Early Experience with ASDL in lcc

David R. Hanson
Microsoft Research
Roadmap

- About: Abstract Syntax Description Language (ASDL) [3 slides]
- About: The lcc code-generation interface [3]
- Using ASDL to divide lcc into N separate programs [2]
- The details ... ZZZ... [11]
- Good: finds bugs, constrains algorithms, define data structures, ... [1]
- Bad: redundant data structures, constrains algorithms, graphs, ... [2]
- What happens next? [1]

- Executive Summary: ASDL works in a real compiler
Abstract Syntax Description Language (ASDL)

- **What:** A language for specifying tree intermediate representations — tree data structures, e.g., ASTs
- **Why:** Promote interoperability of compiler parts written in different languages
- **How:** Given an ASDL specification, generate
  - Data structure declarations in C, C++, Java, ML, or Haskell
  - Code to read/write data structures as language- and machine-independent “pickles”

![Flowchart: Front end; C pickle —> Optimizer; ML pickle —> Code generator; Java]

- **Cynic’s view:** Uses RPC technologies — IDLs, stub generation — to build modular compilers
Example: IR for Assignments + Print Statements

- **In IR.asdl:**
  
  \[
  \begin{align*}
  \text{stm} & = \text{SEQ}(\text{stm}, \text{stm}) \\
  & \quad | \text{ASGN}(\text{identifier}, \text{exp}) \\
  & \quad | \text{PRINT}(\text{exp}*) \\
  \text{exp} & = \text{OP}(\text{binop}, \text{exp}, \text{exp}) \\
  & \quad | \text{ID}(\text{identifier}) \\
  & \quad | \text{CON}(\text{int}) \\
  \text{binop} & = \text{ADD} \mid \text{SUB} \mid \text{MUL} \mid \text{DIV}
  \end{align*}
  \]

- **Given IR.asdl, generator emits for C (ditto for Java, ...)**
  - `IR.h` declarations for types, functions
  - `IR.c` code for constructors, readers, writers

- “Producers” use IR.[ch] to build and write IR pickles
- “Consumers” use IR.[ch] to read IR pickles
Example: Generated Code

- IR.h holds type declarations, prototypes:
  
  ```c
  typedef struct exp_s *exp_ty;
  struct exp_s {
      enum {OP_enum, ID_enum, CON_enum} kind;
      union {
          struct OP_s { binop_ty binop1; exp_ty exp1,exp2;} OP;
          struct ID_s { identifier_ty identifier1;} ID;
          struct CON_s { int_ty int1;} CON;
      } v;
  }
  ```

- IR.c holds constructors, etc., e.g., to construct OPs:
  
  ```c
  exp_ty OP(binop_ty binop1, exp_ty exp1, exp_ty exp2) {
      IR_exp_ty ret = alloc(sizeof(*ret));
      ret->kind = OP_enum;
      ret->v.OP.binop1 = binop1;
      ret->v.OP.exp1 = exp1; ret->v.OP.exp2 = exp2;
      return ret;
  }
  ```
lcc 4.x Code-Generation Interface

- Shared data structures: nodes, symbols, types, metrics
- 33 generic intermediate representation (IR) operators
  
<table>
<thead>
<tr>
<th>CNST</th>
<th>ARG</th>
<th>ASGN</th>
<th>INDIR</th>
<th>CVF</th>
<th>CVI</th>
<th>CVP</th>
<th>CVU</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEG</td>
<td>CALL</td>
<td>RET</td>
<td>ADDRG</td>
<td>ADDRF</td>
<td>ADDRL</td>
<td>ADD</td>
<td>SUB</td>
</tr>
<tr>
<td>LSH</td>
<td>MOD</td>
<td>RSH</td>
<td>BAND</td>
<td>BCOM</td>
<td>BOR</td>
<td>BXOR</td>
<td>DIV</td>
</tr>
<tr>
<td>MUL</td>
<td>EQ</td>
<td>GE</td>
<td>GT</td>
<td>LE</td>
<td>LT</td>
<td>NE</td>
<td>JUMP</td>
</tr>
</tbody>
</table>

- 6 type suffixes: F I U P B V
- Up to 9 sizes: 5 integer, 3 floating, 1 pointer
- Semantics + size overlap reduce operator repertoire
  - ~130 type/size-specific operators for 32-bit targets
  - CVIF4 CVIF8
  - CVII1 CVII2 CVII4
  - CVIU1 CVIU2 CVIU4
  - MULF4 MULF8 MULI4 MULU4
18 Code-Generation Functions

- Initialize/finalize a back end
  - progbeg, progend
- Define/initiate symbols
  - address, local, defsymbol, global, import, export
- Initialize/finalize scopes
  - blockbeg, blockend
- Generate/emit code
  - gen, emit, function
- Generate initialized data
  - defconst, defstring, defaddress, space, segment
- Instruction selection specified by short machine specs.
  - IBURG-style tree pattern matcher
  - Optimal local code
Interface Records & Cross Compilation

