B$ into a predicate clause with A as predicate. Additionally introduce B as a new last argument in the predicate A. Now, use a unique variable for where the predicate is used in P. This variable can now be unified with B. If a solution is found for some unification of B, by definition of the predicate clause it holds for A.

Note that there are therefore fewer assumptions in the transformed Prolog program since the substitution of the left hand side by the right hand side is allowed but not the converse. So the definitions in the two languages are not strictly the same, however Simple Prover only uses the left implication of the equivalence nonetheless.

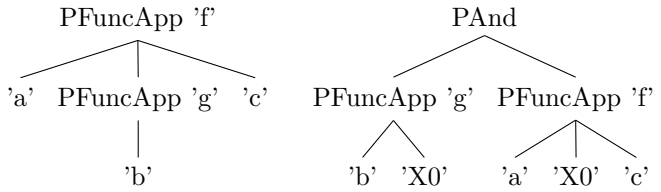

Fig. 3: Transformation for a clause body where a nested function application is transformed into a sequence of terms connected by the and-operator.

1) Clause Body Transformation: So far a detailed description on how predicates are treated in the body has been avoided. The Prolog AST has been transformed so that the last argument of every predicate represents the output. Let a predicate P_1 depend on another predicate P_2. Then P_1 is transformed to a Prolog sequence consisting of P_1 and P_2 where P_2 is substituted by a unique variable in P_1, as exemplified by fig. 3.

This procedure is accomplished by first annotating every function application in the body AST with a unique name. The uniqueness is achieved using a monadic catamorphism with a *supply* monad that increments the name. The AST is then annotated by fixing the AST with Cofree instead of Fix that attaches a maybe monad including the annotation (see §III-A). The annotated AST is traversed in post-order using catamorphism and yield one element for each annotate child. Let $T(X)$ be a transformation that adds the root annotation of X as the final element in the top predicate and replaces all other subtrees in X with its annotation if such exists. Then each element is transformed using T.

Call the resulting list the *dependencies* and the transformed root the *body root*. This procedure then returns both the dependencies and the transformed root using T as expressed in the example on fig. 4.

2) Building Clause from Head and Body: Depending on the structure of the clause body it is possible to simplify the clause in different ways. First, if the body root is a compound term, the last argument of that compound term is appended as an argument to the head term. Otherwise the body root itself is appended to the head instead. If the body is now empty the clause can further be simplified by transforming it into a fact consisting only of the head.

D. Compound Terms

We have to distinguish between compound terms with and without an associated predicate. Take the following section from Simple Prover defined in Isabelle's inner syntax.

```
frees (Uni p # base s)
```

`base` is defined using an equivalence in Isabelle and is thus a predicate in the Prolog AST. However, `Uni p` is no predicate and should therefore not get appended an argument treated as an output. Currently whether to interpret it as a predicate or not is based on whether the functor is capitalized or not. This leverages the fact that Simple Prover introduces explicit structure by consistently annotating the function names differently. A more robust method that does not depend on this additional specification could base the distinction on whether there existed a clause or not for the compound term in question.

E. Singletons

The steps described prior to this section are sufficient for compiling a working Prolog version of Simple Prover. However, warnings are thrown since singleton variables occur. This happens if variables are bound to a name used nowhere in that scope. The name is redundant since it is not used to unify the variable so it should be replaced by an anonymous variables.

A catamorphism creates a list of all variables used in a clause with repetitions. This is transformed into a set consisting of variables with only one occurrence. Every variable with only one occurrence, based on a lookup in this set, is replaced by an anonymous variable using a final catamorphism.

What is left is a AST with no clauses containing variables with only a single occurrence.

VI. Results

It was possible to transform the definition of Simple Prover in Isabelle into a Prolog program. This program was verified by several Prolog unit tests and 33 end-to-end tests on valid formulas to ensure that it behaved identically to the Isabelle implementation. It does not compare with the confidence the formally verified version in Isabelle provides. However, it does suggest that the transformation is reasonable even though it is uncertain if some edge case is not covered. Given the fact that Simple Prover relies so little on language features by containing the necessary definitions itself, a discrepancy between evaluation in Isabelle and Prolog is even less likely.

One crucial point where the Prolog version deviates is w.r.t. invalid formulas. P would be to never terminate whereas the compile version terminates since it runs out of stack space. An invalid formula would lead to a Prolog program repeatably applying the recursively defined rule

```
prover([H|T],Y) :- solves([H|T],X0), prover(X0,Y)
```

For the minimal invalid formula $p(x)$ the same arguments are parsed to `prover` indefinitely as evident from the following trace:

```
Call: (1) check(pre(1, 0, []), 1)
Call: (2) prover([[(0, pre(1, 0, []))]], 1)
Call: (3) solves([[(0, pre(1, 0, []))]], _11118)
...
Call: (3) prover([[(0, pre(1, 0, []))]], 1)
```

