A TASTY Alternative

Version 0.07
22 February 2017
Martin Odersky
Eugene Burmako
Dmytro Petrashko

This document proposes a new serialization format for typed syntax trees of Scala programs. It aims to be

- **compact**: All numbers and references are length encoded, trees are inlined by default but may be shared.
- **lazy**: Trees can be explored from the top, and one can suspend reading subtrees at any time. The only section that needs to be scanned in full is the name table. Laziness is important for the compiler frontend as most of the information in a TASTY file is not needed then.
- **extensible**: New sections can be defined at will. Additional elements always carry their length, so can be safely skipped in older readers.
- **precise**: The format given here is in essence a serialized abstract syntax tree for typed Scala. Some of the main classifications, e.g. between types, terms and paths are reflected in the grammar.

Picklers and unpicklers for the format have been implemented in https://github.com/lampepfl/dotty/pull/394.

The present document gives a syntax summary of the format. More explanations are found in the TASTY reference manual.

**Notation:**

We use BNF notation. Terminal symbols start with at least two consecutive upper case letters. Each terminal is represented as a single byte tag. Non-terminals are mixed case. Prefixes of the form *lower case letter*_ are for explanation of semantic content only, they can be dropped without changing the grammar.

**Micro-syntax:**

```
LongInt       = Digit* StopDigit // big endian 2's complement, fits in a Long w/o overflow
Int           = LongInt // big endian 2s complement, fits in an Int w/o overflow
Nat           = LongInt // non-negative value, fits in an Int without overflow
Digit         = 0 | ... | 127
StopDigit     = 128 | ... | 255 // value = digit - 128
```
Macro-format:

File          = Header majorVersion_Nat minorVersion_Nat UUID
   nameTable_Length Name* Section*
Header        = 0x5C 0xA1 0xAB 0x1F
UUID          = Byte*16 // random UUID
Section       = NameRef Length Bytes
Length        = Nat // length of rest of entry in bytes
Name          = UTF8 Length UTF8-CodePoint* QUALIFIED Length qualified_NameRef selector_NameRef
   SIGNED Length original_NameRef resultSig_NameRef paramSig_NameRef*
   EXPANDED Length original_NameRef
   OBJECTCLASS Length object_NameRef
   SUPERACCESSOR Length accessed_NameRef
   DEFAULTGETTER Length method_NameRef paramNumber_Nat
   SHADOWED Length original_NameRef
   MANGLED Length mangle_NameRef name_NameRef
...
NameRef       = Nat // ordinal number of name in name table, starting from 1.

Note: Unqualified names in the name table are strings. The context decides whether a name is
a type-name or a term-name. The same string can represent both.

Standard-Section: "ASTs" TopLevelStat*

TopLevelStat  = PACKAGE Length Path TopLevelStat*
   Stat
Stat          = Term Length NameRef Type rhs_Term Modifier*
   VALDEF Length NameRef Type rhs_Term Modifier*
   DEFDEF Length NameRef TypeParam* Params* return_Type rhs_Term
   Modifier*
   TYPEDEF Length NameRef (Type | Template) Modifier*
   IMPORT = IMPORTED Length qual.Term Selector*
   RENAMED Length from_NameRef to_NameRef
   // Imports are for scala.meta, they are not used in the backend
Selector      = IMPORTED NameRef
   RENAMED Length from_NameRef to_NameRef
   // Imports are for scala.meta, they are not used in the backend
TypeParam     = TYPEPARAM Length NameRef Type Modifier*
Params        = PARAMS Length Param*
Param         = PARAM Length NameRef Type rhs_Term Modifier*
   // rhs_Term is present in the case of an aliased class parameter.
Template      = TEMPLATE Length TypeParam* Param* Parent* Self? Stat*
   // Stat* always starts with the primary constructor.
Parent = Application
Type
Self = SELFDEF
    selfName_NameRef selfType_Type
Term = Path
    Application
    IDENT NameRef Type // used when ident's type is not a TermRef
    SELECT possiblySigned_NameRef qual_Term
    NEW cls_Type
    SUPER Length this.Term mixinTrait_Type?
    PAIR Length left.Term right.Term
    TYPED Length expr.Term ascription_Type
    NAMEDARG Length paramName_NameRef arg.Term
    ASSIGN Length lhs.Term rhs.Term
    BLOCK Length expr.Term Stat*
    IF Length cond.Term then.Term else.Term
    CLOSURE Length meth.Term target_Type env.Term*
    MATCH Length sel.Term CaseDef*
    RETURN Length meth_ASTRef expr.Term?
    TRY Length expr.Term CaseDef* finalizer.Term?
    REPEATED Length elem_Type elem.Term*
    BIND Length boundName_NameRef patType_Type pat.Term
    ALTERNATIVE Length alt.Term*
    UNAPPLY Length fun.Term ImplicitArg* patType_Type pat.Term*
    SHARED Application = APPLY Length fn_Term arg.Term*
        TYPEAPPLY Length fn_Term arg.Type*
    CaseDef = CASEDEF Length pat.Tree rhs.Term guard.Term?
    ImplicitArg = IMPLICITArg Length arg.Term
    ASTRef = Nat // byte position in AST payload