- Interface records hold target-specific interface data

  ```
  typedef struct interface {
    Metrics charmetric; ... structmetric;
    unsigned little_endian:1; ... unsigned unsigned_char:1;
    void (*address)(Symbol p, Symbol q, long n);
    void (*blockbeg)(Env *e);
    ...
    void (*space)(int n);
    Xinterface x;
  } Interface;
  ```

- Compile-time option binds front end to an interface record:
  ```
  % lcc -Wf-target=mips/irix -S wf1.c
  ```

- Front end uses indirect calls, e.g.,
  ```
  (*IR->defsymbol)(p);
  ```
lcc is a Monolithic Compiler — By Design

- **Small footprint (x86 under NT 4.0):**
  - 873 KB  front end + all back ends
  - 21  common back end support, e.g., register allocator
  - 55  MIPS back end
  - ...  
  - 89  X86/Linux back end

- **Interface has upcalls, so assumes one address space**
Use ASDL to Divide lcc

% lcc -Wf-target=mips/irix
   -Wf-asdl -S wf1.c
% mv wf1.s wf1.pickle
...
% pass2 wf1.pickle >wf1.s
% 2html wf1.pickle >wf1.html
Icc ASDL Grammar (~70 lines)

- **Encodes interface data structures + calls**
  - Pickles depend on the target, e.g., type sizes, alignments
  - Pickles depend on platform, e.g., system-dependent header files

- **Pickles include excess baggage necessary for pass2**

```c
module rcc {
    program = (int nuids, int nlabels, item* items, 
               interface* interfaces, int argc, string *argv)
    item = Symbol(symbol symbol) 
          | Type(type type) attributes(int uid)
    symbol = ... -- symbol-table entries
    type = ... -- type representation
    interface = ... -- interface function calls
    node = ... -- IR trees
}
```

- **Use integers to represent unique items, e.g., symbols**
- I.e., To encode DAGs
Symbols

symbol = (identifier id, int type, int scope,
        int sclass, int ref, int flags)

struct node {
    int count;
    ...
};
main() { struct node *root; ... }

(id = root, type = 10, scope = LOCAL, sclass = AUTO,
 ref = 120000, flags = addressed)

- Built-in types: identifier, int
- Identifiers are unique
- Types identified by unique integers (uids)
Types

field = (identifier id, int type, int offset, int bitsize, int lsb)

enum = (identifier id, int value)

type = INT
| UNSIGNED
| FLOAT
| VOID
| POINTER(int type)
| ENUM(identifier tag, enum* ids)
| STRUCT(identifier tag, field* fields)
| UNION(identifier tag, field* fields)
| ARRAY(int type)
| FUNCTION(int type, int* formals)
| CONST(int type)
| VOLATILE(int type)

attributes(int size, int align)

- ASTs represent types
Example: Representing struct node

```c
struct node {
    int count;
    struct node *left;
    struct node *right;
    char *word;
};
```

11: `STRUCT(size = 16, align = 4, fields = [
    id   type  offset  bitsize  lsb
    (count, 12, 0, 0, 0),
    (left, 10, 4, 0, 0),
    (right, 10, 8, 0, 0),
    (word, 13, 12, 0, 0) ] )`

10: `POINTER(size = 4, align = 4, type = 11)`

13: `POINTER(size = 4, align = 4, type = 8)`

8: `INT(size = 1, align = 1)`
Associating uids with Types and Symbols

```
program = (int nuids, int nlables, item* items, interface* interfaces, int argc, string *argv)
item = Symbol(symbol symbol)
    | Type(type type) attributes(int uid)

(nuids = 97, nlables = 55, items = [
    Type(uid = 8, INT(size = 1, align = 1)),
    Type(uid = 12, INT(size = 4, align = 4)),
    ...
    Symbol(uid = 23, (id = printf, type = 29, ... )),
    ...
    Type(uid = 94, ARRAY(size = 32000, ..., type = 11)),
    Symbol(uid = 1, (id = words, type = 94, ... )) ]
interfaces = [ ... ], argc = 5,
argv = [ "x86\win32\rcc.exe", "-target=x86/win32",
    "-asdl", ... ]
```

See `wf1.html`
IR Trees

node = CNST(int value)
    CNSTF(real value)
    ARG(node left, int len, int align)
    ASGN(node left, node right, int len, int align)
    CVT(int op, node left, int fromsize)
    CALL(node left, int type)
    CALLB(node left, node right, int type)
    RET
    ADDRG(int uid)
    ADDRL(int uid)
    ADDRF(int uid)
    Unary(int op, node left)
    Binary(int op, node left, node right)
    Compare(int op, node left, node right, int label)
    LABEL(int label)
    BRANCH(int label)
    CSE(int uid, node node)
attributes(int suffix, int size)
Example: char *s; int c; *s++ = c

- t1 = s; s = t1 + 1; *t1 = c;

