MLRISC Annotations

Allen Leung
New York University
leunga@cs.nyu.edu

Lal George
Bell Laboratories
george@research.bell-labs.com

1 Introduction

In the original Lisp 1.5 by John McCarthy[MAE+62], every atomic symbol had an associated property list where useful information about the symbol could be stored. The property list was automatically created when the symbol was read for the first time. Each property was preceeded by an *indicator*, some of which were:

- PNAME — the BCD print name for input-output use.
- SUBR — a function pointer defined by a machine language subroutine.
- APVAL — the permanent value for the atomic symbol considered as a variable.

The function `get(x,i)` was used to find a property of `x` whose indicator is `i`, and there were similar functions to remove and add new properties.

The `Annotations` module written in Standard ML, serves a similar purpose and is used extensively for exchanging information between different phases in the MLRISC customizable code generator [Geo97]. Different optimization phases may attach annotations to individual instructions, basic blocks, hyperblocks, or compilation units. These annotations may then be read and updated in subsequent phases. For instance, annotations can be used to propagate comments from the program text to the assembly output.

This document describes the interface to the annotation mechanism and its use within the MLRISC system.

2 Interface

Figure 1 shows the interface to the `Annotations` modules. A property is represented by the type `annotation`, and a property list is a collection of annotations. The function `new` takes an optional pretty printing function as argument, and returns a set of functions to create and manipulate properties. They are:

`create` : Creates an annotation.

`rmv` : Removes the annotation from a property list.

`set` : Adds a new annotation to a property list, or replaces an existing one.

```
signature ANNOTATIONS = sig
   exception NoProperty

   type annotation
   type annotations = annotation list
   type propList = annotations
   type 'a property =
        { get      : annotations -> 'a option,
          peek     : annotation  -> 'a option,
          lookup   : annotations -> 'a,
          contains : annotations -> bool,
          set      : 'a * annotations -> annotations,
          rmv      : annotations -> annotations,
          create   : 'a -> annotation
        }
   type flag = unit property

   val new : ('a -> string) option -> 'a property
   val newFlag : string -> flag
   val toString : annotation -> string
end
```

Figure 1: Signature of annotations

`contains` : Returns `true` if the annotation exists in the property list.

`lookup` : Returns the value of the annotation if it exists in the property list; raises `NoProperty`.

`peek` : Returns the value carried by an annotation as an `option` type.

`get` : is like `peek` but works on property lists.

2.1 Simple Example

The example from Lisp 1.5 can be illustrated by the following, where a symbol g is created with an APVAL and PNAME property. First, distinct types are created for each annotation:

```
open Annotations;
datatype pname = PNAME of string
datatype apval = APVAL of real
```

Next, we create annotations that will carry values of each of these types:

```
val pnameAn : pname property = new NONE;
val apvalAn : apval property = new NONE;
fun mkApvalAn(pv) = (#create apvalAn) (APVAL pv)
fun mkPnameAn(pn) = (#create pnameAn) (PNAME pn)
```

mkApvalAn will create an annotation carrying the type apval and likewise for mkPnameAn. Finally, the only thing associated with a symbol is its property list. In the code fragment below, we create a symbol g with an APVAL and PNAME property.

```
datatype symbol = SYM of  propList ref
val g = SYM (ref [mkApvalAn 9.8, mkPnameAn "g"])
```

As a client, one may wish to add additional properties such as a comment to the already existing properties associated with the symbol. This can be easily accomplished using the following:

```
datatype comment = COMMENT of string
val commentAn :comment property = new NONE
fun addComment (SYM props, msg) = props := (#set commentAn)(COMMENT msg,!props)
val _ = addComment(g, "Newton's gravitation constant")
```

2.2 Realistic Example

First we declare a datatype node which represents nodes in a directed graph. Each node carries a list of successors nodes and a property list. A new property called name is defined which holds the name of the nodes. The function newNode can be used to create a new node, and the function setSucc used to set the list of successors.

```
datatype node = NODE of {succ: node list ref, info: Annotations.propList ref}
val name = Annotations.new(SOME(fn s => s))
fun newNode text = NODE{succ=ref [], info=ref [#create name text]}
fun setSucc(NODE{succ,info},nodes) = succ := nodes
```

