TWEAST: A Simple and Effective Technique to Implement Concrete-Syntax AST Rewriting Using Partial Parsing

Akim Demaille Roland Levillain Benoît Sigoure

EPITA Research and Development Laboratory (LRDE), Paris, France

24th Annual ACM Symposium on Applied Computing (SAC)
Waikiki Beach, Honolulu, Hawaii, USA – March 9 - 11, 2009

Intent

Context and Scope

- Implementation of front ends of compilers, interpreters and other language processing tools.
- Scope restricted to the front end of these tools.

Facts

- Program transformation based on rewriting rules is a useful paradigm for the implementation of the aforementioned tools.
- Rewriting rules are often expressed using the abstract syntax of the processed language, by manipulating Abstract Syntax Trees (ASTs)...
- ... But concrete syntax is much more legible! Compare `Op (Int (1), Plus, Int (2))` with `1 + 2`.

Intent

Context and Scope

- Implementation of front ends of compilers, interpreters and other language processing tools.
- Scope restricted to the front end of these tools.

Facts

- Program transformation based on rewriting rules is a useful paradigm for the implementation of the aforementioned tools.
- Rewriting rules are often expressed using the abstract syntax of the processed language, by manipulating Abstract Syntax Trees (ASTs)...
- ...But concrete syntax is much more legible!
  Compare '`Op (Int (1), Plus, Int (2))`' with '`1 + 2`'.

Intent

Context and Scope

- Implementation of front ends of compilers, interpreters and other language processing tools.
- Scope restricted to the front end of these tools.

Facts

- Program transformation based on rewriting rules is a useful paradigm for the implementation of the aforementioned tools.
- Rewriting rules are often expressed using the abstract syntax of the processed language, by manipulating Abstract Syntax Trees (ASTs). . .
- . . . But concrete syntax is much more legible!
  Compare '`Op (Int (1), Plus, Int (2))`' with '`1 + 2`'.

Intent

Context and Scope

- Implementation of front ends of compilers, interpreters and other language processing tools.
- Scope restricted to the front end of these tools.

Facts

- Program transformation based on rewriting rules is a useful paradigm for the implementation of the aforementioned tools.
- Rewriting rules are often expressed using the abstract syntax of the processed language, by manipulating Abstract Syntax Trees (ASTs)...
- ...But concrete syntax is much more legible!
  Compare '`Op (Int (1), Plus, Int (2))`' with '`1 + 2`'.

Intent

Context and Scope

- Implementation of front ends of compilers, interpreters and other language processing tools.
- Scope restricted to the front end of these tools.

Facts

- Program transformation based on rewriting rules is a useful paradigm for the implementation of the aforementioned tools.
- Rewriting rules are often expressed using the abstract syntax of the processed language, by manipulating Abstract Syntax Trees (ASTs)...
- ...But concrete syntax is much more legible!
  Compare '`Op (Int (1), Plus, Int (2))`' with '`1 + 2`'.

Intent (cont.)

More Facts

- There are several tools to implement concrete-syntax AST rewriting (ASF+SDF [van den Brand et al., 1995], Stratego/XT [Bravenboer et al., 2006], TXL [Cordy, 2006])...
- ...but then you have to depend on an extra language/tool/framework.

Goal

Design a simple and adaptable framework to generate and rewrite ASTs using the concrete syntax of the processed language.

Intent (cont.)

More Facts

- There are several tools to implement concrete-syntax AST rewriting (ASF+SDF [van den Brand et al., 1995], Stratego/XT [Bravenboer et al., 2006], TXL [Cordy, 2006])...
- ...but then you have to depend on an extra language/tool/framework.

Goal

Design a simple and adaptable framework to generate and rewrite ASTs using the concrete syntax of the processed language.

Intent (cont.)

More Facts

- There are several tools to implement concrete-syntax AST rewriting (ASF+SDF [van den Brand et al., 1995], Stratego/XT [Bravenboer et al., 2006], TXL [Cordy, 2006])...
- ...but then you have to depend on an extra language/tool/framework.