Path = Constant
    TERMREFdirect sym_ASTRef
    TERMREFsymbol sym_ASTRef qual_Type
    TERMREFpkg fullyQualified_NameRef
    TERMREF possiblySigned_NameRef qual_Type
    THIS clsRef_Type
    SKOLEMtype refinedType_ASTRef
    SHARED path_ASTRef

Constant = UNITconst
    FALSEconst
    TRUEconst
    BYTEconst Int
    SHORTconst Int
    CHARconst Nat
    INTconst Int
    LONGconst LongInt
FLOATconst Int
DOUBLEconst LongInt
STRINGconst NameRef
NULLconst
CLASSconst Type
ENUMconst Path

Type = Path
TYPEREFdirect sym_ASTRef
RecTThis type_ASTRef
TYPEREFsymbol sym_ASTRef qual_Type
TYPEREFpkg fullyQualified_NameRef
TYPEREF possiblySigned_NameRef qual_Type
SUPERType Length this_Type underlying_Type
REFINEDtype Length underlying_Type refinement_NameRef info_Type
APPLIEDtype Length tycon_Type arg_Type*
TYPEBOUNDS Length low_Type high_Type
TYPEALIAS Length alias_Type (COVARIANT | CONTRAVARIANT)?
ANNOTATED Length fullAnnotation_Term underlying_Type
ANDtype Length left_Type right_Type
ORtype Length left_Type right_Type
BIND Length boundName_NameRef bounds_Type
    // for type-variables defined in a type pattern
BYNAMEtype Length underlying_Type
POLYtype Length result_Type NamesTypes    // needed for refinements
METHODtype Length result_Type NamesTypes    // needed for refinements
PARAMtype Length binder_ASTref paramNum_Nat // needed for refinements
NOTYPE
SHARED
    type_ASTRef

NamesTypes = ParamType*
NameType = paramName_NameRef typeOrBounds_ASTRef

Modifier = PRIVATE
    INTERNAL    // package private
    PROTECTED
PRIVATEqualified qualifier_Type    // will be dropped
PROTECTEDqualified qualifier_Type    // will be dropped
ABSTRACT
FINAL
SEALED
CASE
IMPLICIT
LAZY
OVERRIDE
INLINE    // macro
ABSOVERRIDE    // abstract override
STATIC    // mapped to static Java member
OBJECT  // an object or its class
TRAIT   // a trait
LOCAL   // private[this] or protected[this]
SYNTHETIC // generated by Scala compiler
ARTIFACT // to be tagged Java Synthetic
MUTABLE  // a var
LABEL   // method generated as a label
FIELDAccessor // getter or setter
CASEAccessor // getter for case class param
COVARIANT // type param marked "+"
CONTRAVARIANT // type param marked "-"
SCALA2X  // Imported from Scala2.x
DEFAULTParameterized // Method with default params
INSUPERCALL  // defined in the argument of a constructor supercall
Annotation

Annotation = ANNOTATION Length tycon_Type fullAnnotation_Term

Note: Tree tags are grouped into 5 categories that determine what follows, and thus allow to compute the size of the tagged tree in a generic way.

Category 1 (tags 0-63): tag
Category 2 (tags 64-95): tag Nat
Category 3 (tags 96-111): tag AST
Category 4 (tags 112-127): tag Nat AST
Category 5 (tags 128-255): tag Length <payload>

Standard Section: "Positions" sourceLengthNat Assoc*

Assoc = addr_Delta offset_Delta offset_Delta?
       // addr_Delta :
       // Difference of address to last recorded node.
       // All but the first addr_Deltas are > 0, the first is >= 0.
       // 2nd offset_Delta:
       // Difference of end offset of addressed node vs parent node. Always <= 0
       // 1st offset Delta, if delta >= 0 or 2nd offset delta exists:
       // Difference of start offset of addressed node vs parent node.
       // 1st offset Delta, if delta < 0 and 2nd offset delta does not exist:
       // Difference of end offset of addressed node vs parent node.
       // Offsets and addresses are difference encoded.
       // Nodes which have the same positions as their parents are omitted.

Delta  = Int  // Difference between consecutive offsets / tree addresses,