Prop: A C++-based Pattern Matching Language

Allen Leung*

Courant Institute of Mathematical Sciences

March 6, 1996

Abstract

In this paper we introduce Prop, a multiparadigm extension of C++ with Standard ML-style algebraic datatypes and pattern matching, tree rewriting, DATALOG-style forward chaining inference, and constraint logical programming. Applications written in Prop can utilize various cooperating formalisms, integrated into the object oriented paradigm of the base language. We use efficient automata-based and semantics based algorithms to generate various pattern matching constructs into efficient and lightweight C++ programs. Interoperability with the base language is achieved transparently since all high level data structures in Prop are mapped into classes. Furthermore, we use conservative garbage collection schemes to minimize interaction with existing code by eliminating the need for manual storage management. Typical Prop program sources are 2–10 times more compact than equivalent programs written in C++. Our benchmarks also show that programs written in Prop's high level formalisms are competitive with native C++ programs.

Keywords: *pattern matching, object-oriented programming, rewriting, semantic based optimization, compiler generation*

1 Introduction

In this paper we introduce Prop, an extension of C++[Str91, ?] that includes string matching, algebraic datatypes, Standard ML-style pattern matching[HMM86, RMH90, ?, ?], pretty printing, tree rewriting[SPvE93], DATALOG-style inference[?], constraint logic programming[?] and simple persistence as built-in features. Prop is designed as a development language for interpreters, compilers, and language translation and transformation tools[ASU86, ?]. It simplifies the construction of these systems by providing high level declarative and rule based formalisms on top of the traditional procedural and object-oriented paradigms of the base language.

We design Prop with two main objectives in mind: the first is to improve the productivity of programmers working in a mainstream imperative language — especially in domains heavy in symbolic manipulation such as interpreters, compilers, program analysis and transformation — by introducing high level formalisms based on trees and graphs, equational programming, finite set theory, and logic. By providing the users with a wide array of high level data structures and program combining forms, we encourage the use of the appropriate

*Current address: Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012-1185; +1 (212) 998-3518; leunga@valis.cs.nyu.edu

level of abstraction in each component of a language processing system. Rather than being restricted to a single imperative mode of thinking, users can utilize applicative, transformational, equational, deductive, imperative or even a combination of formalisms in a large system. For example, syntactic analysis can be performed with the parsing/lexical analysis constructs of Prop; semantic analysis with pattern matching, tree rewriting, and inference; optimization with the SETL-style sublanguage; and, finally, code generation and machine language level tools development with tree reduction with dynamic programming[?] and pattern matching with bitstring[?].

The second objective is high performance, portability and full backward compatibility with C++. To make this possible, all features are translated into C++ with a source to source translator written in Prop itself, using efficient automata-based and semantics-based algorithms. Maximal compatibility with the base language is maintained by mapping all Prop's high level data structures of directly into C++ classes. Programs written in Prop are lightweight and efficient: i.e. unneeded features are never included in the runtime system of a program, and those that are included are first transformed into interpretation free C++ code. Programs written in Prop can readily utilize existing code and libraries with little change. An optional conservative garbage collector based on [ED93, AF] can also be linked into the runtime system for Prop programs that desire automatic memory reclamation.

Prop will appeal to two main groups of users. The first are programmers working in C++ but are seeking to incorporate higher level tools without abandoning existing libraries and object oriented frameworks. The second are programmers working in other high level languages but would like to integrate with C++ to take advantage of the abundance of tools and libraries while not giving up all the high level features present in the former languages.

1.1 Related work

This section is under construction.

Source to source translation
The C language has been often used as a portable assembly language for high level languages. Examples of these range from object-oriented languages such as C++(e.g. c-front[?]); various dialects of functional languages(e.g. *sml2c*[?]), *scheme-c*[?]); and various dialects of logic programming languages(e.g. Mercury[?].)

Pattern matching, rewriting and inference

Program transformation tools

1.2 Organization of this paper

This paper is structured as follows. In section 2, we give a general overview of Prop and its programming features. In section 3 we describe the implementation of the translator. And finally, in section ?? we describe the runtime system of the language.

2 The Language

The Prop language is fashioned as a superset of C++ and contains a number of extensions. While the extension language does not provide very high level symbolic manipulation and deductive formalisms offered in modern program transformation systems, it does provide a convenient set of constructs for many common symbolic programming tasks, features that are either impossible or hard to simulate using C++'s classes or object oriented programming. The language provides:

- an algebraic datatype specification language to define the structure of user defined data;
- a grammar-based specification language that generates parsers and pretty printers that translate between text and abstract syntax; and
- rule based pattern matching, rewriting and inference constructs that manipulate and transform datatypes.

The **datatype compiler** is responsible for generating low level classes and their member functions to implement these high level datatypes. For example, it is possible to annotate an algebraic datatype to be garbage collectible, which signals the datatype compiler to automatically generate member functions that provide type feedback to the garbage collector. Similarly, a datatype defined to be persistent will have a number of member functions generated automatically to perform the task of serialization and reconstitution.

Aspects of object oriented programming have also been integrated with the datatype specification language. For example, it is possible to attach members, member functions or destructors to variants of a datatype. A datatype or a variant of a datatype may also inherit from other classes. In general, non-virtual inheritance can be seen as a sort of cartesian product while datatype variants can be seen as its dual: the sum.

Furthermore, to provide direct control of the data structure mapping process and to provide reuse of existing classes, it is possible for the user to directly specify the datatype to class mapping using the *view* mechanism. Using this mechanism, existing classes can be manipulated in an algebraic form transparently in the pattern matching constructs.

Pattern matching, rewriting, and inferences are compiled into inlined or table driven code with the **pattern matching compiler** in the translator. Since Prop began as an experiment on efficient compilation of pattern matching constructs, pattern matching code generated by the translator are very efficient, comparable to, if not surpassing, handcrafted code.

ML-style pattern matching, which is performed top-down, is compiled into a DFA-like decision tree, then transformed into inlined `switch` and `if` statements. An algorithm derived from the work on adaptive matching[?] is used in the pattern matching optimizer. Rewriting in Prop is currently performed in a bottom-up manner. Rewriting rules are gathered and compiled into a bottom-up tree automaton using algorithms described in[Cha87, ?]. Finally, inference rules are compiled into a table-driven RETE-network.

Design principles

Although the design and development of Prop have progressed in an ad hoc manner, mostly driven by need, and sometimes by passing interesting, certain design principles have been maintained. These are as follows:

- *Keep the extension language syntactically simple.* First C++ is a language with few keywords and has a complex(many would say unreadable) syntax. In extending the language, we often opt to introduce new keywords rather than taking advantage of unused combinations of existing symbols and keywords. The syntax is in many regard borrowed directly from Standard ML: the syntax for algebraic datatype definitions is almost identical. The advantage of this approach is that it is immediately clear where the extension language is being utilized.

