Representation of Trivial Path in AST

by Sven Nilsen, 2019

In this paper I represent an insight about trivial paths from an experimental automated solver.

Earlier work has shown that function identity changes with the trivial path. This is problematic for solvers, since either semantics is unpreserved or it requires complex data structures. Here I show a method that treats the trivial path as a binary operator extending the inner AST node.

The major insight is that a trivial path can be treated as a binary operator that passes in information to operations when transforming the AST (Abstract Syntax Tree) to logical equivalent expressions. This binary operator can be thought of as a data structure extension such that an encapsulated function satisfies the semantics of constrained functions. This provides the solver with sufficient information to preserve semantics of the AST under transformation.

In Rust code:

```
pub enum Expr {
    // Existential path can be thought of as a unary operator.
    Ex(Box<Expr>),
    // Trivial path can be thought of as a binary operator.
    Tri(Box<Expr>, Box<Expr>),
    // Symmetric path e.g. `add[even]` can be thought of as a binary operator.
    SymPath(Box<Expr>, Box<Expr>),
    // Application uses symmetric paths or references.
    App(Box<Expr>, Vec<Expr>),
    // Other more advanced nodes.
    …
    // Referencing enum variants representing unary operators.
    Un(fn(Box<Expr>) → Expr),
    // Referencing enum variants representing binary operators.
    Bin(fn(Box<Expr>, Box<Expr>) → Expr),
    ...
    // Standard path semantics functions.
    Id(Box<Expr>),
    True1(Box<Expr>),
    …
    // Literals.
    Bool(bool),
    I64(i64),
    …
}
```

Here, a clever trick is used to reduce the complexity of the AST. Instead of keeping a separate struct for referencing functions, the AST **refers to its own enum variants**. One can think about this as a problem of reducing the AST to concrete enum variants from paths.

This method satisfies the core axiom of path semantics, because referencing enum variants from a function pointer is like creating a symbol with associated collections of symbols. It also abstracts over the generated code.

In a general theorem prover, the standard lambda calculus is extended for path semantics with e.g. Higher Order Operator Overloading and Natural Lambda Properties, such that enum variants represent both the functions in the executing code and themselves. Instead of computing through evaluation only, it is generalized to operations that transforms the AST.

To keep operations under control for automated theorem proving, one can use a recursive addressing pattern that control which branch of an expression is being transformed:

```
pub enum Op {
    // Run this operation on the left child.
    Left(Box<Op>),
    // Run this operation on the right child.
    Right(Box<Op>),
    …
    // Operations for existential paths.
    ExId,           // $∃id_T$ => $true_{1T}$
    ExTrue1,        // $∃true_{1T}$ => $id_{bool}$
    …
    // Operations for normal paths.
    SymId,          // f[id] => f
    NotSymNot,      // not[not] => not
    …
}

impl Op {
    // Pass in trivial path to operator, usually `&Un(True1)` (no domain constraints).
    fn perform(&self, expr: &Expr, tri: &Expr) → Option<Expr> { … }
}
```

The operator can then make the right choices to preserve the semantics.

The trivial path is the only AST node that alters the information to the operator. If one consider this from a higher order perspective, it makes sense in a deep intuition about mathematics, because the domain constraint of the operator is changed by a self-referential mathematical object. In this context, the trivial path refers to itself in a way that also refers to modification of the domain constraint, which is exactly what the trivial path does. It both depends on operator and modifies their semantics. This is why it has a self-referential nature for operators.

One can use a list of default operations and then extend this by “lifting” them up for left/right branches up to a maximum depth level. Instead of solving at deep levels, one can repeat the solving, using a goal heuristic to pick out a promising candidate in the localized generated category.

Post-processing filtering in combination with operation composition (concatenating lists of operations) can also be used to reduce output to a small category with morphisms being proofs of transformation between objects. This information can be matched with edges along combinatorial discrete spaces.