ART Memo 31 (Version 2)
The Definition of the Augmented Redex Trail

Olaf Chitil

April 16, 2002

1 Introduction

Definition of the new trace structure, the Augmented Redex Trail, and its file representation.

2 The Structure

```
data Ref a   = Ref FilePos | Root | Unevaluated | Entered | Lambda

data Module = Module     {name :: String, file :: String, traced :: Bool}

data SrcPos = SrcPos     {module :: Ref Module, position :: Int}

data Exp    = App        { parent :: Ref Exp, use :: Ref SrcPos
                         , result :: Ref Exp
                         , fun    :: Ref Exp, args :: [Ref Exp] }
            | ValueApp   { parent :: Ref Exp, use  :: Ref SrcPos
                         , fun    :: Ref Atom, args :: [Ref Exp] }
            | Char       { parent :: Ref Exp, use :: Ref SrcPos, char :: Char }
            | Int        { parent :: Ref Exp, use :: Ref SrcPos, int :: Int }
            | Integer    { parent :: Ref Exp, use :: Ref SrcPos, integer :: String }
            | Rat        { parent :: Ref Exp, use :: Ref SrcPos
                         , num :: Int , denom :: Int}
            | Rational   { parent :: Ref Exp, use :: Ref SrcPos
                         , numerator :: String , denominator :: String }
            | Float      { parent :: Ref Exp, use :: Ref SrcPos, float :: Float }
            | Double     { parent :: Ref Exp, use :: Ref SrcPos, double :: Double }
            | ValueUse   { parent :: Ref Exp, use :: Ref SrcPos, value :: Ref Atom }
            | ConstUse   { parent :: Ref Exp, use :: Ref SrcPos, const :: Ref Exp }
            | ConstDef   { context :: Ref Exp
                         , result  :: Ref Exp, var :: Ref Atom }
            | Guard      { parent :: Ref Exp, use  :: Ref SrcPos
                         , result :: Ref Exp, cond :: Ref Exp }
            | Case       { parent :: Ref Exp, use  :: Ref SrcPos
                         , result :: Ref Exp, cond :: Ref Exp }
            | If         { parent :: Ref Exp, use  :: Ref SrcPos
                         , result :: Ref Exp, cond :: Ref Exp }
            | FieldUpdate{ parent  :: Ref Exp    , use     :: Ref SrcPos
                         , result  :: Ref Exp    , arg     :: Ref Exp
                         , binders :: [Ref Atom], bindees :: [Ref Exp] }
            | Projection { parent :: Ref Exp, use :: Ref SrcPos
                         , exp    :: Ref Exp }
            | Hidden     { parent :: Ref Exp, result :: Ref Exp }
            | Forward    { result :: Ref Exp }
            | DoStmt     { stmt :: Ref Exp, use :: Ref SrcPos }

data Atom = Variable           { def    :: SrcPos
                               , fixity :: Byte   , arity :: Byte
                               , local  :: Bool   , name :: String }
          | Constructor        { def    :: SrcPos
                               , fixity :: Byte   , arity :: Byte
                               , name   :: String }
          | ConstructorWFields { def    :: SrcPos
                               , fixity :: Byte   , arity :: Byte
                               , name :: String , fields :: [Ref Atom] }
          | Abstract           { description :: String }
```


Nodes

The main data constructors for expressions are *App*, *ValueUse*, *ConstUse* and *ConstDef*. Whereas applications are straightforward we have to distinguish two kinds of constant expressions: those that are values and those that are variables of arity zero. The former are represented by *ValueUse*; the *value* field points to any kind of atom except a variable of arity zero. For the representation of the latter both *ConstUse* and *ConstDef* are used. For every use of such a constant a *ConstUse* is created. Its *const* field points to a *ConstDef* which represents the reduction and which is shared between many uses. Its *context* field points to the redex of the enclosing scope of the definition, which is *Root* for a top-level variable. Its *var* field points to a *Variable* of arity 0.

Another expression is added for update with named fields. After evaluation the values of argument and result are applications of the same constructor.

The *Projection* node gives a new parent and source reference to any expression that is the result of a projection.

A *Hidden* node is always the result of the *App* or *ConstDef* of an untraced variable. It thus marks that information of the full computation is missing at this point in the trace.

A *Forward* node appears in a data structure that is the result of an untraced computation. It appears between all subexpressions of the data structure. It is only needed to enable representation of cyclic data structures. Unfortunately we do not know any method for reducing their number.