We can create the graph shown in Figure 2 as follows:

```
val a = newNode("a")
val b = newNode("b")
val c = newNode("c")
val d = newNode("d")
val e = newNode("e")
val _ = setSucc(a, [a,b,c])
val _ = setSucc(c, [d,e])
val _ = setSucc(d, [a])
val _ = setSucc(e, [d])
```

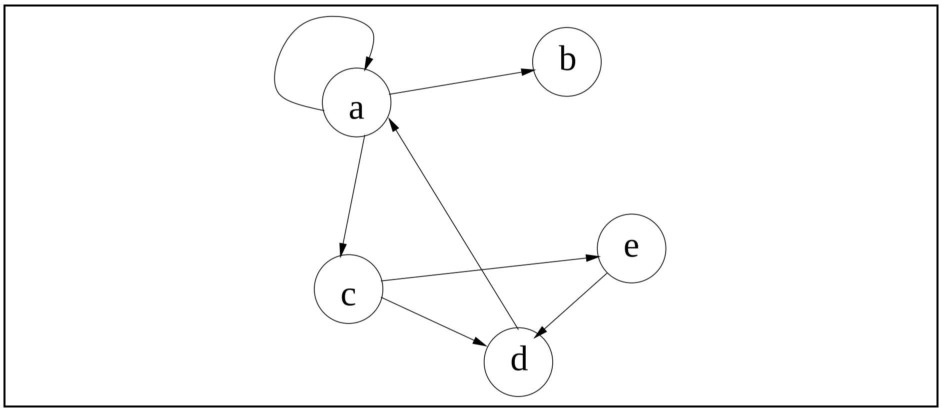

Figure 2: Example graph

Suppose we want to write depth-first search routine that simply visits each node and print out its name. This can be done as follows.

```
fun dfs root =
let val marker = Annotations.new NONE
    fun visit(NODE{succ,info}) =
      if #contains marker (!info) then ()
      else (print("Visiting "^ #lookup name (!info)^ "\n");
          info := #set marker ((),!info);
          app visit (!succ))
in visit root
end
```

In function `dfs`, we generate a new property called `marker`, and use this to indicate whether a node has been visited. For each node being visited, method `#contains` is used to check whether an annotation of type marker already exists. If so, the node has already been visited; otherwise, we print its name and add a new marker to its property list. Note that the function `Annotations.new` is generative and generates a new property each time it is invoked.

When we execute `dfs c`, we get the output:

```
Visiting c
Visiting d
Visiting a
Visiting b
Visiting e
```

3 Annotations in MLRISC

3.1 Basic Annotations

The MLRISC system defines and uses many annotations for its internal use. Several basic properties that are also of interest to an MLRISC client are predefined in the structure `BasicAnnotations`. These are:

`COMMENT` This annotation carries a string as an argument. This annotation has no effect, but can be used to attach comments to various data structures used in the compiler. For example, we can attach comments to instructions generated by the front-end, or attach comments to basic blocks. In the most general case, comments can be propagated from the source code all the way to the final assembly output.

`EXECUTION_FREQ` This annotation carries a real number, representing the execution frequency of a basic block, or a compilation unit.

`BRANCH_PROB` This annotation carries an integer, representing the branch probability (in percentages) of a conditional branch. It can be attached to conditional branch instructions.

`DEFUSER/DEFUSERF/DEFUSECC` These annotations carry a pair of integer lists, representing the definitions and uses of general purpose, floating point and condition code registers. These annotations can be used to override the default definition/use of an instruction. These are mainly used in combination with the user defined pseudo instruction mechanism of MLRISC.

`CTRL` This annotation carries a value of the type `ctrl_dep`, which is defined as:

```
datatype ctrl_dep = CTRL_DEF of int | CTRL_USE of int
```

These annotations can be attached to basic blocks and instructions, in order to propagate control and anti-control dependence to the backend. Intuitively, `CTRL_DEF` represents the definition of a virtual control dependence variable, while `CTRL_USE` represents a use.

`REGINFO` This annotation carries a function of type `int -> string`. The function can be used to annotate the registers in a program so that extra information is printed during assembly output. This mechanism currently provided for debugging the GC-safety mechanism, but may also be used for other purpose by the client. More information on this is discussed in Section 3.3.

`NO_OPTIMIZATION` If attached to a compilation unit, this annotation signals that the compilation unit should not be optimized.