- *Strong static typing.* Unlike languages such as *Lisp*, *SETL*, or *Prolog*, in which the high level data structures used, i.e. s-expressions, sets and Herbrand terms are essentially untyped, strong static typing is used in Prop's datatypes. This makes it possible detect many semantic errors at compile time and it also makes it possible to generate fast matching code.

- *Algebraic datatypes as unifying data.* We use algebraic datatypes as a unifying data structure in Prop. Cooperation between various high level formalisms in Prop is achieved by using algebraic datatypes as a common medium. For example, using algebraic datatypes to represent abstract syntax trees, ASTs created in the syntactic formalism can be used directly in subsequent pattern matching, rewriting, and inference phases of an application. Datatypes can be readily converted into text format using the pretty printing formalism, or marshalled into binary form and stored in an object store using the persistence capability.

Compromises

For a number of reasons, some unavoidable compromises have been made in the design and implementation of Prop:

- Since C++ is a complex language, both in syntax and semantics, it is nearly impossible to perform semantics analysis on C++ programs without extraordinary efforts. In particular, the work of a compiler frontend would have to be duplicated in order to gather enough information for analysis and transformation. We feel that the effort should be put into the efficient compilation of the extension language, so in the current version of the translator semantics information is not gathered for C++ code fragments.

- To retain maximum compatibility, even binary compatibility, with existing C++ code, libraries and applications, the code generated by Prop must be C++ obeying the usual calling convention, memory layout, etc.

  For example, although it would have been a good thing to provide first class functions in Prop, for this to be possible in a general way local variables must be transformed and packaged into closures, which would have altered the calling convention of functions. Since we have to provide binary compatibility with C++, this transformation is undesirable. Thus until a satisfactory scheme is discovered Prop cannot have first class functions.

- Finally, C++ users who are unfamiliar with higher level programming languages are typically suspicious of features that incur a cost in runtime speed or space, even if the cost is well-justified by other concerns such as programming ease. Thus in the implementation of Prop we have omitted many higher level features(such as higher order unification, Prolog style logical variables, etc), that are desirable but cannot be efficiently mapped into C++ without additional research.

2.1 Algebraic datatypes and pattern matching

Algebraic datatypes and pattern matching in Prop are borrowed from modern typed functional languages such as ML, Haskell, and Hope. In these languages, user defined datatypes are tree-like tagged variants that may be structurally decomposed using pattern matching constructs. In Prop, we can also view algebraic datatypes in the same way. However, there are a few important departures:

- Datatype values can be tested for object identity using ==, which in turn can be used to determine structural sharing. Furthermore, destructive assignments can be performed on datatypes. Thus in general datatypes in Prop are graphs(or regular trees).

- Datatypes assignment uses a reference semantics instead of the value semantics adopted in C++.
- The same datatype can be viewed in many different ways, depending on the context. For example, for general program manipulation, datatypes can be seen as simply labeled attributed trees. For rewriting, they can be viewed as ground terms. And during inference, datatypes can be seen as tuples.

In the following we shall give a brief overview of the pattern matching features of Prop. For most users of modern declarative languages many of these features are already familiar constructs.

A brief tour on pattern matching

Algebraic datatypes are specified using `datatype` definitions, which define the inductive structure of one of more types using a tree-grammar like syntax. In addition, pretty printers, lexical scanners, parsers, persistence I/O methods and garbage collection inferences can also be specified with additional options in the same construct. When a datatype is declared, the following operations are implicitly defined by the datatype compiler: (1) the constructors for all the variants of a type; (2) the identity test operator `==`, and the assignment operator `=` for this type; and (3) the member functions needed to decompose a datatype value during pattern matching.

We'll select the internals of a compiler for a simplified imperative language as the running example in this paper. Suppose that in this language an expression is composed of identifiers, integer constants and the four arithmetic operators. Then the structure of the abstract syntax tree can be specified as follows:

```
datatype Exp = INT (int)
             | ID  (const char *)
             | ADD (Exp, Exp)
             | SUB (Exp, Exp)
             | MUL (Exp, Exp)
             | DIV (Exp, Exp)
             ;
```

The abstract syntax of an expression such as *a * b - 17* can be constructed directly in a prefix syntax, directly mirroring that of the definition. The Prop datatype compiler will automatically generate a C++ class hierarchy to represent the variants of type `Exp`. Datatype constructor functions(not to be mistaken with C++'s class constructors) will also be automatically generated using the same names as the variants.

```
Exp formula = ADD(MUL(ID("a"),ID("b")),INT(17));
```

Datatype values can be decomposed using the `match` statement, which can be seen as a generalization of C's `switch` construct. Pattern matching is a combination of conditional branching and value binding. For example, a typical evaluation function for the type `Exp` can be written as in the following example. Notice that each arm of a `case` is in fact a pattern(with optional variables) mirroring the syntax of a datatype. The pattern variables(written with the prefix `?` in the sequel) of a matching arm is *bound* to the value of the matching value, which can be subsequently referenced and modified in the action of an arm.

```
int eval (Exp e, const map<const char *, int>& env)
{  match (e)
   {  case INT ?i:        return ?i;
      case ID  ?id:       return env[?id];
      case ADD (?e1,?e2): return eval(?e1,env) + eval(?e2,env);
```

```
      case SUB (?e1,?e2): return eval(?e1,env) - eval(?e2,env);
      case MUL (?e1,?e2): return eval(?e1,env) * eval(?e2,env);
      case DIV (?e1,?e2): return eval(?e1,env) / eval(?e2,env);
   }
}
```

Why object-orientedness is insufficient

Although a comparable evaluation function can be written in object oriented style using late binding, as in below, in general pattern matching is much more powerful than late binding in C++, which only allows dispatching based on the type of one receiver.

```
  // Class definitions
  class Exp {
  public:
     virtual int eval(const map<const char *, int>& env) const = 0;
  };
  class INT : Exp {
     int i;
  public:
     int eval(const map<const char *, int>& env);
  };
  class ID : Exp {
     const char * id
  public:
     int eval(const map<const char *, int>& env);
  };
  ...

  // Member functions
  int INT::eval(const map<const char *, int>& env) const { return i; }
  int ID ::eval(const map<const char *, int>& env) const { return id; }
  int ADD::eval(const map<const char *, int>& env) const
     { return e1->eval(env) + e2->eval(env); }
  int SUB::eval(const map<const char *, int>& env) const
     { return e1->eval(env) - e2->eval(env); }
  int MUL::eval(const map<const char *, int>& env) const
     { return e1->eval(env) * e2->eval(env); }
  int DIV::eval(const map<const char *, int>& env) const
     { return e1->eval(env) / e2->eval(env); }
```

