Proposal for a bachelor’s thesis

Equational reasoning modulo tuples

Supervisors: Dr. Andreas Lochbihler
Professor: Prof. David Basin

Issue date:

Prerequisites

- Good skills in functional programming (e.g., from the FMFP course)
- Knowledge about logic and the \( \lambda \)-calculus (e.g., from the FMFP course)

Introduction

In mathematics and functional programs, operators often bind variables. For example, the quantifier \( \forall y \in A. \ P(y) \) binds the variable \( y \) in the formula \( P \), the integral \( \int_a^b f(x) \, dx \) binds \( x \), and in Haskell the following \texttt{do} block binds the variable \texttt{name}.

\begin{verbatim}
    do
        name <- getLine
        putStrLn ("Hello" ++ name ++ ".")
\end{verbatim}

Such operators come with various laws that allow us to reason about them. For example, the universal quantifier is commutative, namely

\[
(\forall x \in A. \forall y \in B. \ P(x,y)) = (\forall y \in B. \forall x \in A. P(x,y))
\]

for all \( A, B, \) and \( P \). Similarly, the operator \( (>>=) \) underlying the \texttt{do} notation is associative as expressed by

\[
(x >>= (\ y -> f y >>= g)) = ((x >>= f) >>= g)
\]

For example, one can prove that the program above and the one below do the same – even if nothing is known about \texttt{getLine}, \texttt{putStrLn}, and \( +{} \).

\begin{verbatim}
    do
        msg <- do
            name <- getLine
            return (name ++ ".")
        putStrLn ("Hello" ++ msg)
\end{verbatim}
Such proofs (both about mathematical notions and of functional programs) can be done in the proof assistant Isabelle/HOL [5], which checks that every proof step is correct and automates many of the trivial ones.

Instead of variables, we can bind tuples, too, e.g., \( \forall (x, y) \in \mathbb{R}^2, \forall (m, n) \in \mathbb{N}^2. \ldots \) and do

\[
(name, handle) \leftarrow \text{System.IO.openTempFile } /tmp\text{ "tmp.txt"} \\
... 
\]

Then, however, reasoning is more complicated. For example, Haskell desugars this snippet as follows, where a case expression takes the tuple apart, and so does Isabelle/HOL.

\[
\text{System.IO.openTempFile } /tmp\text{ "tmp.txt"} \\
\quad >>=(\x \to \text{case } x \text{ of } (name, handle) \to \ldots )
\]

Suppose that \( \ldots \) is of the form \( f \text{ name handle } >>= g \). To use the associativity law, our reasoning must deal with the case that has slipped in between the abstraction \( \\lambda x \to \) and the bind operator \( >>= \). Unfortunately, Isabelle/HOL’s reasoning does not support this yet.

**Objectives**

In this project, rewriting modulo tuples shall be implemented in Isabelle/HOL as a first step towards reasoning modulo tuples. This can be done as a new proof tactic on top of the existing reasoning engine.

Rewriting with an equation \( \text{lhs} = \text{rhs} \) in an expression \( t \) consists of two steps. First, one must find all the subexpressions to be replaced. To that end, one examines every subexpression and checks whether the left hand side \( \text{lhs} \) matches the subexpression. This requires an implementation of an algorithm for higher-order *product* pattern matching. Ordinary higher-order pattern matching has already been implemented in Isabelle by Nipkow [3]. Fettig and L"ochner [2] have generalised Nipkow’s approach to handle tuples, but only at an abstract level without implementation. When the matching succeeds, the algorithm yields an instantiation for the variables in the equation such that \( \text{lhs} \) becomes equal to the subexpression except for splitting and collapsing of tuples.

Second, the tuples in the left hand side of the instantiated equation must be split or collapsed such that it becomes syntactically equal to the subexpression to be replaced. Then, the instantiated and transformed equation can be given to Isabelle’s proof checker (which needs not know anything about tuples). To that end, the matching algorithm must be extended to compute – in addition to the instantiation – a transformation that takes care of splitting and collapsing the tuples as needed.

**Tasks**

This project can be subdivided into the following tasks:

1. Make yourself familiar with Isabelle/HOL (e.g., by working through the relevant tutorials [4], [6]).
2. Study the relevant literature on term rewriting and higher-order pattern matching \cite{1,2,3}.

3. Adapt the higher-order product pattern matching algorithm to the term representation used by Isabelle/HOL and implement it in SML. Write regression test cases.

4. Extend the algorithm to compute the transformation that takes care of splitting and collapsing the tuples.

5. Implement a proof method for single-step rewriting modulo tuples. Write regression test cases.

6. (optional) Adapt the algorithm to expand tuples in the higher-order product patterns on the fly such that the unification algorithm can deal with type variables.

7. (optional) Generalise the implementation from pairs to user-defined (non-extensible) record types.

8. (optional) Generalise the existing syntax for tuple patterns to user-defined (non-extensible) records.

9. Write the final report and prepare the presentation.

**Deliverables**

The following deliverables are due at the end of the project:

- **Final report** The final report should consist of an introduction; a theoretical background section; one or more sections describing the challenges in Isabelle/HOL, the implementation, and the test cases; and a conclusion. The report may be written in English or German. Three copies of the report must be delivered to the supervisor.

- **Isabelle/HOL theories** Complete Isabelle/HOL development that runs with the latest release or a recent developer’s version.

- **Presentation** At the end of the project, a presentation of 20 minutes must be given during an InfSec group seminar. It should give an overview and discuss the most important highlights of the work.

**References**