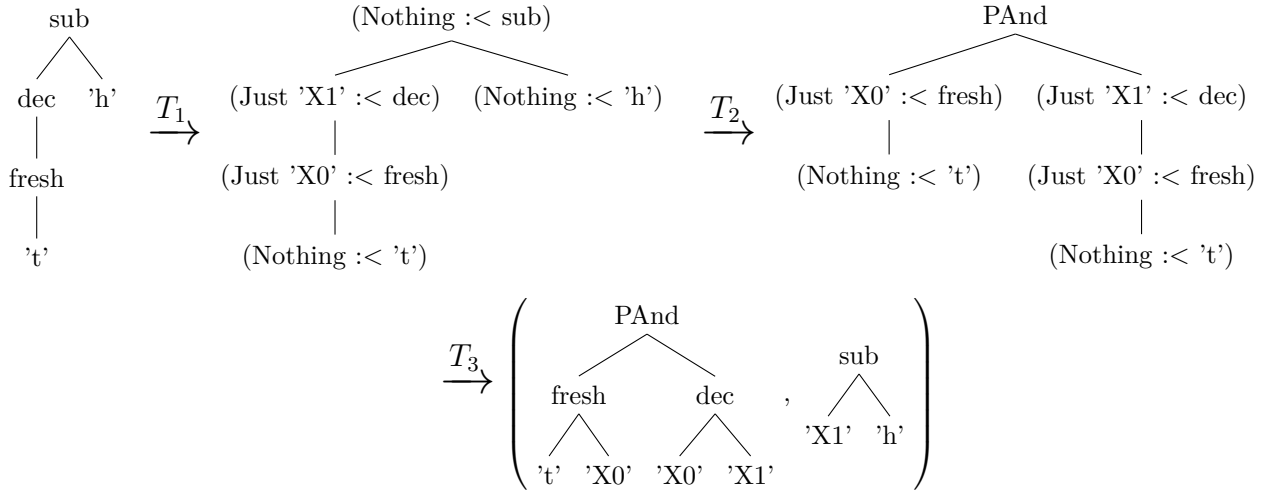

Fig. 4: Transformation of body in $\mathcal{P}_{eq}$ to a Prolog and-sequence with slight abuse of Cofree notation to consistently display it as a tree. Data constructors are left out for brevity where it is clear from the context. We divide the transformation into three parts 1) T_1 that annotates the AST 2) T_2 which creates a subtree for each annotated vertex (referred to as dependencies), and lastly 3) T_3 which uses the annotations to replace the subtrees in the dependencies and similarly for the root vertex.

Tail call optimization should prevent this. The program execution seems to match that of the original implementation. It is thus a question of whether it is optimized in a similar way to the execution of inner syntax in Isabelle.

It has several limitations however if we look outside this specific case study; the Isabelle parser is incomplete since it cannot parse all valid Isabelle syntax. Only the subset of operators used by Simple Prover was implemented. However, this is a trivial extension but outside the scope of this paper.

The parser still parses more Isabelle syntax then what the Prolog transformation allows for. This is because the equation-based approach used by Simple Prover is expected. It requires a top level equation with only one function application on the left side. For a more general transformation a review of the semantics of the Isabelle inner syntax is required.

VII. Conclusion

A program specified in Isabelle's HOL syntax, Simple Prover, was transformed into Prolog by a parser and compiler written in Haskell using a series of catamorphisms. It was evaluated on several logical formulas to ensure that it behaved similarly to the original formally verified implementation. The only observed inconsistency is w.r.t. invalid formulas for which the program runs out of stack space instead of never terminating. The study presented have provided preliminary work for a more general code generation of Prolog from Isabelle. We leave with a few remarks on how the code generation can be improved and generalized.

A. Further Work

We identify four main directions, for moving towards a more general Prolog code generation.

One problem with the current implementation is that the Prolog AST is not strict enough since the initial transformation into the Prolog AST has to capture an intermediate structure which is not necessarily a valid Prolog program. To ensure a valid AST through the type system a final step could convert it to another stricter prolog AST. Eventually, the aim is to implement a verified code generation of Prolog in Isabelle to verify the semantics are preserved.

Simple Prover uses a minimal subset of the built-in methods by defining e.g. if-else and equivalence. A transformation using if-else and cut in Prolog is needed to capture programs that leverage built-in methods. This could be further extended to use built-in boolean values which would in turn allow for maintaining e.g. equal as an infix operator.

Apart from using more of the features in Prolog it could also be desirable to simplify code. It could be possible to obtain cleaner code by generalizing concepts such as fold and map. These cases are already present in Simple Prover in the form of `solves` and `base` respectively.

Finally, disregarding Simple Prover an application independent analysis of the Isabelle semantics is necessary to create a truly general mapping to Prolog.

VIII. Acknowledgements

I would like to thank Jørgen Villadsen for supervising the project and Andreas From for providing helpful feedback on both the Haskell implementation and the presentation of it.

References

[1] Jacob Andersen and Claus Brabrand. "Syntactic language extension via an algebra of languages and transformations". In: *Electronic Notes in Theoretical Computer Science* 253.7 (2010), pp. 19–35.

[2] Jacob Andersen, Claus Brabrand, and David Raymond Christiansen. "Banana Algebra: Compositional syntactic language extension". In: *Science of Computer Programming* 78.10 (2013), pp. 1845–1870.

[3] *Control.Comonad.Cofree*. https://hackage.haskell.org/package/free-4.12.4/docs/Control-Comonad-Cofree.html#t:Cofree. (Accessed on 01/26/2018).

[4] Florian Haftmann and Lukas Bulwahn. *Code generation from Isabelle/HOL theories*. Technical Report, http://isabelle. in. tum. de/dist/Isabelle2013-2/doc/codegen. pdf, 2013.

[5] Erik Meijer, Maarten Fokkinga, and Ross Paterson. "Functional programming with bananas, lenses, envelopes and barbed wire". In: *Functional Programming Languages and Computer Architecture*. Springer, 1991, pp. 124–144.

[6] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *A Proof Assistant for Higher-Order Logic*. 2013.

[7] *SWI-Prolog – Manual*. http://www.swi-prolog.org/pldoc/man?section=glossary. (Accessed on 01/26/2018).

[8] Jørgen Villadsen, Anders Schlichtkrull, and Andreas Halkjær From. "Code Generation for a Simple First-Order Prover". In: *Isabelle Workshop, Nancy, France*. 2016.

[9] Makarius Wenzel. *The Isabelle/Isar Reference Manual*. http://isabelle.in.tum.de/doc/isar-ref.pdf. (Accessed on 12/25/2017).