For example, in the following function we use nested patterns, non-linear patterns (i.e. patterns with multiple occurrences of a pattern variable), and guards to perform algebraic simplification of an expression. Although the patterns are relative simple in this example, in general arbitrarily complex patterns may be used.

```
Exp simplify (Exp redex)
{  // recursive traversal code omitted ...

   // match while repeats the matching process
```

```
    // until no more matches are found.
    match while (redex)
    {  ADD(INT 0,  ?x):     { redex = ?x; }
    |  ADD(INT ?x, INT ?y): { redex = INT(?x+?y); }
    |  ADD(?x,     INT 0)   { redex = ?x; }
    |  SUB(?x,     INT 0):  { redex = ?x; }
    |  SUB(?x,     ?x):     { redex = INT(0); }
    |  SUB(INT ?x, INT ?y): { redex = INT(?x-?y); }
    |  MUL(INT 0,  ?x):     { redex = INT(0); }
    |  MUL(?x,     INT 0):  { redex = INT(0); }
    |  DIV(?x,     ?x):     { redex = INT(1); }
         // don't divide by zero.
    |  DIV(INT ?x, INT ?y) | ?y != 0: { redex = INT(?x/?y); }
    |  ...
    }
    return redex;
}
```

Pattern matching in Prop is also not restricted to one datatype at a time. In the following example, we use matching on multiple values to define equality on expressions inductively. For variety, we'll use the `fun` variant of `match`, which defines a function in rule form. Notice that the last case of the match set uses *wild cards* `_` to catch all the other non-equal combinations. Since C++ does not provide multiple dispatching, implementing binary (or n-ary) operations on variant datatypes are in general cumbersome and verbose in object-oriented style. In contrast, using an applicative pattern matching style many manipulations and transformations on variant datatypes with tree-like or graph-like structure can be expressed succinctly.

```
fun equal INT ?i,     INT ?j: bool: { return ?i == ?j; }
  | equal ID  ?a,     ID  ?b:       { return strcmp(a,b) == 0; }
  | equal ADD(?a,?b), ADD(?c,?d):   { return equal(?a,?c) && equal(?b,?d); }
  | equal SUB(?a,?b), SUB(?c,?d):   { return equal(?a,?c) && equal(?b,?d); }
  | equal MUL(?a,?b), MUL(?c,?d):   { return equal(?a,?c) && equal(?b,?d); }
  | equal DIV(?a,?b), DIV(?c,?d):   { return equal(?a,?c) && equal(?b,?d); }
  | equal _,          _:            { return false; }
  ;
```

More examples

As another example, we can specify the term structure of *well-formed formulas* in proposition calculus as follows. Notice that the constructors `F` and `T` are nullary.

```
   datatype Wff = F
                | T
                | Var     (const char *)
                | And     (Wff, Wff)
                | Or      (Wff, Wff)
                | Not     (Wff)
                | Implies (Wff, Wff)
                ;
```

Datatypes that are parametrically polymorphic, such as lists and trees, can be defined by parameterizing them with respect to one of more types. For example, both lists and tree below are parametric on one type argument `T`.

```
datatype List<T> = nil
                 | cons(T, List<T>);
datatype Tree<T> = empty
                 | leaf(T)
                 | node(Tree<T>, T, Tree<T>);

List<int> primes = cons(2,cons(3,cons(5,cons(7,nil))));
List<int> more_primes = cons(11,cons(13,primes));
Tree<char *> names = node(leaf("Church"),"Godel",empty);
```

As a programming convenience, Prop has a set of built-in list-like constructors syntactic forms. Unlike languages such as ML, however, these forms are not bound to any specific list types. Instead, it is possible for the user to use these forms on any datatypes with a natural binary *cons* operator and a nullary *nil* constructor. For instance, the previous list datatype can be redefined as follows:

```
datatype List<T> = #[] | #[ T ... List<T> ];
List<int> primes = #[ 2, 3, 5, 7 ];
List<int> more_primes = #[ 11, 13 ... primes ];
List<char *> names = #[ "Church", "Godel", "Turing", "Curry" ];

template <class T>
   List<T> append (List<T> a, List<T> b)
   {  match (a)
      {  case #[]:          return b;
         case #[hd ... tl]: return #[hd ... append(tl,b)];
      }
   }
```

Notice that the empty list is written as `#[]`, while *cons(a,b)* is written as `#[ a ... b ]`. An expression of the special form `#[a, b, c]`, for instance, is simple syntactic sugar for repeated application of the *cons* operator, i.e.

```
#[a, b, c] == #[a ... #[ b ... #[ c ... #[] ] ] ].
```

List-like special forms are not limited to datatypes with only two variants. For example, we can define a datatype similar in structure to S-expressions in *Lisp* or *Scheme*. Here's how such as datatype may be defined(for simplicity, we'll use a string representation for atoms instead of a more efficient method):

```
datatype Sexpr = INT    (int)
               | REAL   (double)
               | STRING (char *)
               | ATOM   (const char *)
               | #()
               | #( Sexpr ... Sexpr )
where type Atom = Sexpr  // synonym for Sexpr
;
```

With this datatype specification in place, we can construct values of type `Sexpr` in a syntax close to that of *Lisp*. For example, we can define lambda expressions corresponding to the combinators I, K and S as follows:

```
Atom LAMBDA = ATOM("LAMBDA");
Atom f      = ATOM("f");
Atom x      = ATOM("x");
Atom y      = ATOM("y");
Atom NIL    = #();

Sexpr I = #(LAMBDA, #(x), x);
Sepxr K = #(LAMBDA, #(x), #(LAMBDA, #(y), x));
Sepxr S = #(LAMBDA, #(f),
             #(LAMBDA, #(x),
                #(LAMBDA, #(y), #(#(f,x), #(g,x)))));
```

Similar to list-like forms, vector-like forms are also available. This addresses one of the flaws of the C++ language, which lacks first class arrays. Vectors are simply homogeneous arrays whose sizes are fixed and are determined at creation time. Random access within vectors can be done in constant time. Unlike lists, however, the prepending operation is not supported. Vectors literals are delimited with the composite brackets `(| ... |)`, `[| ... |]`, or `{| ... |}`. In the following example the datatype `Exp` uses vectors to represent the coefficients of the polynomials:

```
datatype Vector<T> = (| T |);
datatype Exp = Polynomial (Var, Vector<int>)
             | Functor (Exp, Vector<Exp>)
             | Atom (Var)
             | ...
where type Var = const char *;
Exp formula = Polynomial("X", (| 1, 2, 3 |));
```