Goal

Design a simple and adaptable framework to generate and rewrite ASTs using the concrete syntax of the processed language.

Intent (cont.)

More Facts

- There are several tools to implement concrete-syntax AST rewriting (ASF+SDF [van den Brand et al., 1995], Stratego/XT [Bravenboer et al., 2006], TXL [Cordy, 2006])...
- ...but then you have to depend on an extra language/tool/framework.

Goal

Design a simple and adaptable framework to generate and rewrite ASTs using the concrete syntax of the processed language.

Foreword

- Examples use C++, but the approach is applicable to any general purpose language.
- No specific tool is required. Illustrations make use of the GNU Bison parser generator [Corbett et al., 2003], but this is not a requirement.
- Applications: program transformation within a small compiler for a simple language, Tiger [Appel, 1998].

TWEAST: A Simple and Effective Technique to Implement Concrete-Syntax AST Rewriting Using Partial Parsing

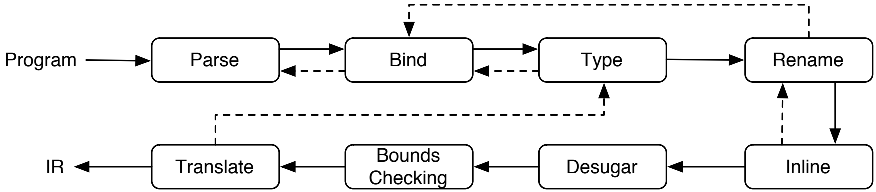
Concrete-Syntax Manipulation

Front End & Tasks

- A front end can be decomposed as a sequence of tasks.
- Tasks communicate by exchanging Abstract Syntax Trees (ASTs).
- In our Tiger compiler, we found it convenient to order tasks (solid arrows) according to their dependencies (dashed arrows).

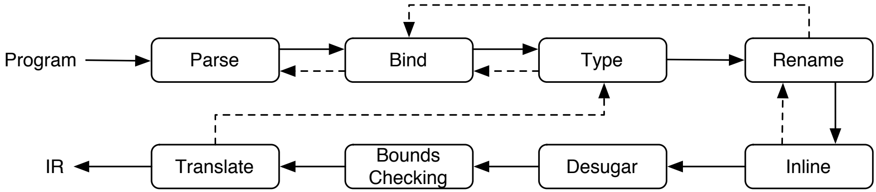
Front End & Tasks

- A front end can be decomposed as a sequence of tasks.
- Tasks communicate by exchanging Abstract Syntax Trees (ASTs).
- In our Tiger compiler, we found it convenient to order tasks (solid arrows) according to their dependencies (dashed arrows).



Front End & Tasks

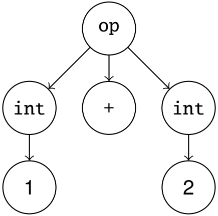
- A front end can be decomposed as a sequence of tasks.
- Tasks communicate by exchanging Abstract Syntax Trees (ASTs).
- In our Tiger compiler, we found it convenient to order tasks (solid arrows) according to their dependencies (dashed arrows).



Abstract Syntax Manipulation

- Each task manipulates an AST (traversal, generation, rewriting)
- Usually done using the abstract notation of the tree.
- The abstract syntax directly maps a tree to a textual, linear form.

`1 + 2`



`Op (Int (1), Plus, Int (2))`

Abstract Syntax Manipulation

Example

- Parsing a Boolean "and" operator as an `if-then-else` construct:

$$A \mathbin{\&} B \quad \rightarrow \quad \texttt{if}\ A\ \texttt{then}\ B \mathrel{<>} 0\ \texttt{else}\ 0$$

```
exp: exp "&" exp
  {
    $$ = new If($1,
                new Op($3, Op::NotEqual, new Int(0)),
                new Int(0));
  };
```

- '&' can be considered syntactic sugar in Tiger.
- We desugar it as a core language construct.
- Understandable, yet not very concise nor really scalable.