3.2 Instruction Streams

Many phases in MLRISC, such as the assembler, machine code emitter, and instruction selection modules, communicate with each other via an interface called streams. A stream has the interface described in Figure 3.

```
signature INSTRUCTION_STREAM = sig
   structure P : PSEUDO_OPS

   datatype ('a,'b,'c,'d,'e,'f) stream =
      STREAM of
      { beginCluster: int -> 'b,
        endCluster  : 'c -> unit,
        emit        : 'a,
        pseudoOp    : P.pseudo_op -> unit,
        defineLabel : Label.label -> unit,
        entryLabel  : Label.label -> unit,
        comment     : string -> unit,
        annotation  : Annotations.annotation -> unit,
        exitBlock   : 'd -> unit,
        alias       : 'e -> unit,
        phi         : 'f -> unit
      }
end
```

Figure 3: The stream interface

Each stream implements a set of methods `beginCluster`, `endCluster`, `emit`, etc., which can be used to construct a program in linearized form. How these methods behave depend on context, but in general, the method `annotation` can be used to attach an annotation to the current basic block; while the method `comment` can be used to attach a `COMMENT` annotation (again to the current basic block).

3.3 Assemblers

The signature `INSTRUCTION_EMITTER` defined in Figure 4 is used in all MLRISC assemblers and machine code emitters. The function `makeStream` takes a list of annotations that modifies the format of the assembly output. Currently, only the annotation `REGINFO` is recognized. The annotation `REGINFO` carries a function f of type `int -> string`. If `REGINFO` is present in the argument to `makeStream`, when the assembler emits the output for the register r, it will attach the string $f(r)$ to the textual output for r. If $f(r)$ raises an exception, no output will be attached.

For example, in the SML/NJ runtime system certain registers are dedicated. For example, `$13` is reserved as the allocation pointer, while `$9` is the heap limit pointer. Using `REG_INFO`, we can mark these dedicated registers. For example,

```
signature INSTRUCTION_EMITTER = sig
   structure I : INSTRUCTIONS
   structure C : CELLS
   structure S : INSTRUCTION_STREAM
   structure P : PSEUDO_OPS
      sharing I.C = C
      sharing S.P = P

   val makeStream : Annotations.annotations ->
                     ((int -> int) -> I.instruction -> unit,
                      unit,'b,'c,'d,'e) S.stream
end
```

Figure 4: Signature for assemblers

if $f(13)$ = `"(allocptr)"`, $f(9)$ = `"(limitptr)"`, and $f(\mathrm{p})$ = `"(new object)"`. Given the Alpha instructions

```
cmpult $9, $13, t
bne    t, invokegc
...
addq   $13, 4, p
```

the assembler will pretty print these as:

```
cmpult $9 (limitptr), $13 (allocptr), t
bne    t, invokegc
...
addq   $13 (allocptr), 4, p (new object)
```

3.4 MLTree

MLTree is the tree based RTL language that MLRISC uses to communicate between the front-end and the code generator backend. MLTree has the (abbreviated) signature shown in Figure 5.

The constructors `ANNOTATION`, `MARK`, `FMARK`, and `CCMARK` can be used to attach annotations to MLTree statements, integer expressions, floating point expressions, and conditional expressions. These attached annotations are propagated to the generated instructions.

The type `stream` defined in signature `INSTRUCTION_STREAM` is specialized further in the signature `MLTREE`. The stream method `endCluster` takes a list of annotations as arguments. These annotations are global to a compilation unit.

All instruction selection modules in MLRISC are given the interface shown in Figure 6. The function `selectInstruction` maps a stream of MLTree statements into a stream of target instructions.

```
signature MLTREE = sig
  structure Constant : CONSTANT
  structure PseudoOp : PSEUDO_OPS
  structure Region   : REGION
  structure Stream   : INSTRUCTION_STREAM
     sharing Stream.P = PseudoOp

  datatype stm = (* statements *)
      ...
    | BCC of ccexp * Label.label
    | ANNOTATION of stm * Annotations.annotation

  and rexp =  (* integer expressions *)
      LI of int (* load constant integer *)
    | ADD of ty * rexp * rexp
      ...
    | MARK of rexp * Annotations.annotation

  and fexp = (* floating point expressions *)
      ...
    | FMARK of fexp * Annotations.annotation

  and ccexp = (* condition expressions *)
      ...
    | CCMARK of ccexp * Annotations.annotation

  and mlrisc = CCR of ccexp | GPR of rexp | FPR of fexp

  type ('i,'regmap) stream =
       ('i -> unit,'regmap,Annotations.annotations,
        mlrisc list, alias, phi) Stream.stream
end
```