A *DoStmt* just marks an expression as an action of a do-block.

Evolution of node entries

A *result* field is *Unevaluated* when an *Exp* node is recorded in the trace. When the value of the *Exp* node is demanded, the field turns into *Entered*; at the point when the *Exp* node is reduced, the field is turned into a reference to the reduct. Despite the existence of a reduct the value of the *Exp* may still be $\perp$. A *ValueUse* node, the nodes for specific literals and a *ConstUse* node do not have a *result* field. For a *ValueUse* node the result is the *value*, for the other specific literal nodes similarly, and for a *ConstUse* node the result is obtained via the *const* field from the *ConstDef* node.

Constraints

Root is the parent of every top-level caf, that is, variable of arity 0.

In general a *result* field may be *Unevaluated*, *Entered* or a proper file position. A *parent* field may be *Root* or a proper file position. All other reference fields may only be proper file positions.

The *value* field is not used in three stages like a *result* field, because it shall be possible to obtain the value even when it was never demanded, for example for verbose display.

Use source position

In addition to the previous description there are two variants of *Exp*: one with fields exactly as given and one with these fields except for the *use* source reference. This will save space for untraced code where no use source references are recorded. A source reference will always point to a *SrcPos* node and the source positions in *Atom*s are never bogus.

Atoms

A *Lambda* is of type *Ref Atom* and may only appear as *value* of a *ValueUse*. Similar to a *Variable* it is the name of a λ-abstraction. This way it is more convenient to record than if it were another kind of *Exp*.

Large numbers are stored as strings. This format is portable and easy to read and write, although a bit costly in space and time. However, small numbers are stored efficiently.

Field names are stored with constructors, so that constructor expressions can be displayed with field names. Field names are the same as selector variables.

Abstract represents the values of an abstract data type (the string giving the type constructor) and is used for storing the error message of an aborted computation.

3 Obtaining Equations from an ART

The following code gives an idea on how references to trace nodes can be transformed into printable equations, (unevaluated) expressions and values. The code does not handle cycles and possibly some arguments should better raise an error.

Warning: outdated!!!

```
eqn :: Ref Exp -> SEqn
eqn ref = SEqn (exp ref) (val ref)

exp :: Ref Exp -> SExp
```



```
exp Unevaluated = SUnevaluated
exp Entered = SBottom
exp (Ref pt) = let e = getNode trace pt in
  case e of
    App{} -> SApp (val (fun e) : map val (args e))
    ValueUse{} -> atom (value e)
    ConstUse{} -> exp (const e)
    ConstDef{} -> atom (var e)
    Guard{} -> SGuard (exp (cond e))
    Case{} -> SCase (exp (cond e))
    If{} -> SIf (exp (cond e))
    Projection{} -> exp (exp e)
    Hidden{} -> SHidden
    Forward{} -> exp (result e)

val :: Ref Exp -> SExp
val Unevaluated = SUnevaluated
val Entered = SBottom
val (Ref pt) = let e = getNode trace pt in
  case e of
    ValueUse{} -> if entered e || verbose
                    then val (value e)
                    else SUnevaluated
    ConstUse{} -> if entered e
                       then val (const e)
                       else if verbose
                              then exp (const e)
                              else SUnevaluated
    Projection{} -> val (exp e)
    _ -> case (result e) of
           Unevaluated -> if verbose then exp e else SUnevaluated
           Entered -> SBottom
           ref@(Ref resPt) ->
             if resPt == pt
               then exp e -- this is the value, only relevant for App
               else val ref

atom :: Ref Atom -> SExp
```

4 The File Format

The ART file is a sequence of nodes, each of which is a sequence of bytes. There are several types of node, of varying size. Every node starts with a tag byte that identifies the node.

To slightly simplify access, fields are ordered so that varying length fields are at the end of a node.

A fileoffset is a pointer of 4 bytes. *Root* is represented by 0, *Unevaluated* is represented by 1 and *Entered* is represented by 2. A *Lambda* is represented by 4. Proper fileoffsets are greater than 4. Currently the highest bit is always 0, so that 2 GB can be addressed.

All data is stored in network byte order.

A boolean field is simply represented by a bit in the tag byte of a node.