Abstract Syntax Manipulation

Example

- Parsing a Boolean “and” operator as an `if-then-else` construct:

$$A \mathbin{\&} B \quad \rightarrow \quad \texttt{if}\ A\ \texttt{then}\ B <> 0\ \texttt{else}\ 0$$

```
exp: exp "&" exp
  {
    $$ = new If($1,
                new Op($3, Op::NotEqual, new Int(0)),
                new Int(0));
  };
```

- ‘&’ can be considered syntactic sugar in Tiger.
- We desugar it as a core language construct.
- Understandable, yet not very concise nor really scalable.

Abstract Syntax Manipulation

Example

- Parsing a Boolean "and" operator as an `if-then-else` construct:

$$A \;\&\; B \quad \rightarrow \quad \texttt{if}\ A\ \texttt{then}\ B <> 0\ \texttt{else}\ 0$$

```
exp: exp "&" exp
  {
    $$ = new If($1,
                new Op($3, Op::NotEqual, new Int(0)),
                new Int(0));
  };
```

- '&' can be considered syntactic sugar in Tiger.
- We desugar it as a core language construct.
- Understandable, yet not very concise nor really scalable.

Abstract Syntax Manipulation

Example

- Parsing a Boolean “and” operator as an `if-then-else` construct:

$$A \mathbin{\&} B \quad \rightarrow \quad \texttt{if } A \texttt{ then } B <> 0 \texttt{ else } 0$$

```
exp: exp "&" exp
  {
    $$ = new If($1,
                  new Op($3, Op::NotEqual, new Int(0)),
                  new Int(0));
  };
```

- ‘&’ can be considered syntactic sugar in Tiger.
- We desugar it as a core language construct.
- Understandable, yet not very concise nor really scalable.

Abstract Syntax Manipulation

Example

- Parsing a Boolean “and” operator as an `if-then-else` construct:

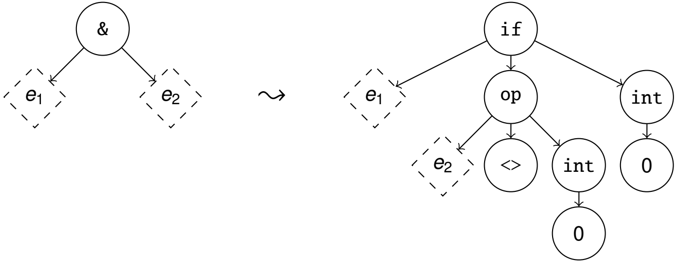
$$A \mathbin{\&} B \quad \rightarrow \quad \texttt{if } A \texttt{ then } B <> 0 \texttt{ else } 0$$

```
exp: exp "&" exp
  {
    $$ = new If($1,
                new Op($3, Op::NotEqual, new Int(0)),
                new Int(0));
  };
```

- ‘&’ can be considered syntactic sugar in Tiger.
- We desugar it as a core language construct.
- Understandable, yet not very concise nor really scalable.

Program Transformation

- The previous example illustrates a program transformation in the parser.
- Roughly, a substitution of an abstract syntax subtree pattern.
- Some leaves of the pattern are labels called metavariables.



Concrete Syntax Manipulation

- The abstract syntax notation is effective, but clutters the transformation.
- Concrete syntax is preferable in many cases.
- We propose a simple architecture where the previous example can be rewritten as this:

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                   "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- Principle: re-use the existing parser and pretty-printer (“unparser”) to implement partial parsing.

Concrete Syntax Manipulation

- The abstract syntax notation is effective, but clutters the transformation.
- Concrete syntax is preferable in many cases.
- We propose a simple architecture where the previous example can be rewritten as this:

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                     "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- Principle: re-use the existing parser and pretty-printer ("unparser") to implement partial parsing.