Figure 5: Signature of MLTree

```
signature MLTREECOMP = sig
    structure T : MLTREE
    structure I : INSTRUCTIONS
       sharing T.Constant = I.Constant

    val selectInstructions :
        (I.instruction,I.C.regmap) T.stream ->
        (T.stm,I.C.regmap) T.stream
end
```

Figure 6: Interface of instruction selection modules

In SML/NJ [AM91], a branch that perform a garbage collection test is unlikely to be taken. We can help out the backend by attaching a branch probability annotation to the branch that states that the branch probability is zero:

```
structure T = MLTree
emit(T.ANNOTATION(
       T.BCC(gctest,gcLabel),
       #create BasicAnnotation.BRANCH_PROB 0))
```

In this example, the `BRANCH_PROB` annotation will be attached to the branch instruction generated during instruction selection.

In general, annotations are attached to the instruction that is "closest" to the annotation. For example, when we write

```
T.MV(32, s, T.MARK(T.ADD(32, T.REG(32, r), T.LI 256), a))
```

the annotation `a` will be attached to the add instruction generated. On the Alpha [Sit92] architecture, the follow instruction with the annotation `a` attached will be generated.

```
addl r, 256, s /* a */
```

A similar statement

```
T.MV(32, s, T.ADD(32, T.MARK(T.REG(32, r), a), T.LI 256))
```

generates the following instructions

```
mov  r, t  /* a */
addl t, 256, s
```

where `t` is a temporary. Notice that a move instruction is inserted to hold the annotation.

Multiple annotations may be attached to the same instruction. For example, if we write

```
T.MV(32, s, T.MARK(T.MARK(
     T.LOAD(32, T.ADD(64, T.REG(64, p), T.REG(64, q)),
            addr), a), b)
```

the code generator will emit a load instruction with the annotations `a` and `b` attached.

```
add p, q, t
ldl 0(t), s /* a */ /* b */
```

Since the Alpha does not have indexed addressing modes, the `add` instruction is necessary for computing the intermediate address. The annotations are only attached to the closest instruction, which in this case is `ldl`.

3.5 Instructions

Internally, MLRISC implements instruction level annotations by extending each target machine instruction set with the variant `ANNOTATION`:

```
datatype instruction =
  ...
| ANNOTATION of {i:instruction, a:Annotations.annotation}
```

We can manipulate instruction level annotations through the abstract interface provided in the instruction properties modules ⟨*Arch*⟩`Props` modules, where *Arch* is `Sparc`, `Hppa`, `x86`, `Alpha`, etc. These modules matches the signature `INSN_PROPERTIES` that contains the following two functions for extracting the property list associated with an instruction, and for annotating an instruction.

```
val getAnnotations :
     I.instruction -> I.instruction * Annotations.annotation list
val annotate       :
     I.instruction * Annotations.annotation -> I.instruction
```

Function `getAnnotations` splits an instruction into the instruction part and the property list part. Function `annotate` returns a new instruction marked with an annotation. For example, to mark an instruction `instr` with the annotations from the property list `P`, we can say:

```
val instr' = foldr annotate instr (!P)
```

> User Hints: Remember to handle the variant `ANNOTATION` when manipulating instructions in their concrete form. For example, when writing the client callbacks for the register allocator. □

3.6 Program Flow Graphs

MLRISC currently has two different program representations, clusters and CFGs. Clusters are defined as a set of ML datatypes, while CFGs are implemented using MLRISC Graph library, which is more general and allows more complex transformations [1]. Their signatures are presented in Figure 7 and Figure 8.

Block level annotations are stored in `BBLOCK` (for clusters) and `BLOCK` (for CFG). Global annotations are stored in `CLUSTER` (for clusters) and in `INFO` (for CFG). On the CFG, we can also access these via the functions `getAnnotations` and `setAnnotations`. Note that CFG also support annotations attached to individual control flow edges (the property list is label `a`).