Pattern laws

Commonly used patterns can be given synonyms so that they can be readily reused without undue repetition. This can be accomplished by defining pseudo datatype constructors to stand for common patterns using *datatype law* definitions. For example, the following set of laws define some commonly used special forms for a *Lisp*-like language using the previously defined `Sexpr` datatype.

```
datatype law Lambda(x,e)  = #(ATOM "LAMBDA", #(x), e)
           | Quote(x)     = #(ATOM "QUOTE", x)
           | If(a,b,c)    = #(ATOM "IF", a, b, c)
           | Nil          = #()
           | ProgN(exprs) = #(ATOM "PROGN" ... exprs)
           | SpecialForm  = #(ATOM ("LAMBDA" || "IF" ||
                                    "PROGN" || "QUOTE") ... _)
           | Call(f,args) = ! SpecialForm && #(f ... args)
           ;
```

Notice that the pattern `SpecialForm` is meant to match all special forms in our toy language: i.e. *lambdas*, *ifs*, *progn*'s and *quotes*. The pattern disjunction connective `||` is used to link these forms together. Since

we'd like the `Call` pattern to match only if the S-expression is not a special form, we use the pattern negation and conjunction operators, `!` and `&&` are used to screen out special forms. With these definitions in place, an interpreter for our language can be written thus:

```
Sexpr eval (Sexpr e)
{  match (e)
   {  Call(?f,?args):     { /* perform function call */ }
   |  Lambda(?x,?e):      { /* construct closure */ }
   |  If(?e,?then,?else): { /* branching */ }
   |  Quote(?x):          { return ?x; }
   |  ...:                { /* others */ }
   }
}
```

As an interesting note, the special form pattern can also be rewritten using regular expression string matching, as in the following:

```
datatype law SpecialForm = #(ATOM /LAMBDA|IF|PROGN|QUOTE/ ... _)
```

Variants of match

Besides the usual plain pattern matching, a few variants of the `match` construct are offered. We'll briefly enumerate a few of these:

- The `matchall` construct is a variant of `match` that executes all matching rules(instead of just the first one) in sequence.
- Each rule of a `match` statement can have associated cost expressions. Instead of selecting the first matching rule to execute as in the default, all matching rules are considered and the rule with the least cost is executed. Ties are broken by choosing the rule that comes first lexically. For example:

  ```
  match (ir)
  {  ADD(LOAD(?r0),?r1) \ e1: { ... }
  |  ADD(?r0,LOAD(?r0)) \ e2: { ... }
  |  ADD(?r0, ?r1)      \ e3: { ... }
  |  ...
  }
  ```

- A `match` construct can be modified with the `while` modifier, as in the following example. A match modified thus is repeatedly matched until none of the patterns are applicable. For example, the following routine uses a `match while` statement to traverse to the leftmost leaf.

  ```
  template <class T>
  Tree<T> left_most_leaf(Tree<T> tree)
  {  match while (tree)
     {  case node(t,_,_):  tree = t;
     }
     return tree;
  }
  ```

- Finally, the `matchscan` variant of match can be used to perform string matching on a stream. For example, a simple lexical scanner can be written as follows.

```
int lexer (istream& in)
{  matchscan while (in)
   {  /if/:                      { return IF; }
   |  /else/:                    { return ELSE; }
   |  /while/:                   { return WHILE; }
   |  /for/:                     { return FOR; }
   |  /break/:                   { return BREAK; }
   |  /[0-9]+/:                  { return INTEGER; }
   |  /[a-zA-Z_][a-zA-Z_0-9]*/: { return IDENTIFIER; }
   |  /[ \t]+/:                  { /* skip spaces */ }
   |  ... // other rules
   }
}
```

2.2 Tree rewriting

While plain pattern matching described in the previous section is adequate for more complex program manipulation involving tree- or graph-like data structures, higher level constructs such as rewriting and inference are also available. In the rewriting formalism, equational rules of the form *lhs* `->` *rhs* are specified by the user. During processing, each instance of the lhs in a complex tree is replaced by an instance of the rhs, until no such replacement is possible. Equational rules can often be used to specify semantics based simplification (e.g. constant folding and simplification based on simple algebraic identities) or transformation(e.g. code selection in a compiler backend[AGT89]).

Unlike plain pattern matching, however, the structural traversal process in rewriting is implicitly inferred from the type structure of an algebraic datatype, as specified in its definition. Thus when changing the structure of a datatype, unaffected patterns in rewriting rules do not have to be altered.

There are two main forms of rewriting modes available:

- The first is **normalization** mode: a given tree is reduced using the matching rules until no more redexes are available. There are two modes of operations available:
  - in **replacement** mode, the redex of a tree will be physically overwritten.
  - in **applicative** mode, on the other hand, a new tree corresponding to the replacement value will be constructed.

  Replacement mode is used as the default since it is usually the more efficient of the two.
- The second form is **reduction** and **transformation**. In this mode a tree parse of the input term is computed. If cost functions are attached to the rules, then they will also be used to determine a minimal cost reduction sequence. During this process attached actions of a rule may be invoked to synthesize new data.

Rewrite class

Each independent set of rewriting rules in Prop is encapsulated in its own **rewrite class**. A rewrite class is basically a normal C++ class with a set of rewriting rules attached. During rewriting, the data members and the member functions are visible according to the normal C++ scoping rules. This makes it is easy to encapsulate additional data computed as a side effect during the rewriting process.

A rewriting example

Consider an abbreviated simplifier for the well-formed formula datatype `Wff` defined in the previous section. The rewrite class for this can be defined as follows. Since there is no encapsulated data in this example, only the default constructor for the class needs to be defined. A rewrite class definition requires the rewriting protocol, which is simply a list of datatypes involved in the rewriting traversal process, to be specified. In this instance only `Wff` is needed.

```
rewrite class Simplify (Wff)
{
public:
   Simplify() {}
};
```

The rewrite rules for the simplifier can then be specified succinctly as follows. Like the `match` statement, in general the rhs of a rewrite rule can be any statement. A special statement `rewrite(e)` can be used to rewrite the current redex into another form. If the rhs is of the form `rewrite(e)`, then it can be abbreviated to `e`, as in below:

```
rewrite Simplify
{  And(F,  _):      F
|  And(_,  F):      F
|  And(T,  ?X):     ?X
|  And(?X, T):      ?X
|  Or (T,  _):      T
|  Or (_,  T):      T
|  Or (F,  ?X):     ?X
|  Or (?X, F):      ?X
|  Not(Not(?X)):    ?X
|  Not(And(?X,?Y)): Or(Not(?X), Not(?Y))
|  Not(Or(?X,?Y)):  And(Not(?X), Not(?Y))
|  Implies(?X,?Y):  Or(Not(?X), ?Y)
|  And (?X, ?X):     ?X
|  Or (?X, ?X):      ?X
|  Implies (?X, ?X): ?X
// etc ...
};
```

The rewrite class definition creates a new class of the same name. This new class defines an implicit `operator ()` with the protocol below. This member function can be invoked to perform the rewriting in a functional syntax.

```
class Simplify : ... {
{  ...
```

```
public:
   void operator () (Wff);
   // Wff operator () (Wff); // if rewrite class is applicative
};

Wff wff = ...;
Simplify simplify; // create a new instance of the rewrite class
simplify(wff);    // rewrite the term wff
```