A list is represented as a byte giving its length followed by its elements. Hence it can at most have 255 elements. So a string is represented as a byte giving its length followed by the characters of the string, without terminating `\0`. For strings longer than 254 characters the length byte is 255 and it is followed by a two byte length (highest first) succeeded by the characters. Thus strings can have a length of 65536 and may contain `\0`.

Below the *use* field is not mentioned. If Bit 6 is set, then a *use* field is the *first* field of a node, in front of the remaining fields.

The lower 5 bits of the tag byte are used to identify the node sort as given below. Bit 5 is used for the boolean field that some nodes have (each has at most one). Bit 6 is used to indicate the presence of a *use* field (only in *Exp* nodes). Bit 7 is free to be used by hat-check.

File header at beginning of file

8 literal character bytes (`"Hat1.13\0"`)
+ fileoffset (0 or error entry point for trace : Ref Exp)
+ fileoffset (0 or 2 (interrupted) or error explanation :: Ref (Atom.Abstract)

Module

Tag 0:	1^+ bytes (module name length n) + n bytes (module name :: [Char]) + 1^+ bytes (filename length m) + m bytes (module filename :: [Char]) Bit 5 = traced

SrcPos

Tag 1:	fileoffset (module info :: Ref Module) + 4 bytes (line * 10000 + column)

Exp

Tag 2:	App	fileoffset (parent :: Ref Exp)	+ fileoffset (result :: Ref Exp) + fileoffset (fun :: Ref Exp) + 1 byte (arity) + arity * fileoffset (args :: [Ref Exp])
		Bit 6 = with use srcpos	
Tag 3:	App	fileoffset (parent :: Ref Exp)	+ fileoffset (fun :: Ref Atom) + 1 byte (arity) + arity * fileoffset (args :: [Ref Exp])
		Bit 6 = with use srcpos	
Tag 4:	Char	fileoffset (parent :: Ref Exp)	+ 1 byte (char :: Char)
		Bit 6 = with use srcpos	
Tag 5:	Int	fileoffset (parent :: Ref Exp)	+ 4 bytes (int :: Int)
		Bit 6 = with use srcpos	
Tag 6:	Integer	fileoffset (parent :: Ref Exp)	+ 1^+ bytes (length) + length bytes (integer :: [Char])
		Bit 6 = with use srcpos	

Tag 7:	Rat	fileoffset (parent :: Ref Exp)	+ 4 bytes (num :: Int) + 4 bytes (denom :: Int)
		Bit 6 = with use srcpos	
Tag 8:	Rational	fileoffset (parent :: Ref Exp)	+ 1^+ bytes (numerator length) + n length bytes (numerator :: [Char]) + 1^+ bytes (denominator length) + d length bytes (denominator :: [Char])
		Bit 6 = with use srcpos	
Tag 9:	Float	fileoffset (parent :: Ref Exp)	+ 4 bytes (float :: Float)
		Bit 6 = with use srcpos	
Tag 10:	Double	fileoffset (parent :: Ref Exp)	+ 8 bytes (double :: Double)
		Bit 6 = with use srcpos	
Tag 11:	ValueUse	fileoffset (parent :: Ref Exp)	+ fileoffset (value :: Ref Atom)
		Bit 6 = with use srcpos	
Tag 12:	ConstUse	fileoffset (parent :: Ref Exp)	+ fileoffset (const :: Ref Exp)
		Bit 6 = with use srcpos	
Tag 13:	ConstDef	fileoffset (parent :: Ref Exp)	+ fileoffset (result :: Ref Exp) + fileoffset (var :: Ref Atom)
		Bit 6 = 0	
Tag 14:	Guard	fileoffset (parent :: Ref Exp)	+ fileoffset (result :: Ref Exp) + fileoffset (cond :: Ref Exp)
		Bit 6 = with use srcpos	
Tag 15:	Case	fileoffset (parent :: Ref Exp)	+ fileoffset (result :: Ref Exp) + fileoffset (cond :: Ref Exp)
		Bit 6 = with use srcpos	
Tag 16:	If	fileoffset (parent :: Ref Exp)	+ fileoffset (result :: Ref Exp) + fileoffset (cond :: Ref Exp)
		Bit 6 = with use srcpos	
Tag 17:	FieldUpdate	fileoffset (parent :: Ref Exp)	+ fileoffset (result :: Ref Exp) + fileoffset (arg :: Ref Exp) + 1 byte (arity) + arity fileoffsets (binders :: [Ref Atom]) + arity fileoffsets (bindees :: [Ref Exp])
		Bit 6 = with use srcpos	
Tag 18:	Projection	fileoffset (parent :: Ref Exp)	+ fileoffset (exp :: Ref Exp)
		Bit 6 = with use srcpos	
Tag 19:	Hidden	fileoffset (parent :: Ref Exp)	+ fileoffset (result :: Ref Exp)
		Bit 6 = 0	
Tag 20:	Forward	fileoffset (result :: Ref Exp)	
		Bit 6 = 0	
Tag 21:	DoStmt	fileoffset (stmt :: Ref Exp)	
		Bit 6 = with use srcpos	