Concrete Syntax Manipulation

- The abstract syntax notation is effective, but clutters the transformation.
- Concrete syntax is preferable in many cases.
- We propose a simple architecture where the previous example can be rewritten as this:

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                     "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- Principle: re-use the existing parser and pretty-printer (“unparser”) to implement partial parsing.

Concrete Syntax Manipulation

- The abstract syntax notation is effective, but clutters the transformation.
- Concrete syntax is preferable in many cases.
- We propose a simple architecture where the previous example can be rewritten as this:

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                 "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- Principle: re-use the existing parser and pretty-printer (“unparser”) to implement partial parsing.

Concrete Syntax and TWEAST

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                  "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- `Tweast()` creates an object composed of
  - a growing string with "gaps" (`"if (...) then (...) <> 0 else 0"`)
  - two (sub-)ASTs (the already parsed operands of '&', represented by `$1` and `$3`).
- This object is called Text With Embedded Abstract Syntax Trees (TWEAST).

Concrete Syntax and TWEAST

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                   "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- `Tweast()` creates an object composed of
  - a growing string with “gaps” (`"if (...) then (...) <> 0 else 0"`)
  - two (sub-)ASTs (the already parsed operands of ‘&’, represented by `$1` and `$3`).
- This object is called Text With Embedded Abstract Syntax Trees (TWEAST).

Concrete Syntax and TWEAST

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                     "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- `Tweast()` creates an object composed of
  - a growing string with “gaps” (`"if (...) then (...) <> 0 else 0"`)
  - two (sub-)ASTs (the already parsed operands of ‘&’, represented by `$1` and `$3`).
- This object is called Text With Embedded Abstract Syntax Trees (TWEAST).

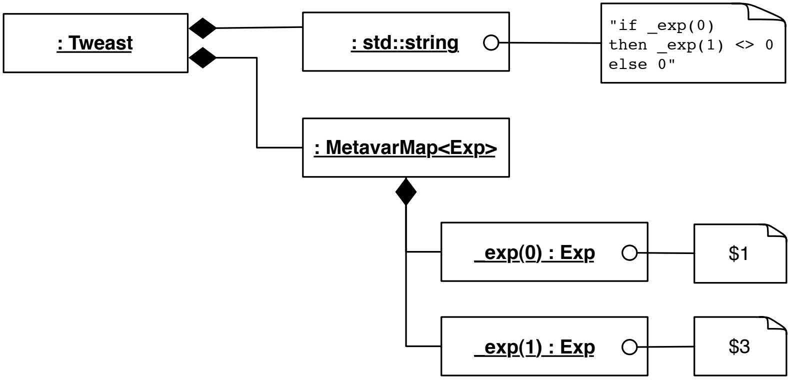
Concrete Syntax and TWEAST

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                     "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- `Tweast()` creates an object composed of
  - a growing string with “gaps” (`"if (...) then (...) <> 0 else 0"`)
  - two (sub-)ASTs (the already parsed operands of ‘&’, represented by `$1` and `$3`).
- This object is called Text With Embedded Abstract Syntax Trees (TWEAST).

Concrete Syntax and TWEAST (cont.)

The TWEAST object holds the data of the (partially constructed) desugared ‘&’ expression:



Concrete Syntax and TWEAST (cont.)

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                   "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- This object represents a state of partial parsing:
  - the sub-ASTs are the product or a previous parsing,
  - while the string is to be parsed later to produce the final AST.
- The call to `parse()` finishes the parsing: it builds an AST for the whole expression, without reparsing the operands (`$1` and `$3`).
- Operator '<<' constructs this object step by step:
  Expression `tweast << x`

Concrete Syntax and TWEAST (cont.)

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                   "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- This object represents a state of partial parsing:
  - the sub-ASTs are the product or a previous parsing,
  - while the string is to be parsed later to produce the final AST.
- The call to `parse()` finishes the parsing: it builds an AST for the whole expression, without reparsing the operands (`$1` and `$3`).
- Operator ‘<<’ constructs this object step by step:
  Expression `tweast << x`

Concrete Syntax and TWEAST (cont.)