[1] Please see documentation on this library for more details.

```
signature FLOWGRAPH = sig
  structure C : CELLS
  structure I : INSTRUCTIONS
  structure P : PSEUDO_OPS
  structure W : FREQ
     sharing I.C = C

  datatype block =
      PSEUDO of P.pseudo_op
    | LABEL of Label.label
    | BBLOCK of { blknum      : int,
                  freq        : W.freq ref,
                  annotations : Annotations.annotations ref,
                  liveIn      : C.cellset ref,
                  liveOut     : C.cellset ref,
                  succ               : edge list ref,
                  pred               : edge list ref,
                  insns              : I.instruction list ref
                }
    | ENTRY of {blknum : int, freq : W.freq ref,
                succ : edge list ref}
    | EXIT of {blknum : int, freq : W.freq ref,
               pred : edge list ref}
  withtype edge = block * W.freq ref

  datatype cluster =
      CLUSTER of {
        blocks: block list,
        entry : block,
        exit  : block,
        regmap: C.regmap,
        blkCounter : int ref,
        annotations : Annotations.annotations ref
      }
end
```

Figure 7: Signature of a cluster

```
signature CONTROL_FLOW_GRAPH = sig
   structure I : INSTRUCTIONS
   structure P : PSEUDO_OPS
   structure C : CELLS
   structure W : FREQ
      sharing I.C = C

   type weight = W.freq

   datatype block_kind =  ...

   and block =
      BLOCK of
      {  id          : int,
         kind        : block_kind,
         freq        : weight ref,
         data        : data list ref,
         labels      : Label.label list ref,
         insns       : I.instruction list ref,
         annotations : Annotations.annotations ref
      }

   and edge_kind = ...

   and info =
       INFO of
       { regmap      : C.regmap,
         annotations : Annotations.annotations ref,
         firstBlock  : int ref,
         reorder     : bool ref
       }

   and edge_info =
       EDGE of { k : edge_kind,
                 w : weight ref,
                 a : Annotations.annotations ref
               }

   type cfg = (block,edge_info,info) Graph.graph

   ...
   (* Get global annotations *)
   val setAnnotations : cfg * Annotations.annotations -> unit
   val getAnnotations : cfg -> Annotations.annotations
end
```

Figure 8: Siganture of a CFG

3.7 Visualization

MLRISC has a visualization library that translates graphs into graphically viewable form using tools such as VCG [San94] and daVinci [Wer98]. How a graph is presented external is determined by the set of formats defined in the structure `GraphLayout` (see Figure 9).

```
structure GraphLayout = struct
   datatype format =
     LABEL of string
   | COLOR of string
   | NODE_COLOR of string
   | EDGE_COLOR of string
   | TEXT_COLOR of string
   | ARROW_COLOR of string
   | BACKARROW_COLOR of string
   | BORDER_COLOR of string
   | BORDERLESS
   | SHAPE of string
   | ALGORITHM of string
   | EDGEPATTERN of string
   | ...
end
```

Figure 9: Visualization formats

In the same structure, a "style" property is defined. Annotations of this type can be inserted into the CFG representation, and during visualization, these annotations can be used to override the default display format.

```
val STYLE : format list Annotations.property
```

For example, if we insert the annotation

```
#create STYLE [COLOR "blue", LABEL "backedge"],
```

into the property list of a CFG edge, the edge will be displayed in blue with the (extra) label "backedge".

4 Acknowledgements

The implementation of annotations is adopted from the one used in MLTon by Stephen Weeks.

References

[AM91] A. Appel and D. MacQueen. Standard ML of New Jersey. *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, 1991.

[Geo97] L. George. Mlrisc: Customizable and reusable code generators (`http://cm.bell-labs.com/cm/cs/what/smlnj/doc/MLRISC/index.html`). Technical report, Bell Laboratories, Murray Hill, 1997.

[MAE+62] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* The MIT Press, 1962.

[San94] Georg Sander. Graph layout through the VCG tool. Technical Report A03/94, Universität des Saarlandes, October 1994.

[Sit92] R. L. Sites, editor. *Alpha Architecture Reference Manual.* Digital Equipment Corporation, 1992.

[Wer98] M. Werner. *daVinci* v2.1.x online documentation `http://www.tzi.de/~davinci/doc_v2.1`. June 1998.