Relation

Relation algebra has been studied in mathematics and logic for more than one and half century. In the recent twenty years, relation algebra has been found to be a useful tool in computer science and software engineering. An simple example is that a program is viewed as a binary relation with input as domain and output as codomain. Allegory theory is an abstraction of relations just as category theory to functions. Roughly speaking, an allegory consists of a collection of abstract relations and an interface \(<\text{dom}, \bowtie, \cap, \vdash>\). Proving things in allegory theory is not an easy task.

In general, Bird and de Moor observe recently that “the calculus of relations has gained a good deal of notoriety for the apparently enormous number of operators and laws one has to memorise in order to do proofs effectively.”

Decision Procedure

There is no normal form for expressions in allegories, but for a certain graphical representation, normal forms do exist. Therefore, the first step is to convert both sides of an equation to graphs. The implementation of converting is similar to ShowS in Haskell, which converts an expression in linear time. The conversion rules are shown here:

1: S \rightarrow R \rightarrow 2: F
(a) Variable R

1: S \rightarrow R \rightarrow 2: F
(b) \I

1: S \rightarrow R \rightarrow 3: \rightarrow Q \rightarrow 2: F
(c) R \vdash Q

1: S \rightarrow R \rightarrow 2: F
(d) R \cap Q

2: F \rightarrow R \rightarrow 1: S
(e) R^\sim

2: SF \rightarrow \rightarrow R \rightarrow 3:
(f) \text{dom}(R)

Figure 1: Rules

The decision procedure decides whether an equation $L = R$ is valid in all allegories.

It starts with converting both sides to graphs, and then reduces these graphs to normal forms. Reducing a graph is to find another graph which contains less nodes and is both a sub-object and a quotient-object of the given graph. A normal form of a graph cannot be reduced further.

The last step is to test whether there exists an isomorphism between the two normal forms. This is known to be an NP-complete problem. Therefore, the computation could be expensive.
More To Do

Our final goal is to integrate AllegEq to a theorem prover. If AllegEq acts as an external program to a prover, we have to make the prover “trust” the results from AllegEq. Ideally a proof should be generated when AllegEq claims two expressions are equal.

Haskell

Haskell is the most popular pure functional programming language. In particular, it is a polymorphically typed, lazy, purely functional language, quite different from most other programming languages. The language is named after Haskell Brooks Curry, whose work in mathematical logic serves as a foundation for functional languages.

AllegEq is implemented in Haskell. It allows us to reason about the implementation more easily. For example, the requirement for \((\text{conv } gf) \ g\) is \(gf \sim \land \ g\) and the implementation of function is \(\text{swapSF } (gf (\text{swapSF } g))\), where \(\text{swapSF } g \cong g \sim\).

\[
\begin{align*}
\text{conv } gf \ g &= (\text{Implementation}) \\
&= \text{swapSF } (gf (\text{swapSF } g)) \\
&\equiv (gf \land g) \sim \\
&= gf \sim \land g \sim \\
&= gf \sim \land g
\end{align*}
\]