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                   "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- This object represents a state of partial parsing:
  - the sub-ASTs are the product or a previous parsing,
  - while the string is to be parsed later to produce the final AST.
- The call to `parse()` finishes the parsing: it builds an AST for the whole expression, without reparsing the operands (`$1` and `$3`).
- Operator '<<' constructs this object step by step:
  Expression `tweast << x`

Concrete Syntax and TWEAST (cont.)

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                  "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- This object represents a state of partial parsing:
  - the sub-ASTs are the product or a previous parsing,
  - while the string is to be parsed later to produce the final AST.
- The call to `parse()` finishes the parsing: it builds an AST for the whole expression, without reparsing the operands (`$1` and `$3`).
- Operator ‘<<’ constructs this object step by step:
  Expression `tweast << x`
  - populates `tweast`’s inner string when `x` is a string;
  - registers `x` and creates a new metavariable when `x` is an AST.

Concrete Syntax and TWEAST (cont.)

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                   "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- This object represents a state of partial parsing:
  - the sub-ASTs are the product or a previous parsing,
  - while the string is to be parsed later to produce the final AST.
- The call to `parse()` finishes the parsing: it builds an AST for the whole expression, without reparsing the operands (`$1` and `$3`).
- Operator ‘<<’ constructs this object step by step:
  Expression `tweast << x`
  - populates `tweast`’s inner string when `x` is a string;
  - registers `x` and creates a new metavariable when `x` is an AST.

Concrete Syntax and TWEAST (cont.)

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                   "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- This object represents a state of partial parsing:
  - the sub-ASTs are the product or a previous parsing,
  - while the string is to be parsed later to produce the final AST.
- The call to `parse()` finishes the parsing: it builds an AST for the whole expression, without reparsing the operands (`$1` and `$3`).
- Operator ‘<<’ constructs this object step by step:
  Expression `tweast << x`
  - populates `tweast`’s inner string when `x` is a string;
  - registers `x` and creates a new metavariable when `x` is an AST.

