The Structure of ANDF: Principles and Examples

Stavros Macrakis
Open Software Foundation

ANDF is a language-independent and machine-independent intermediate language. It is organized by some fundamental principles, which we explain and illustrate.

1. Introduction

ANDF (the Architecture-Neutral Distribution Format) defines the data passed from an ANDF producer (which is language-dependent and machine-independent) to an ANDF installer (which is language-independent and machine-dependent). ANDF is both language-independent and machine-independent. An ANDF producer is like a compiler front end (syntax and semantics analyzer), and an installer is like a compiler back end (code generator and optimizer). ANDF itself is thus a sort of compiler intermediate language. This paper presupposes an understanding of compiler intermediate languages in general, and should be useful to compiler writers and language designers.

ANDF is organized by a few fundamental principles which give it coherence and universality. This paper introduces them one by one, with examples from a C or other language source and an example ANDF equivalent (other translations may also be valid).

The ANDF in this document is formatted for easy reading. Much of it is syntactic sugar, notably the keywords identifying fields. The discussion of its linear, binary form near the end explains how it is efficiently encoded into roughly the same space as object code.

There are several layers to the explanation. An example follows each one. In the examples, '`...`' represents parts that aren't relevant to the particular point being made. '`<<...>>`' means 'the ANDF for that C construct'. This document is based on a pre-release version of ANDF, so details may change; however, the general principles remain valid.

2. Requirements

Neutrality

ANDF makes no commitment to a particular source language or target machine. When different languages or machines have different requirements, the differences are usually covered by parameterization of ANDF constructs, not by special-purpose code.

Complete information

ANDF preserves the information in the source program that could be useful to the code generator. Thus the installed code can be as efficient as normally compiled code.

For instance, if an array index is of a type which makes it impossible for indexing to fail, this fact will be preserved in the ANDF, and the unnecessary range check eliminated.

Wide spectrum for transformation

ANDF can be used as a high-level or low-level intermediate language. Thus most optimizations (whether machine-specific or not) can be expressed as ANDF-to-ANDF transformations. The optimization engine can remain constant across implementations, even if the particular optimizations are

different. In particular, ANDF can represent both machine-independent data types specified semantically and machine-dependent data types specified operationally (number of bits).

3. General Structure

ANDF is tree-structured

ANDF is a tree built out of constructors like `plus` (integer addition) and `integer` (specifies integer types). Every constructor belongs to a category, or "sort", and specifies the sorts of its constituents. Constructs are nested within one another, so the full definition of a top-level object (*e.g.* a procedure) is a single tree.

Constructs have values

Executable code is represented by constructors of sort EXP. Each such constructor specifies the concrete type, or "shape", of its value and its EXP arguments. A void value is of shape Top. Constructs which do not return (such as goto's) have a notional return value of shape Bottom.

ANDF constructs are orthogonal

ANDF constructs mean the same thing no matter where they are used. Any entity (*e.g.* a local variable) a construct defines is meaningful only within its body, and not outside of it. (A special mechanism is used for global definitions.)

C version:

```
{
  int i;              — declaration looks just like
  i = 0;              — a sequential statement in C
  ...
}
```

ANDF version:

```
variable(...
        tag⇒2,          — variable 'name'
        value⇒ make_value(INTEGER,...),
        body⇒           — body is nested in ANDF
           sequence(<<i = 0>>,
                    <<...>>)
            )
```

Rationale:

Unambiguous. Simplifies high-level optimizations.

All attributes are explicit

When a construct has options or attributes, they are all explicit. Most constructs have a fixed number of parameters (fixed arity), except for lists of uniform meaning (*e.g.* the statements of a block). There are no globally-set defaults which modify ANDF semantics for various languages.

C version:

```
int i;
```

ANDF version:

```
variable(visible⇒true,
         tag⇒2,
         value⇒ make_value(INTEGER,...),
         construct_in_scope⇒...
           )
```

Note that `visible` controls the visibility of the variable after exceptions and in (*e.g.*) nested procedures, and is always true for C variables.

Rationale:

Unambiguous. Requires no reconstruction in the installer (back end).

Wide-spectrum and mixed-language programs may have different values for attributes within the same expression tree (*e.g.* mixed-language inlining).

Different languages may use different values for attributes.

Potential size problem dealt with by clever binary format. (*q.v.*)

Operations are completely specified

All operations are completely specified for both normal and exceptional cases independently of target machine. Where the source language specifies machine-dependent behavior, this is stated explicitly in the ANDF code. When producer analysis of the program determines that errors are impossible, this information is passed along to avoid unnecessary error checks.

C version:

```
i + 1
```

ANDF version:

```
plus(
     ov_error⇒wrapped,
     contents(
              INTEGER(~signed_int),
              obtain_tag(2)
             ),
     make_int(~signed_int, 1)
    )
```

The behavior of `plus` on overflow demonstrates the explicitness of exceptional behavior. C semantics specify that overflow should result in the wrapped (modular) result; other languages may raise an exception. Thus ANDF must specify which behavior is required.