Atom

Tag 26:	Variable	fileoffset (module info :: Ref Module) + 4 bytes (line * 10000 + column) + 1 byte (fixty) + 1 byte (arity) + 1^+ bytes (name length) + name length bytes (name :: [Char])

	Bit 5 = locally defined
Tag 27: Constructor	fileoffset (module info :: Ref Module) + 4 bytes (line * 10000 + column)
	+ 1 byte (fixty) + 1 byte (arity)
	+ 1^+ bytes (name length) + name length bytes (name :: [Char])
	(+ arity fileoffsets (fields :: Ref Atom))
	Bit 5 = with fieldnames
Tag 28: Abstract	1^+ bytes (description length) + descr. length bytes (description :: [Char])

5 Deferred Extensions

Here some extensions are outlined which might be considered later.

5.1 Children of Hidden Reductions

For the construction of an EDT from an ART it needs to be determined for every *Hidden* node if it has children and which these children are. Currently the only method of finding these children is a linear search of the file. The children have the given *Hidden* node as parent.

Note that for every untraced reduction the algorithmic debugger comes across it has to determine if the node has children: if the call of an untraced variable from traced code has no children, then no question is asked about it. Only if it a question is asked and answered with *no*, then the set of children is needed.

There are various options for obtaining the necessary information.

1. For every *Hidden* node the algorithmic debugger comes across the remaining trace is linearly searched.

2. Before algorithmic debugging the trace is once linearly searched for all children of all *Hidden* nodes. The information is stored separately, possibly in main memory. It may be large.

3. A *Hidden* node has an additional bit that states if the node has children. Only if it has children and the user enquires about the reduction the algorithmic debugger has to linearly search the remaining trace.

4. A *Hidden* node has a list of its traced children. The list is constructed from *Child* nodes and terminated by *Root*. Possibly the result of the untraced computation is also recorded as a child.

```
data Exp   = ...
           | Hidden { parent :: Ref Exp
                    , result :: Ref Exp, children :: Ref Child }
data Child = Child  { child  :: Ref Exp, sibling :: Ref Child }
```

5.2 Entered Values

Many *Exps* always denote values (*ValueApp*, *Char*,*Int*,*Integer*,*Rat*,*Rational*,*Float*,*Double*,*ValueUse*,*Const* Often they correspond to values appearing literally in the program source. However, theses values may never have been demanded by a computation. To distinguish these two sorts of

values these *Exps* could have an additional boolean field *entered* (bit 5) to mark that they were demanded. Viewing tools will usually not show non-demanded values.

5.3 Free Variables

Locally defined variables may have free variables. The identifiers of these free variables are stored once, with the local variable. The expressions of these free variables are stored in another node, which lives between *Atoms* and *Exps*.

```
data Id   = Id              { position :: Int, name :: String }
data Atom = ...
          | LocalVariable { def    :: SrcPos
                          , fixity :: Byte   , arity    :: Byte
                          , name   :: String , freeVars :: [Id] }
          | Closure       { var :: Ref Atom, freeVars :: [Ref Exp] }
```

It is unclear if *Ids* should be shared and if they should have a context reference to determine the variable defined by the enclosing definition. The names of free variables are not unique and confusion is easily possible.

Which variables should be considered as free is a non-trivial issue separate from the definition of the ART. It is discussed in Section 7.1 of Henrik's thesis. Basically only λ-bound variables should be free variables. However the free variables of a let-bound variable also comprises the free variables of the free let-bound variables in its definition. The definition is recursive and the least fixpoint has to be used. Note that the trace transformation considers variables in irrefutable patterns and in $n + k$ patterns as let-bound, but for the purpose of free variables they are λ-bound (a let-bound variant is defined in terms of a λ-bound variant which is introduced by the transformation).