Nodes are written in prefix

ASGN P 4
ADDRL P 4 uid = 42 t1
Unary INDIR P 4
ADDRL P 4 uid = 37 s

ASGN P 4
ADDRL P 4 uid = 37
Binary ADD P 4
Unary INDIR P 4
ADDRL P 4 uid = 42
CNST I 4 1
Example: char *s; int c; *s++ = c

- t1 = s; s = t1 + 1; *t1 = c;
Interface Function Calls

interface = Export(int p)
| Import(int p)
| Global(int p, int seg)
| Local(int uid, symbol p)
| Address(int uid, symbol q, int p, int n)
| Segment(int seg)
| Defaddress(int p)
| Deflabel(int label)
| Defconst(int suffix, int size, int value)
| Defconstf(int size, real value)
| Defstring(string s)
| Space(int n)
| Function(int f, int* caller, int* callee,
          int ncalls, interface* codelist)
| Blockbeg
| Blockend
| Forest(node* nodes)
Interface Encodes Code-Generation Calls

(nuids = 97, nlabels = 55, items = [...],

    interfaces = [
        Export(p = 4 main),
        Segment(seg = CODE),
        Function(f = 4 main, ...),
        ...
        Local(uid = 27, symbol = (id = s, ...)),
        Local(uid = 28, symbol = (id = s, ...)),
        Function(f = 22 err, caller = [ 27 ], callee = [ 28 ],
            ncalls = 2, codelist = [
                Blockbeg(),
                Forest(nodes = ...),
                Forest(nodes = ...),
                Blockend(),
                Forest(nodes = ...)]),
        ...
        Defstring(? %s\n") ]

    argc = 5, argv = [...]
)

err(s) char *s; {
    printf("? %s\n", s);
    exit(1);
}
Statistics

- **Programmer-written code:**
  - 70 lines ASDL specification
  - 358 ASDL “back end”
  - 681 pass2.c
  - 559 2html.c

- **Generated code (from ASDL spec.):**
  - 313 lines rcc.h
  - 1870 rcc.c

- **Executables:**
  - 853 KB rcc
  - 846 pass2
  - 202 2html

- **Pickles:**
  - 4059 bytes wf1.pickle
  - 2148 wf1.o (x86 under Linux)
Good Things

- Helps find bugs

  ```c
  f(void) { extern int x; ... }
  int x;
  ```
  - Bug: two symbol-table entries with identical contents

  ```c
  static int x;
  f(void) { extern int x; ... }
  ```
  - Bug: two symbol-table entries, and inner x appears to be static when declared, extern when used (!)

- Constrains algorithms/binding times

  ```c
  f(x, y) char x; int y; { ... }
  ```
  is compiled as

  ```c
  f(? int x', ? int y') { ? char x = x'; ? int y = y'; ...}
  ```
  - Back end sets ?, front end emits only necessary assignments
  - pass2 must emit assignments
Bad Things

- **Redundant data structures**
  - Everything written to a pickle has at least two representations
    - Types
      - Symbols
        - Nodes
          - Calls
            - Other
  - **Icc**
    - Type
    - Symbol
    - Node
    - Code
  - **ASDL**
    - Type
    - Symbol
    - Node
    - Interface
      - Item
        - Program
          - Sequence
  - Plus constructors, allocators, etc. — use ASDL from the outset

- **Constrains algorithms/binding times**
  - \[ a = b\times c + b\times d \]
  - Load \( b \) into a temp? Not if \( b \) is in a register, or temp isn’t
  - Front end inserts possibly useless temps; pass2 eliminates them

- **Little help in dealing with graphs**
  - Mappings of integers to data structures, e.g., item, are common
Ambiguity

- **Bad: ASDL grammar admits invalid trees, code lists**
  
  ```
  program = (int nuids, int nlabels, item* items, 
             interface* interfaces, int argc, string *argv)
  node = ...
     | CVT(int op, node left, int fromsize)
  ...
     | Binary(int op, node left, node right)
  interface = ...
     | Function(int f, int* caller, int* callee, 
               int ncalls, interface* codelist)
  ```

- **Good: Ambiguity permits compact grammars**

- **Hindsight: Ambiguity admits bugs**
  - Debugging binary pickles at runtime is tedious
  - Non-ambiguous grammars help catch errors at compile-time, and are better documentation
What Happens Next?

- **Add more information, e.g., for optimization**
  - Flow graph, live ranges
  - Debugging information, e.g., visibility information, source coordinates

- **Investigate XML — isomorphic to ASDL**
  - Leverage widely available XML tools?
  - XML Document Type Declaration (DTD) \(\leftrightarrow\) ASDL
  - Use XML’s ID/IDREF attributes to handle graphs?

- **Design new code-generation interfaces**
  - Target-independent; ANDF?
  - Family of code-generation interfaces?

- **Use ASDL elsewhere**
  - Data structure definitions
  - Interface specifications