State caching

Replacements during rewriting often require state information to be recomputed to further the matching process. Since computation of state encoding can involve a complete traversal of a term, replacement can become expensive if the replacement term is large. For instance, consider the following replacement rule, which replaces all expressions of the form *2*x* into *x+x*:

```
rewrite class StrengthReduction
{
   MUL (INT 2, ?x):  ADD(?x, ?x)
   ...
}
```

Since the subterm `?x` could be arbitrarily large, recomputing the state encoding for `ADD(?x,?x)` takes time in proportion to the size of `?x`. In order to speedup this replacement process, state encoding caching can be enabled, which in the example above means that the state encoding for `ADD(?x,?x)` can be recomputed directly from the state encoding of `?x`. State caching is enabled by adding a `rewrite` qualifiers in the definition of a datatype, as in:

```
datatype Exp :: rewrite
   = INT (int)
   | ID  (const char *)
   | ADD (Exp, Exp)
   | SUB (Exp, Exp)
   | MUL (Exp, Exp)
   | DIV (Exp, Exp)
   ;
```

Conditional rewriting and actions

Rewriting rules may be guarded with predicates to limit their applicability. In addition, the *rhs* of a rewrite rule is not limited to only a replacement expression: in general, any arbitrarily complex sequence of code may be used. For example, in the following set of rewriting rules we use guards to prevent undesirable replacements to be made during expression constant folding:

```
rewrite class ConstantFolding
{  ADD (INT a, INT b):  INT(a+b)
|  SUB (INT a, INT b):  INT(a-b)
```

```
|  MUL (INT a, INT b):
   {  int c = a * b;                      // silent overflow
      if (a == 0 || b == 0 || c / b == a) // no overflow?
      {  rewrite(INT(c)); }
      else
      {  cerr << "Overflow in multiply\n"; }
   }
|  DIV (INT a, INT b) | b == 0:  { cerr << "Division by zero\n"; }
|  DIV (INT a, INT b):  INT(a/b)
|  // etc...
};
```

The `rewrite` statement

While the rewrite class construct provides a very general abstraction for rewriting, in general its full power is unneeded. It is often convenient to be able to perform rewriting on a term without having to make a new name for a class just for the occasion, especially if member functions and member data are unneeded. To accommodate these situations, the **`rewrite`** statement is provided to perform a set rewriting transformations on a term without having to define a temporary rewrite class. It is simply syntactic sugar for the more general(but cumbersome) rewrite class and rewrite rules specifications. For example, a simplify routine for type `Exp` defined above can be specified as follows:

```
Exp simplify (Exp e)
{  // transformations on e before
   rewrite (e) type (Exp)
   {  ADD (INT a, INT b):  INT(a+b)
   |  SUB (INT a, INT b):  INT(a-b)
   |  MUL (INT a, INT b):  INT(a*b)
   |  ...
   }
   // transformations on e after
   return e;
}
```

The `rewrite` normally performs the replacement in place. An applicative version of the same can be written as follows[1]:

```
Exp simplify (Exp e)
{  rewrite (e) => e type (Exp)
   {  ADD (INT a, INT b):  INT(a+b)
   |  SUB (INT a, INT b):  INT(a+b)
   |  MUL (INT a, INT b):  INT(a*b)
   |  ...
   }
   return e;
}
```

[1] The variable e is assigned the new copy.

Confluence and termination
This section is under construction.

Commutivity and associativity
This section is under construction.

2.3 Inference

Semantic processing, such as data flow analysis, in compilers and other language processors can frequently be specified as in a rule-based, logical deductive style. In Prop, deductive inference using forward chaining is provided as a built-in mechanism, orthogonal to pattern matching and rewriting, for manipulating user-defined algebraic datatypes.

Similar to rewriting classes, **inference classes** may be used for data encapsulation. An inference class is a combination of a C++ class, a database of inference relations, and a collection of inference rules of the form *lhs* `->` *rhs*. The lhs of an inference rule is a set of patterns in conjunctive form. During the inference process, a rule is fired when its lhs condition is satisfied. A fired rule then executes the corresponding rhs action, which may assert or retract additional relations from the database. Using multiple inheritance, it is possible to combine a rewriting class with an inference class such that the rewriting process generates new relations to drive the inference process, or vice versa.

Datatype relations are not a distinct kind of data structure but are in fact simply algebraic datatypes declared to be such. For example, in the following definition three relation types `Person`, `Parent` and `Gen` are defined.

```
datatype Person :: relation = person (const char *)
    and  Parent :: relation = parent (const char *, const char *)
    and  Gen    :: relation = same_generation (const char *, const char *);

instantiate datatype Person, Parent, Gen;
```

Using these relations we can define an inference class that computes whether two persons are in the same generation. Nine axioms are defined (i.e. those whose lhs are empty) in the following. The two inference rules state that (1) the same person is the same generation, and (2) two persons are in the same generation if their parents are in the same generation.

```
inference class SameGeneration {};

inference SameGeneration
{  -> person("p1") and person("p2") and
      person("p3") and person("p4") and
      person("p5");

   -> parent("p1", "p2") and
      parent("p1", "p3") and
      parent("p2", "p4") and
      parent("p3", "P5");

   person ?p -> same_generation (?p, ?p);
```

```
    parent (?x, ?y) and parent (?z, ?w) and same_generation (?x, ?z)
    -> same_generation(?y, ?w);
};
```

In general, datatypes qualified as `relations` will inherit from the base class `Fact`, while a rewrite class definition implicitly defines two member functions used to assert and retract facts in the internal database. For example, in the above example, the following protocol will be automatically generated by the inference compiler.

```
class SameGeneration : ...
{
public:
    virtual Rete&    infer       ();       // start the inference process
    virtual ReteNet& operator += (Fact *); // assert fact
    virtual ReteNet& operator -= (Fact *); // retract fact
};
```

Using these methods, an application can insert or remove relations from an inference class. This will in turn trigger any attached inference rules of the class.

Another example

Consider the following example, which is used to compute Pythagorean triangles. Only one axiom and two rules are used. The axiom and the first rule are used to assert the relations `num(1)` to `num(n)` into the database, where `n` is limited by the term `limit(n)`. The second inference rule is responsible for printing out only the appropriate combinations of numbers.

```
datatype Number :: relation = num int | limit int;

inference class Triangle {};

inference Triangle
{  ->  num 1;

       num m
   and limit n | n > m
   ->  num (m+1);

       num a
   and num b
   and num c | a < b && b < c && a*a + b*b == c*c
   ->  { cout << a << " * " << a << " + "
              << b << " * " << b << " = "
              << c << " * " << c << "\n";
       };
};
```

Now, to print all the triangle identities lying in range of 1 to 100, we only have to create an instance of the inference class, insert the limit, and start the inference process, as in below:

```
Triangle triangle;
triangle += limit(100);
triangle.infer();
```

Operational semantics of inference
This section is under construction.

Safe negation
This section is under construction.

Indexing and index specifications
This section is under construction.

2.4 Syntactic formalisms

Language processors frequently have to manipulate the language in the source level. Instead of depending on external tools such as parser generators and lexer generators, parsing and lexical analysis constructs are integrated with the Prop language in the form of **syntax classes** and **lexer classes**.

In addition, higher level syntactic formalisms are also available. Some of these are:

- *Lisp*-like meta quoting. Using meta-quoting, programs can manipulate a language in its source level syntax. Translation into the abstract syntax is done automatically by the translator. This makes it possible to express complex program transformations using a succinct syntax.
- *Pretty printing.* Algebraic datatypes may specify their pretty printed format in a declarative manner.

Lexical analysis

The **scanner class** mechanism in Prop provides functionalities similar to what is offered in modern lexical analyzer generators such as *lex*[Les75], *flex*[Pax90], and *dlg*[PDC91] etc. Instances of scanner classes are simply lexical scanners (with possibly additional encapsulated states) taking input from a I/O stream and generating a stream of lexemes.

Unit datatypes (i.e. algebraic datatypes with only non-argument taking variants) can be used to stand for lexemes generated by a lexical scanner. These lexemes are also used as terminals inside a syntax definition. Optional string and/or regular expression patterns may be attached to a lexeme, which can used to specialize a lexical scanner inside a scanner class. For example, the following datatype definition defines a set of lexemes for the C language.

```
datatype C_Token :: lexeme
   = IF        "if"
   | ELSE      "else"
   | FOR       "for"
   | WHILE     "while"
   | GOTO      "goto"
   | CONTINUE  "continue"
   | BREAK     "break"
```

```
    | RETURN    "return"
    | ID        /[a-zA-Z_][a-zA-Z_0-9]*/
    | INTEGER   /[0-9]+/
    | STRING    /"([^"\n]|\\.)"/
    | CHARACTER /´([^´\n]|\\.)´/
    | ...  // etc.
    ;
```

In the above, we use the constructor `IF` stands for the lexeme `"if"`, `ELSE` to stand for `"else"`, etc. However, it is frequently more convenient to simply use the same name for the abstract and the printed representation for keywords. We allow the user to abbreviate the lexeme constructors definitions by specifying only the printed representations. For example, we may rewrite the previous datatype definition as follows:

```
datatype C_Token :: lexeme
   = "if" | "else" | "for" | "while" | "goto"
   | "continue" | "break" | "switch" | "case"
   | "return" | "default" | "struct" | "union"
   | "register" | "volatile" | "extern" | "static"
   | ID        /[a-zA-Z_][a-zA-Z_0-9]*/
   | INTEGER   /[0-9]+/
   | STRING    /"([^"\n]|\\.)"/
   | CHARACTER /´([^´\n]|\\.)´/
   | ...  // etc.
   ;
```

Now, to refer to the constructor with the printing representation `"if"`, `T"if"` can be used. In general, `T"???"` refers to the token with a printed representation of `"???"`.

Lexeme abbreviations

Complex regular expressions can be built from simple ones using **lexeme aliases**. These aliases are defined within the lexeme alias definitions. For example, complex regular expressions standing for integers, reals and friends can be defined by composition as follows. Aliases appearing in regular expressions are quoted by braces, a syntactic convention inherited from *yacc*-like tools:

```
lexeme digits   = /[0-9]+/
     | sign     = /[\-\+]/
     | integer  = /{sign}?{digits}/
     | exponent = /[eE]{integer}/
     | fraction = /\.{digits}/
     | mantissa = /{sign}?({fraction}|{digits}{fraction}?)/
     | real     = /{mantissa}{exponent}?/
     ;
```

Lexeme aliases can be reused inside any regular expressions, including patterns inside lexeme datatype definitions.

Lexeme classes

Parser construction

The **syntax class** mechanism in Prop is the functional equivalent of popular parser generators such as *yacc*[?], or *bison*[?]. A syntax class is used to encapsulate the state of a parser for one grammar. Grammar rules are specified in a BNF-like form similar to that of *yacc*. Prop currently generates table driven LALR(1) parsers. Shift/reduce conflicts can be resolved using optional operator precedence information, or with optional **semantics predicates** similar to that in *PCCTS*[PDC91]. Actions may be attached to a grammar, and during parsing, these actions have access to the member data and functions, and inherited and synthesized attributes.

Consider the following partial parser specification for a simple language.

```
syntax class SmallLang
{
public:
   SmallLang(istream&) : Super(istream&) {}
  ~SmallLang() {}
};
```

The production rules of the language are encapsulated in a syntax definition, which also contains the precedence and associativity rules for operators. Notice that terminals may be a single character or a predefined lexeme datatype.

```
syntax SmallLang
{  left: 1 '*' '/';
   left: 2 '+' '-';

   stmt(Stmt): IF expr THEN stmt ELSE stmt ';' { $$ = IFstmt($2,$4,$6); }
             | WHILE expr DO stmt DONE ';'     { $$ = WHILEstmt($2,$4); }
             | expr '=' expr ';'               { $$ = ASSIGNstmt($1,$3); }
             | BEGIN stmt_list END ';'         { $$ = BLOCKstmt($2); }
             ;
   expr(Exp): integer       { $$ = INT($1);  }
            | ident         { $$ = ID($1);    }
            | expr '+' expr { $$ = ADD($1,$3); }
            | expr '-' expr { $$ = SUB($1,$3); }
            | expr '*' expr { $$ = MUL($1,$3); }
            | expr '/' expr { $$ = DIV($1,$3); }
            | '-' expr      { $$ = UMINUS($2); }
            ;
   integer(int): INTEGER    { $$ = atol(lexer.text()); };
   id(int):      IDENTIFIER { $$ = strdup(lexer.text()); };
};
```

It is often more convenient to specify the terminals (lexemes) in their printed form rather than in their encoded form. As a short hand, the printed form of a lexeme can be used in place of the lexeme itself inside a syntax specification. For example, the statement productions can be specified alternatively as follows:

```
   stmt(Stmt): "if" expr "then" stmt "else" stmt ';' { $$ = IFstmt($2,$4,$6); }
```

```
          | "while" expr "do" stmt "done" ´;´     { $$ = WHILEstmt($2,$4); }
          | expr ´=´ expr ´;´                     { $$ = ASSIGNstmt($1,$3);}
          | "begin" stmt_list "end" ´;´           { $$ = BLOCKstmt($2); }
          ;
```

