A Comparison of Domain Specific Optimizer Implementations

Jonathan Riehl, Department of Computer Science, University of Chicago, jriehl@cs.uchicago.edu

Introduction

Domain specific languages (DSLs) present developers with a more succinct and robust means of programming solutions within a well-established problem domain. Optimization of domain specific code presents a problem for language implementors. Optimizers typically only work at the level of language primitives. Peyton-Jones et al. introduced domain specific optimizations (DSOs) as a way to allow optimization in terms of DSL abstractions [3]. Recent applications of the original DSO facility in the Glasgow Haskell Compiler (GHC) illustrate its usefulness for standard library development [1], suggesting DSOs would be a beneficial feature in other languages as well.

GHC uses term rewriting to implement DSOs. Input as pragmas to the compiler, GHC applies a user's rewrites as a part of the optimization phase. We are interested in achieving similar results as a part of building extensible languages, and as a method for rapid experimentation with optimizations in a successor language to concurrent ML. This work analyzes three available rewriting technologies in the context of an optimization framework, with the goal of understanding the trade-offs of each technology.

The optimization framework has the following characteristics:

- At the high level, an optimizer is a transformer of some intermediate representation, mapping from an intermediate representation term to an optional intermediate representation term.
- Domain specific optimizations are represented as a collection of weighted rewrites. Rewrites are a pair of operator application patterns.
- The intermediate representation (IR) is abstracted using an adapter module.
- Each operator is implemented as a SML function that accepts a collection of rewrites and an intermediate representation adapter.
- Three rewriting techniques are implemented: a greedy pattern matcher, a variant of the IBURG tree rewriting algorithm, and rewriting strategies.

Rewrites

My representation of domain specific rewrites uses a pair of intermediate representation (IR) term patterns parameterized over meta-variables.

\[ \forall s, t, \alpha = \langle \alpha \rangle \cdot t \]

The formalism also allows a weight to be associated with a rewrite. In SML, I represent rewrites with the following data structure:

```sml
datatype pattern = Op of string * pattern list | Var of int

type rewrite = { lhs : pattern, rhs : pattern, weight : int }
```

Intermediate Representation (IR)

I abstract over the intermediate representation by using an adapter structure of the following signature:

```sml
signature IR_TERM_ADAPTER = sig
type ir_term
val explode : ir_term ->
(Stamp.stamp * string option *
 ir_term list * (ir_term list ->
 ir_term))

val mk_call : (string * ir_term list) -> ir_term
end (* IR_TERM_ADAPTER *)
```

The `explode()` function decomposes an IR term into a tuple of various data. The stamp is used to uniquely identify terms. The optional string corresponds to a possible operator name that might be recognized by the IR TERM_ADAPTER. The IR TERM_ADAPTER would look at the greatest benefit non-terminal for each term, and attempts to apply any associated rewrite based on the matching grammar production.

Pattern Matching

The simplest approach uses a pattern matcher that matches an intermediate representation term with the left-hand side pattern of a rewrite rule. Unlike the other approaches, this approach requires no additional processing of the rewrite rules.

This optimizer applies the matching routine to each sub-term in a top-down visitation order. If a matching rewrite is found, the right-hand side pattern is constructed before continuing. When two or more left-hand side matches, the rewrite with the greatest weight is used. If rewrites have an identical weight, the optimizer applies the first rewrite found in the subset.

IBURG

Following the work done by Fraser, Hanson and Proebsting, the IBURG optimizer uses dynamic programming to do bottom-up tree rewriting [2]. First, the optimizer must transform the input rewrites into a weighted tree-matching grammar. This is done by flattening the left-hand side rewrite patterns into separate productions and assigning fresh non-terminals to each new production. Meta-variables are matched by a special wild-card production that matches all terms. For example, the `appendStr()` rewrite, to the upper right, would be transformed to the following grammar:

\[ S0 \rightarrow \text{ANY TERM} \]
\[ S1 \rightarrow \text{appendStr} S0 S0 (S0 \rightarrow S1) \]

In a bottom-up traversal pass, the optimizer uses the rewrite grammar to label each IR term with a map from non-terminals to aggregate benefit scores (corresponding to rewrite weights). A second top-down traversal then looks at the greatest benefit non-terminal for each term, and attempts to apply any associated rewrite based on the matching grammar production.

Strategies

Strategies are a set of term rewrite and traversal combinators that form their own domain specific language. Visser et al. have shown how these combinators can be used to implement optimizers [4]. The strategy-based optimizer translates the input rewrites into rewrite strategies, discarding rewrite weights. Using the `appendStr()` example, the following rewrite strategy would be introduced:

```sml
val appendStr =
  appendStrList([x, y, z])
```

The strategy-based optimizer compiles rewrites using the deterministic choice combinator (`<|>`). The resulting transformation strategy is applied to a top-down traversal strategy that must fail if no terns are transformed.

\[ t() = x \in \text{alltry}[t()]
\]

In order to leverage the resulting strategy, the optimizer must translate IR terms into strategy term constructors. If the optimization strategy fails, nothing (RETURN) is returned. Otherwise, the strategy term constructors are translated back to IR terms using `mk_call()`.

Conclusion

No one rewrite technique approach stands out. The pattern matching optimizer is the fastest and easiest to implement. The IBURG optimizer is best able to use the rewrite weights to find globally optimal transformations, but may not scale well as more rewrites are added. The strategy optimizer is the most flexible choice, but comes at runtime costs similar to interpretation.

References