Rationale:

Program semantics can be uniform, despite target machines' varying behavior, yet can take advantage of target-specific behavior when this is acceptable.

Optimization benefits from information on impossible cases.

4. Names and values

Identifiers are replaced with unique Tags

Identifiers are replaced with unique ‘tags’. There are no scope rules for tags (or tokens) because they are guaranteed unique within an ANDF capsule (unit of distribution).

C version:

```
int authorization_number;
resolve_constraints(...)
```

ANDF version:

```
variable(..., tag⇒2341, ...)
make_proc(..., tag⇒3833, ...)
```

Pascal version:

```
integer shadowed;
begin
  integer shadowed;     — not the same variable
  ...
```

ANDF version:

```
variable(..., tag⇒501, ...)
variable(..., tag⇒299, ...)
```

Ada version:

```
function “*” (a,b: integer) return integer;
                        — Unusual function name
function “*” (a,b: view_matrix)
                          return view_matrix;
                        — Overloading
```

ANDF version:

```
make_proc(..., tag⇒7211, ...)
make_proc(..., tag⇒2007, ...)
```

Rationale:

Syntax and semantics of tags are perfectly uniform. Lookup of tags is efficient. Language-dependent rules of scope, overloading, and so on are all treated by the producer. Tags are character-set independent.

Installer doesn’t need any other information, and this representation is more compact.

Inhibits reverse engineering.

Simplifies transformation.

Fetching of name’s content is always explicit

Tags for variables always represent the name (L-value) of the identifier, and must be explicitly dereferenced to get the value.

C version:

```
i = j
```

ANDF version:

```
assign(...
  ptr⇒obtain_tag(2),
  val⇒contents(..., obtain_tag(3))
  )
```

Rationale:

Preserves orthogonality of language. Also recognizes different kinds of value fetching (*cf.* contents_of_volatile).

Types always explicit

Types of all objects, intermediate expressions, and constants are explicit.

C version:

```
100
```

ANDF version:

```
make_int(~signed_int,100)
```

C version:

```
i
```

ANDF version:

```
contents(..., INTEGER(~signed_int),
        obtain_tag(2))
```

5. Types and storage

Concrete types are specified by their attributes

ANDF only represents the concrete aspects of types. These concrete types are called ‘shapes.’

The static semantics of types in general, and static type safety in particular, are language-specific, and so are handled by the producer.

Integer shapes are specified by their minimum and maximum values; floating point shapes are specified by their extreme exponent values and their precision.

C version:

```
enum status {off, on}
enum binding {hardback, paperback}
char
enum Latin_1 {L1_null, ... L1_xxx}
```

ANDF version:

```
Integer(0,1)
Integer(0,1)
Integer(0,255)
Integer(0,255)
```

Ada version:

```
type binding is (paperback, hardback);
type discount is (textbook, paperback,
                  hardback, used);
```

ANDF version:

```
Integer(0,1)
Integer(0,3)
```

Rationale:

Data type semantics vary from language to language, so should not be represented in ANDF.

Data layouts are not specified

ANDF provides primitives for constructing compound data types, such as structures, unions, and arrays. Data layout is the installer's responsibility, as this is machine-dependent. This is a major difference with traditional compiler intermediate languages, where data layout is performed in the front end (or 'middle').

A corollary of this is that concrete data attributes such as size and relative position are not known at translation time in general, and thus must be deferred to installation time.

6. ANDF supports deferred decisions

In a distribution format, unlike a compiler intermediate language, no machine dependencies are tolerable in the front end or producer. Thus, many things must be deferrable to installation time in an unambiguous way.

Unambiguous unique identifiers support linking

Any portable program must depend on a variety of libraries specific to each machine. Identifying procedures (and other entities) in these libraries is crucial. On different machines, the same procedure may have different names, or procedures may have the same name but different functions.

ANDF provides for the registration of unique names (*cf.* Internet addresses) so that global symbols are completely unambiguous.

C version:

`void invert_matrix();` — NAG? IMSL? other?

ANDF version:

`Make_uniq(245,732)` — 245 registered to NAG (*e.g.*)

ANDF supports install-time code parameterization

Not only identifier references, but also values, concrete types (shapes), and even macros must often be deferred to installation. Examples are configuration parameters and data types for libraries (header files in C), where different implementations may have different definitions.

This requirement is completely foreign to compiler intermediate languages, which typically expand out data definition early on.,

ANDF tokens provide general syntax-macro functionality. Syntax macros, like those found in ANDF or Lisp, and unlike the lexical macros found in C and other languages, operate on syntactically meaningful program fragments such as expressions and type definitions.[1] ANDF provides the additional, critical, capability of separating token declaration (similar to C prototypes) from token definition, and allows definitions to be deferred to installation time.