Concrete Syntax and TWEAST (cont.)

```
exp: exp "&" exp
  {
    $$ = parse(Tweast() <<
                  "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

- This object represents a state of partial parsing:
  - the sub-ASTs are the product or a previous parsing,
  - while the string is to be parsed later to produce the final AST.
- The call to `parse()` finishes the parsing: it builds an AST for the whole expression, without reparsing the operands (`$1` and `$3`).
- Operator '<<' constructs this object step by step:
  Expression `tweast << x`
  - populates `tweast`'s inner string when `x` is a string;
  - registers `x` and creates a new metavariable when `x` is an AST.

Implementation of Abstract Syntax Trees

In Object-Oriented Programming (OOP):

Abstract Syntax Tree (AST) nodes are implemented as a hierarchy of classes.

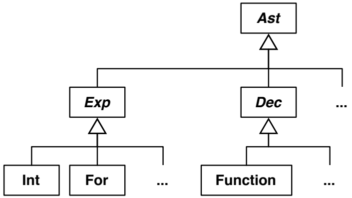
AST traversals are instances of the VISITOR design pattern [Gamma et al., 1995].

Implementation of Abstract Syntax Trees

In Object-Oriented Programming (OOP):

Abstract Syntax Tree (AST) nodes are implemented as a hierarchy of classes.

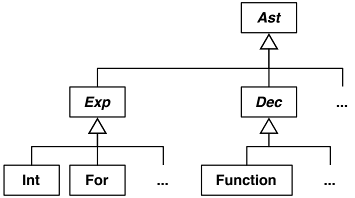
AST traversals are instances of the VISITOR design pattern [Gamma et al., 1995].



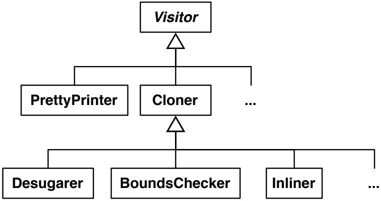
Implementation of Abstract Syntax Trees

In Object-Oriented Programming (OOP):

Abstract Syntax Tree (AST) nodes are implemented as a hierarchy of classes.



AST traversals are instances of the VISITOR design pattern [Gamma et al., 1995].



Rewriting Times

- Program transformation can occur virtually anywhere in the front-end, provided enough information (names, types) is available.
- Either directly at the parsing stage
  - Implemented in the parser.
  - The parser does the job of matching a pattern (through a production).
- Or later, when the AST is built.
  - More semantic information may be available.
  - Implemented as a Visitor (AST traversal).
  - Matching a pattern can be tedious.

Rewriting Times

- Program transformation can occur virtually anywhere in the front-end, provided enough information (names, types) is available.
- Either directly at the parsing stage
  - Implemented in the parser.
  - The parser does the job of matching a pattern (through a production).
- Or later, when the AST is built.
  - More semantic information may be available.
  - Implemented as a VISITOR (AST traversal).
  - Matching a pattern can be tedious.

Rewriting Times

- Program transformation can occur virtually anywhere in the front-end, provided enough information (names, types) is available.
- Either directly at the parsing stage
  - Implemented in the parser.
  - The parser does the job of matching a pattern (through a production).
- Or later, when the AST is built.
  - More semantic information may be available.
  - Implemented as a VISITOR (AST traversal).
  - Matching a pattern can be tedious.

Examples

Applications of AST rewriting

Desugaring I.e, removing syntactic sugar.

→ Language extensions as sugar on top of the core language.

Optimization Replace some patterns by faster equivalent code, or code requiring less resources (e.g., memory).

Code Instrumentation Perform additional tasks for many grounds: safety, debugging, profiling, etc.

Engineering Code renovation & refactoring, automated or semi-automated migrations, etc.

Applications of AST rewriting

Desugaring I.e, removing syntactic sugar.

→ Language extensions as sugar on top of the core language.

Optimization Replace some patterns by faster equivalent code, or code requiring less resources (e.g., memory).

Code Instrumentation Perform additional tasks for many grounds: safety, debugging, profiling, etc.

Engineering Code renovation & refactoring, automated or semi-automated migrations, etc.

Applications of AST rewriting

Desugaring I.e, removing syntactic sugar.

→ Language extensions as sugar on top of the core language.

Optimization Replace some patterns by faster equivalent code, or code requiring less resources (e.g., memory).

Code Instrumentation Perform additional tasks for many grounds: safety, debugging, profiling, etc.

Engineering Code renovation & refactoring, automated or semi-automated migrations, etc.

Applications of AST rewriting

Desugaring I.e, removing syntactic sugar.

→ Language extensions as sugar on top of the core language.

Optimization Replace some patterns by faster equivalent code, or code requiring less resources (e.g., memory).

Code Instrumentation Perform additional tasks for many grounds: safety, debugging, profiling, etc.

Engineering Code renovation & refactoring, automated or semi-automated migrations, etc.

Applications of AST rewriting

Desugaring I.e, removing syntactic sugar.
→ Language extensions as sugar on top of the core language.

Optimization Replace some patterns by faster equivalent code, or code requiring less resources (e.g., memory).

Code Instrumentation Perform additional tasks for many grounds: safety, debugging, profiling, etc.

Engineering Code renovation & refactoring, automated or semi-automated migrations, etc.

Syntactic Sugar Removal

In the Parser

- Desugaring unary minus as a binary minus ($-e \leadsto (0 - e)$).

```
exp: "-" exp
  {
    $$ = parse(Tweast() << "0 - " << $2);
  };
```

- Desugaring Boolean operators as `if-then-else` expressions.