The structure of the abstract syntax tree can be specified as follows:

```
datatype Stmt = IFstmt(Exp,Stmt,Stmt)
              | WHILEstmt(Exp,Stmt)
              | ASSIGNstmt(Exp,Exp)
              | BLOCKstmt(List<Stmt>)
and      Exp = INT (int)
             | ID  (const char *)
             | UMINUS (Exp)
             | ADD (Exp, Exp)
             | SUB (Exp, Exp)
             | MUL (Exp, Exp)
             | DIV (Exp, Exp)
and List<T>  = #[] | #[ T ... List<T> ]
             ;
```

Inherited and synthesized attributes

Semantic predicates

Real life programming languages commonly have non-context free fragments that make it difficult to fit into a parser generator's framework. For example in C, the code fragment

```
T * U;
```

can be either a variable declaration if `T` is a type identifier, or an expression if `T` is a variable. To deal with these types of irregularities, parsers, lexers and the semantic analysis routines frequently have to cooperate and provide feedback to each other during the parsing process. Frequently this involves the insertion of non-declarative actions inside the parser or lexer specifications.

We borrow a new construct, the **semantic predicate**, which allows a clean separation of these context sensitive feedback, from the parser generator tool *PCCTS*[PDC91] Aside from terminals, non-terminals and semantic actions, a production can also contain semantic predicates. During parsing, these semantic predicates are evaluated, and productions which contain unsatisfied predicates will be rejected.

For example, to deal with the type/variable identifier ambiguity in a language like C, the following productions are needed.

```
Stmt:  id (is_type_id($1)) ´*´ id ´;´ // variable definition
    |  id ´*´ id ´;´                  // otherwise, an expression
```

Pretty printing

Pretty printing of datatypes in an external format is a frequently needed operation and Prop also provide a set of formalism to allow rapid specification of pretty printers. By default, if the `printable` qualifier is used within the definition of a datatype, a pretty printer will be automatically generated to print a datatype value in Prop syntax. For example, if the expression type `Exp` is defined as:

```
datatype Exp :: printable = ...
```

the pretty printer method

```
ostream& operator << (ostream&, Exp);
```

will be automatically generated. This pretty printer can be used as a debugging aid, for example, during rapid prototyping.

To get more refined output, the user can attach datatypes with pretty printing formats, which are simply specifications of how a datatype can be linearized for printing. Printing format strings are attached to each constructor after the separator `=>` within a datatype definition. For example, in the following we specify that integers and identifiers in the datatype `Exp` are to be printed as-is, while the binary arithimetic expressions are to be printed with surrounding parentheses:

```
datatype Exp = INT (int)          => _
             | ID  (const char *) => _
             | ADD (Exp, Exp)     => "(" _ " + " _ ")"
             | SUB (Exp, Exp)     => "(" _ " - " _ ")"
             | MUL (Exp, Exp)     => "(" _ " * " _ ")"
             | DIV (Exp, Exp)     => "(" _ " / " _ ")"
             ;
```

The underscore character `_` above is an example of a **meta format character**. Its meaning is to print the next available argument of the constructor using whatever format appropriate for its type. Other meta format characters are also available, which deal with issues such as argument ordering, indention and parenthesization, etc. Please see figure 1 for a detailed listing.

{ and } — nest scope and indent.

/ — newline and indent.

_ — print the next component.

n — print the *n*th component in a tuple.

id — print the record component labeled *id*.

Figure 1: Meta pretty printing formats.

As another example, consider the following datatype definition of an AST for statements. The meta-characters `{` and `}` are used to indent statements occurring in nested scopes:

```
datatype Stmt =
   IF { cond : Exp,
```

```
        yes  : Stmt,
        no   : Stmt
      }              => "if" cond "then" { yes } "else" { no } "endif;"
|  WHILE (Exp, Stmt)  => "while" _ "do" { _ } "end do;"
|  ASSIGN (Id, Exp)   => _ ":=" _ ";"
|  BLOCK (List<Stmt>) => "begin" { _ } "end;"
```

Thus using this definition, the datatype expression

```
WHILE(ID("x"), #[ ASSIGN("x",SUB(ID("x"),INT(1))),
                  ASSIGN("y",MUL(ID("y"),ID("x"))) ]);
```

is pretty printed as

```
while x do
   begin
      x := x - 1;
      y := y * x;
   end;
end do;
```

Generally speaking, the pretty printing generation capability of Prop is meant to be used for simple data structures. Complex printing mechanisms can be handled using the pattern matching or rewriting constructs.

2.5 Meta syntax

This section is under construction.

2.6 Persistence

While the syntax and meta-syntax formalisms of Prop are involved with transforming data structures to and from a textual external form, the persistence formalism is responsible for transformation to and from an binary form.

2.7 Logical Variables, Feature Trees and Constraints

Logical variables
A new extension of Prop involve the extension of **logical variables**. With the introduction of logical variables, tree structures formed from algebraic datatypes become Herbrand terms, pattern matching becomes unification, and rewriting becomes narrowing.

Constraints

Feature trees

2.8 Paradigm composition

While a multiparadigm language is often attractive in concept, to make it practical there must be a consistent method for combining different paradigms in an application. Depending on the paradigms in question, this may have a large effect on the feasibility of this mixing. In general, the main difficulties are: (1) combining the **execution models**, and (2) combining the **data structures**, introduced in various constructs.

In Prop, this challenge of mixing paradigms is met in a number of ways:

- First, algebraic datatypes (and extensions thereof) are used as the uniform data structure in all the pattern matching constructs.
- Secondly, different pattern matching constructs are encapsulated in an object oriented manner using the class structure of C++. Using virtual functions (i.e. late binding) and multiple inheritance, different pattern matching constructs, such as rewriting and inference, can be seamlessly combined.

3 Implementation

In this section we'll briefly describe some of the more interesting implementation techniques we have used and some of the interesting problems that we have encountered.

Data structure mapping

Pattern matching and rewriting

ML-style pattern matching is compiled by first constructing an automaton using an algorithm similar to that of [?]. The automaton is then translated into C++ code directly. Recently, Ramakrishnan et al.[?] proposed an adaptive algorithm which, instead of using a fixed left-to-right matching order, derives an traversal order for each set of patterns. Unfortunately, this algorithm is $co-NP$ for typed patterns, so instead we use a few heuristics proposed in the paper to select the traversal order.

Rewriting in Prop is compiled into bottom-up tree automata[HO82] with an index map compression heuristic[Cha87]. Large index maps are also compressed on a secondary level using [?]'s algorithm for trie compression. BURS-like tree automata[?] are generated for tree automata with fixed reduction cost, while tree automata with runtime determined cost functions are compiled into code that performs cost minimization using dynamic programming. Table lookup optimizations are then performed by encoding compile time determinable lookups into direct `switch` statements.

For typical rewriting systems with less than 500 rules, the rewriting compiler is able to generate the tree automaton in less than 5 seconds. A simple benchmark of computing the 25th Fibonacci number using in place rewriting with a naive exponential algorithm takes 3.29 seconds (GC time .17 sec) of process time on a Sparc 5 workstation, including instrumentation and garbage collection overhead. This benchmark performs 1335317 matches and 364177 replacements. Since rewriting speed is largely independent of the size of the pattern sets this shows that rewriting in Prop can proceed at the speed of over 400,000 matches and over 110,000 replacements per second on typical machines.

Inference

Inference rules in Prop are compiled into a data flow network, then flattened into a table form. During runtime an interpreter using the RETE algorithm is invoked to process the network and dispatch to various matching routines. Management of the α and β memory and token propagation is handled within the interpreter object, thus user management code is not needed.

To speed up the pattern matching process, the **inference rule compiler** is responsible for performing several optimizations:

- Partition the antecedents of the rules into *selects*(single object tests) and *joins*(predicates involving multiple objects). Complex predicates involving multiple clauses are decomposed into simple tests when possible. Antecedents involving tree patterns are then compiled by invoking the pattern matching compiler.
- Simple syntactic based transformations are performed; these include transforming boolean expressions into conjunctive form and pushing down selects.

Garbage collection

We use a conservative garbage collector whose framework is based on the work of Customisible Memory Management[AF] in the PoSSe algebraic system. This scheme is in turns based on the work on mostly-copying garbage collection by Bartlett[Bar88] in the Scheme to C runtime system. Similar to CMM, garbage collectable objects in Prop are all derived from the base class `GCObject`. Each subclass of `GCObject` reimplements the virtual function `trace(GC *)`, which is responsible for providing type feedback by traversing the pointers within the object and calling a garbage collector method for each pointer found. Garbage collector objects are implemented as *visitors*[GHJV95]. This object tracing method is generated automatically for each user defined datatype by the datatype compiler.

The collectors can discover the current root set by scanning the stack, registers, static data and heap areas. Thus no root registration or inefficient smart pointer schemes[Ede92] are necessary: a pointer to a collectable object looks the same as any other pointers and can be placed inside machines registers or otherwise optimized by the compiler in the same manner.

By detaching the traversal method from the specific implementation of the garbage collectors, we are able to implement various garbage collection algorithms that can coexist with each other. Two main collection algorithms have been implemented:

- A **mostly-copying** algorithm based on [Bar88].
- A non-moving **mark-sweep** style collector.

The copying collector is used as the default in Prop since, unlike C or C++ programs, Prop programs written in an applicative style typically generate more short term objects than an equivalent C or C++ program.

Both collectors use the same abstract protocol to communicate with collectable objects, and because of late binding, the actual collection algorithm is determined at runtime. This makes it possible for the user to experiment with various collection algorithms without recompilation.

In addition, the following features have been implemented:

- *Interior pointers.* Pointers can point to the interior of an object. Due to pointer arithmetic in C and multiple inheritance and upcasting in C++, recognizing interior pointers is necessary for collection safety.
- *Multiple heaps and cross heap pointers.* Multiple collectable and non-collectable heaps can coexist at runtime. This makes it possible to, say, use a copying heap to store terms created by rewriting(which

are typically short lived) while using the mark-sweep collector to storage objects with longer life times(in which case it is less expensive to simply to mark the object during collection than moving it to another space). Furthermore, pointers from an object can refer to objects from other heaps.

- *Finalization.* Each collectable heap can be declared to be finalizable or non-finalizable. The C++ destructors are called for each collectable object when its storage is reclaimed[2] And finally,
- *Weak pointers to collectable heaps.*

Some optimizations

We have implemented a few optimizations in the collector. The most important of these is a variant of **blacklisting** scheme proposed in [Boe93] for a mark-sweep style GC: when new memory is allocated, we discard all pages whose addresses may be misidentified as live data in the future. Unlike Boehm's scheme, however, we choose to blacklist an entire page (512 bytes) rather than a single object at a time in our scheme. This is necessary because keeping the blacklist in a granularity of an object makes compaction overly complicated. Of course, blacklisting on a page granularity may cause the system to unnecessarily leave many pages unused. However, we have not observed any degenerate effect due to this scheme in our applications. Furthermore, since unused pages are not dirtied they will not need to be swapped out by virtual memory. For extra efficiently, we plan to implement memory mapping/unmapping with **mmap/munmap**(or their equivalent) for OSes supporting these features.

Future optimizations

We also plan to implement a generation scheme, such as [Bar89]. However, since assignment is a common operation in C++, the cost of implementing a write barrier may become prohibitively high. Furthermore, it is unclear how this can be implemented while maintaining compatibility with existing code. Further research is needed in this area.

4 Conclusion

In this paper we introduced Prop, an extension language for C++ designed for language processing. Using Prop, C++ programmers now have access to a rich set of declarative and/or rule-based formalisms for manipulating high level data structures. Mapping of data structures and mapping of high level constructs are automatic and efficient.

Acknowledgement

Thanks to Ed Osinski of the Courant Institute for his proofreading and helpful comments.

References

[AF] Guiseppe Attardi and Tito Flagella. A customisable memory management framework. Technical report, University of Pisa.

[2] Currently, finalization synchronization must be handled by the object itself.

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Bar88] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, California, Febrarry 1988.

[Bar89] Joel F. Bartlett. Mostly-copying collection picks up generations and C++. Technical Report TN-12, DEC Western Research Laboratory, Palo Alto, California, October 1989.

[Boe93] Hans-Juergen Boehm. Space efficient conservative garbage collection. *ACM SIGPLAN PLDI*, pages 197–206, 1993.

[Cha87] David R. Chase. An imporvement to bottom-up tree pattern matching. *Proceedings Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 168–177, 1987.

[ED93] J.R. Ellis and D.L. Detlefs. Safe, efficient garbage collection for C++. Technical Report CSL-93-4, Xerox Parc, 1993.

[Ede92] Daniel R. Edelson. Comparing two garbage collectors for C++. Technical Report USCS-CRL-93-20, University of California at Santa Cruz, 1992.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[HMM86] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Report ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1986.

[HO82] Christoph H. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[Les75] M. E. Lesk. LEX: a lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[Pax90] V. Paxson. Using flex — a fast lexical analyzer. Technical report, The Regents of the University of California, May 1990.

[PDC91] T. Parr, H. Dietz, and W. Cohen. PCCTS reference manual. Technical Report TR-EE 90-14, Puerdue University, West Lafayette, Indiana, August 1991.

[RMH90] Mads Tofte Robin Miller and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[SPvE93] M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors. *Term Graph Rewriting. Theory and Practice.* John Wiley & Sons, 1993.

[Str91] Bjarne Stroustrup. *The C++ Programming Language. Second Edition.* Addison-Welsey, 1991.