Tokens thus become an abstraction mechanism allowing target-independent specification of interfaces in the source code. The target-dependent

1. See B.M. Leavenworth, “Syntax Macros and Extended Translation” *Commun. ACM* **9**:11:790 (November, 1966).

implementation of types and macros is “linked in” at installation time, just like library procedures.

Producers may also use tokens to translate common constructions, thus reducing ANDF size, just as regular macros can be used as shorthands.

Tokens are also an important mechanism for extensions to ANDF.

Example:

> In most Unix implementations, getchar is a macro, for efficiency. However, the text of this macro varies from implementation to implementation.
>
> The Token mechanism allows leaving the macro call in the intermediate code, to be substituted at installation. This preserves both neutrality and efficiency.

ANDF supports static conditionals (conditional compilation)

An important mechanism used for writing portable code is conditional compilation. Traditional compilers eliminate such static conditionals in the preprocessor. This of course is incompatible with ANDF’s goals. ANDF thus provides for explicit conditionals. Code is then selected at install time through dead code elimination.

C version:

```
#if VAX
  printf(“Running on a Vax”);
#endif
```

ANDF version:

```
exp_cond(
     integer_test_i(equal,...,
                <<VAX>>),
     <<printf(“Running on a Vax”);>> ,
     make_void())
```

7. Control structure

Control structures are escapes from blocks

ANDF provides primitives with which conventional control structures may be built. Most classic control structures (do, for, while) are represented as blocks with escapes, rather than as arbitrary goto's. This simplifies the design of the ANDF producer. ANDF installers may use ANDF-to-ANDF transformations to reduce these to the equivalent goto's.

8. Beyond C

ANDF supports conventional languages

The features of ANDF presented so far are common to all languages, and sufficient for most conventional sequential procedural languages. Although examples have been drawn from C, the same mechanisms suffice for other languages with similar run-time structures, such as C++, Algol 60, Fortran, Pascal, ….

Some conventional sequential languages may need additional ANDF constructs or standard token libraries. Cobol and PL/I, for instance, may benefit from special support for decimal arithmetic.

Other languages may be covered in future extensions

The design process leading to ANDF covered a wide spectrum of languages, from C to Ada, Common Lisp, and ML. The base constructs present in the current version of ANDF are believed to be appropriate for all these languages.

If and when other constructs or standard token libraries become necessary to support other languages, they may be added to the definition of ANDF after technical and business analysis. Of course, full upward compatibility will always be preserved.

9. Binary form is compact but flexible

The presentation form of ANDF used so far in this paper is designed for easy reading by engineers. ANDF's machine form, on the other hand, is designed to be a compact and efficient encoding for storage, transmission, and processing. It is isomorphic to the presentation form, but is highly compressed. Several techniques are used for compression.

Bit-packed encoding

ANDF is fully packed.

Encodings for elementary constants and constructors whose maximum size is known *a priori* are encoded bit-efficiently. For instance, there are only four possible values for a boolean constant: false, true, Token-Nat (a local symbolic constant), and Token-Unique (a global symbolic constant). Thus the encoding is performed in 2 bits (with a following Token code if necessary).

No arbitrary size limits

Fields whose maximum size cannot be known *a priori* have a size determined in the header for the ANDF capsule. For instance, if there are 423 different tags in a given capsule, precisely 9 bits will be reserved to specify each tag for that capsule.

Unlike conventional approaches which must guess at the maximum useful size of a field, and then apply this maximum to all uses, including those that need only a small fraction, space is not wasted, nor is capacity limited.

Tree structure is implicit

Constructs are represented in Polish prefix form. Since most constructs are of fixed arity, no explicit parenthesization is needed. Constructs of variable arity are immediately preceded by a length specification.

Tokens compress common subtrees

The Token mechanism described as a portability feature is also used to compress commonly-occurring idioms. For instance, the C construct `x++` is represented in ANDF as `(temp = x; x = x+1; temp)`. Rather than copy this tree wherever `x++` is written, an ANDF producer for C may use a token which expands to the full definition. The reference producer does this.

Extensible

Despite its compactness, the binary form is extensible. All potentially extensible fields include an escape value, which allows extending to additional values in the future.

10. Conclusion

ANDF was specifically designed as a multi-language, multi-architecture portability and distribution format.

Unlike adaptations of existing compiler intermediate languages, it is not distorted by a language- or machine- specific history.

Unlike low-level programming languages, it does not need to make concessions to human users.

Unlike either, its semantics are defined by a specification, and not by particular machines or tools.

On the contrary, its designers have produced a wide-spectrum intermediate language with many desirable properties:

- Uniform structure
- Language independence
- Machine independence
- Complete and unambiguous definition
- Compact encoding
- Extensibility

For further information please contact:

Stavros Macrakis
macrakis@osf.org
(617) 621-7356