```
exp:
  exp "&" exp
  {
    $$ = parse(Tweast() <<
          "if" << $1 << "then" << $3 << "<> 0 else 0");
  }
| exp "|" exp
  {
    $$ = parse(Tweast () <<
          "if " << $1 << " then 1 " << "else " << $3 << " <> 0");
  };
```

Syntactic Sugar Removal (cont.)

Desugaring a `for` loop as a `while` loop (using a visitor)

```
Ast* Desugarer::operator() (const For& e) {
  Exp* lo   = recurse(e.vardec().init());
  Exp* hi   = recurse(e.hi());
  Exp* body  = recurse(e.body());
  const Symbol& var = e.vardec().name();
  return parse(Tweast() <<
          " let"
          "   var _lo := " << lo <<
          "   var _hi := " << hi <<
          "   var " << var << " := _lo"
          " in"
          "   if _lo <= _hi then"
          "     while 1 do ("
          "       " << body << ";"
          "       if " << var << " = _hi then"
          "         break;"
          "       " << var << " := " << var << " + 1"
          "     )"
          " end");
}
```

Optimization

Inlining of function bodies

- Aim: translate the following code

```
let function add(x : int, y : int) =
  x + y
in
  add(42, 51)
end
```

into

```
let var x := 42
    var y := 51
in
  x + y
end
```

Optimization

Inlining of function bodies (cont.)

```
Ast* Inliner::operator() (const Call& e)
{
  const Function& fun(e.definition());
  // A recursive function cannot be inlined.
  if (recursive_functions_set.has(fun))
    return clone(e);
  else
    {
      Tweast t;
      t << "let";
      // Introduce temporaries to evaluate formal arguments once.
      foreach (const Exp& a, e.args())
        {
          Symbol v = Symbol::fresh();
          t << "var" << v << " : " << a.type() << " := " << clone(a);
        }
      // Inlined call.
      t << "in" << recurse(fun.body()) << "end";
      return parse(t);
    }
}
```

Optimization

More applications

- Loop unrolling (when the bounds are statically known).
- Constants propagation.
- Partial evaluation (when some or all of the terms of an expression are statically known).
- Vectorization.
- Etc.

Code Instrumentation

- Add run-time checks of array accesses (bounds checking).
- Trace the execution of the program by logging events like function entries and exits, memory allocations, etc.
- Record run-time information (time elapsed in functions, memory consumption) for profiling purpose.
- Etc.

Implementing TWEASTs

Overview

Adding support for TWEAST in your favorite tool/language

1. Implement `Tweast` objects and metavariables.
2. Equip the parser and the scanner.
3. Implement transformations as visitors (or in the parser).

Overview

Adding support for TWEAST in your favorite tool/language

1. Implement `Tweast` objects and metavariables.
2. Equip the parser and the scanner.
3. Implement transformations as visitors (or in the parser).

Overview

Adding support for TWEAST in your favorite tool/language

1. Implement `Tweast` objects and metavariables.
2. Equip the parser and the scanner.
3. Implement transformations as visitors (or in the parser).

`Tweast` objects

- Add a class `Tweast` aggregating
  - a growing string;
  - several typed dictionaries for sub-ASTs — expressions, l-values, declarations. . .
- Possibly implement the overloaded ‘<<’ sugar.

Equip the parser and the scanner

- `Tweast` objects create special codes like '`_exp`' in their inner string to materialize metavariables. These must be recognized as valid tokens in the scanner.

```
if _exp(0) then _exp(1) <> 0 else 0
```

- Metavariables (e.g. `_exp(0)`) must be accepted by the parser as valid right-hand sides of the corresponding non-terminal (`exp`).

```
exp: "_exp" "(" INT ")" { $$ = driver.tweast->_exp[$3]; }
```

Equip the parser and the scanner

- `Tweast` objects create special codes like '`_exp`' in their inner string to materialize metavariables. These must be recognized as valid tokens in the scanner.

```
if _exp(0) then _exp(1) <> 0 else 0
```

- Metavariables (e.g. `_exp(0)`) must be accepted by the parser as valid right-hand sides of the corresponding non-terminal (`exp`).

```
exp: "_exp""(" INT ")" { $$= driver.tweast->_exp[$3]; }
```

Equip the parser and the scanner (cont.)

- It is convenient to encapsulate the parser and the scanner as well as the parsing context (input, produced AST, flags, etc.) in a dedicated object.

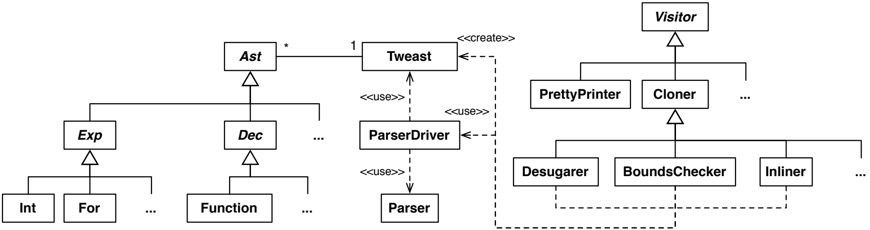
  → The Parser Driver design pattern, as special case of Facade.
  - Supports recursive parsing.
  - Equally parses from an actual file or from a TWEAST.

Transformations as Visitors

- Used to match patterns to be rewritten.
- Rewrite the AST by creating modified copies.
- Derive from a `Cloner` visitor to factor the duplicating code.

Overall Architecture

Classes involved in transformations in run-time concrete syntax:



Conclusions

Conclusion

- TWEASTs provide a simple program transformation framework at a little implementation cost.
- Fairly portable/adaptable to other languages and contexts.
- Concrete syntax saves a lot of source lines of code.

Measure	Abstract Syntax	Concrete Syntax	Gain
C++ `new` expressions	146	1	99%
Non-whitespace characters	995	671	32%
Words	886	340	61%

- Concrete syntax introduces almost no run time penalty.
  - $\approx$ 1.5% of the run time of the front-end.
  - $<$ 0.1% of the run time of the entire compiler pipeline.

Extending the TWEAST concept

Static TWEAST

- A great part of the run time cost of using TWEASTs comes from systematically parsing the string they contain.

→ Factor this cost by parsing the string once, the first time the contents of the TWEAST is used.

→ Apply memoization, using persistent (`static`) TWEAST objects.

```
exp: exp "&" exp
  {
    static Tweast bool_and("if %exp:1 then %exp:2 <> 0 else 0");
    $$ = exp(bool_and % $1 % $3);
  };
```

Extending the TWEAST concept

Full Concrete-Syntax Rewriting

- Matching patterns (with a visitor) is difficult, since it involves abstract syntax.

→ Use concrete syntax for matching as well.

→ Assemble two concrete syntax patterns (match and build) as a rewrite rule.

```
// A rewrite rule translating ‘0 + e’ as ‘e’.
RewriteRule r("0 + %exp:1", "%exp:1");
Ast* ast = r("0 + 42");  // Rewritten as ‘42’.
```

- Requires some extensions of the framework, in particular a generic mechanism to match a tree pattern within an AST.

TWEAST: A Simple and Effective Technique to Implement Concrete-Syntax AST Rewriting Using Partial Parsing

Bibliography I

- Appel, A. W. (1998).
  Modern Compiler Implementation in C, Java, ML.
  Cambridge University Press.

- van den Brand, M. G. J., van Deursen, A., Dinesh, T. B., Kamperman, J. F. T., and Visser, E., editors (1995).
  Proceedings of the Workshop on Generating Tools from Algebraic Specifications (ASF+SDF'95). Technical Report P9504, Programming Research Group, University of Amsterdam.

- Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. (2006).
  Stratego/XT 0.16. Components for transformation systems.
  In *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06)*, Charleston, South Carolina. ACM SIGPLAN.

Bibliography II

Corbett, R., Stallman, R., and Hilfinger, P. (2003).
Bison: GNU LALR(1) and GLR parser generator.
http://www.gnu.org/software/bison/bison.html.

Cordy, J. R. (2006).
The TXL source transformation language.
Science of Computer Programming, 61(3):190–210.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995).
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY.