Generating code for CIRL

J.R.B. Cockett
Department of Computer Science, University of Calgary,
Calgary, T2N 1N4, Alberta, Canada

January 22, 2008

These notes follow Appel's approach to the intermediate representation. It is an expression (or tree) based approach which is well-suited to an implementation in the functional and object oriented style of programming.

The objective of the first two assignments is to get together and develop a simplified back-end for this intermediate representation language. This includes various phases:

A. A parser for CIRL expressions

B. A semantic checker (not required)

C. Code to remove embedded commands

D. An instruction selection phase

E. A register allocation phase for expressions

F. A general register allocation phase

G. Documentation.

The objective of the first assignment is to get the removal of embedded commands completed. Here there is already considerable delicacy over the handling of scope in the language. The objective of the second assignment is to complete the register allocation phase. We shall develop the documentation as we go.

1 CIRL basics

CIRL is a typed intermediate language which can also be used as a stand alone language. It has a minimum of syntactic sugar as it is intended primarily as the target of the front-end of a compiler.

1.1 CIRL program syntax

A CIRL program is a sequence of data definitions, a sequence of function definitions and a CIRL expression:

```
CIRL_prog -> data_defs fun_defs exp.
```

Each data definition starts with the key word `var` and can declare a sequence of arrays and variables of the same type. Arrays declaration must be accompanied by a dimension specifier. The name of a variable so declared is a CIRL location of the specified type. The name of an array, however, is a location of a location of the specified type (thus the actual data of an array is stored in a separate storage block).

```
data_defs -> data_def data_defs |.
data_def -> VAR id_specs : ctype .
id_specs -> ID bounds more_id_specs .
more_id_specs -> , ID bounds more_id_specs | .
bounds -> [ NUM ] | .
```

Next are the function definitions. Function definitions may be arbitrarily nested and can contain local data. The body of a function looks exactly like a CIRL program and the argument list of a function has the same syntax as a declaration list. If an array is passed it must be copied into the local storage ... this is distinguished from passing a location or a pointer to the array.

```
fun_defs -> fun_def fun_defs | .
fun_def -> FUN ID ( data_defs ) : cirl_type [ CIRL_prog ] .
```

CIRL has expressions and commands. Expressions describe a particular calculation of a value while commands determine the program flow and the loading and storing of values. Each commands is ended with a semicolon. There are four commands in CIRL: an assignment, a store, a conditional, and a while loop:

```
com -> ID ASSIGN exp ;
     | loc BACKARROW exp ;
     | IF exp THEN { coms } ELSE { coms } ;
     | WHILE exp DO { coms } ;
coms -> com coms |.
```

CIRL expressions use the locations and the values to compute new locations and values. There are a limited number of operations one can perform on CIRL locations. CIRL locations are either addresses, that is data declared at the outermost level of the program, or local data, that is data declared in a function and so taken as an offset from a frame pointer (the access details for this will be calculated by the system). Given a location one may take an offset from it: if you take an offset the location should be a declared as an array. In addition, a location can be a "dereferenced" expression. This is a type conversion: an expression of type `ptr T`, when dereferenced in this manner, becomes a location of type, `T` (here dereferencing refers only to removing the outer type constructor *not* to retrieving or loading the actual value in memory).

```
loc -> ADDR ( ID )
     | loc OFFSET exp
     | FP ( ID )
     | DEREF( exp )
     | exp.
```

CIRL expressions allow the usual arithmetic and boolean operations. In addition, by "referencing" one can turn a location of type, `T`, into an expression of type `prt T`. Thus, one can pass backward and forward between locations and expressions.

```
exp -> exp OR exp1 | exp1
exp1 -> exp1 AND exp2 | exp2
exp2 -> NOT exp2
      | exp3 compare_op exp3
      | exp3
exp3 -> exp3 addop exp4 | exp4
exp4 -> exp4 mulop exp5 | exp5
exp5  -> SUB exp5
        | { commands } exp
        | MEM ( location )
        | REF ( location )
        | CONST ( bval )
        | TEMP ( ID )
        | ( exp )
        | ID ( exp_list )
addop -> ADD | SUB .
mulop -> MUL | DIV
compare_op -> LEQ|LE|GEQ|GE|EQ|NEQ.
bval -> NUM|REAL|BOOL.
```

The arguments of a function are a comma separated list of expressions (possibly empty):

```
exp_list -> exp more_exp | .
more_exp -> , exp more_exp | .
```

CIRL types are either values of a basic type or locations of a type. In this syntax we omit the "value" constructor:

```
ctype -> CIRLINT | CIRLREAL | CIRLBOOL | CIRLLOC ( ctype ).
```

1.2 Lexemes

The lexer for this parser has:

VAR	=	“var”	ID	=	LETTER (ALPANUM)*
FUN	=	“fun”	OR	=	“||”
AND	=	“&&”	NOT	=	“not”
MEM	=	“mem”	CONST	=	“const”
TEMP	=	“temp”	ADD	=	“+”
REF	=	“ref”	DEREF	=	“deref”
SUB	=	“−”	MUL	=	“*”
DIV	=	“/”	LEQ	=	“<=”
LE	=	“<”	GEQ	=	“>=”
GE	=	“>”	EQ	=	“==”
NEQ	=	“= \ =”	OFFSET	=	“\”
NUM	=	DIGIT (DIGIT)*	REAL	=	(DIGIT)* “·” (DIGIT)*
ASSIGN	=	“:=”	BACKARROW	=	“< −”
IF	=	“if”	THEN	=	“then”
ELSE	=	“else”	WHILE	=	“while”
DO	=	“do”	ADDR	=	“addr”
CIRLINT	=	“int”	CIRLREAL	=	“real”
CIRLBOOL	=	“bool”	LOC	=	“ptr”

1.3 Examples of CIRL programs

Here are some simple example of CIRL programs. They illustrate how one can use CIRL as a basic programming language:

1. Fibonacci numbers:

```
var N:int

function fib(var n:int):int
[ { if mem(fp(n)) < const(2)
    then { t := mem(fp(n)); }
    else { t := fib(mem(fp(n))-const(1))+fib(mem(fp(n))-const(2)); }; } t ]

{ addr(N) <- const(100); } print(fib(mem(addr(N))))
```

2. The greatest common divisor:

```
function gcd(var x:int var y:int):int
 [ { t1 := mem(fp(x)); t2 := mem(fp(y)) ;
     while t1 =/= 0
      { t3:= t1; t1 := gcd(t2 mod t1,t1); t2:= t3; };
   } t2 ]

gcd(const(24),const(92))
```

3. An insertion sort on an array. Note that there need not bean function definitions!

```
var a[100] : int

{ i := 99
; while i >=  0
  { j:= i
  ; while j < 100
      { if mem(mem(addr(a))\j) > mem(mem(addr(a))\j)
        then { x := mem(mem(addr(a))\j)
     ; mem(addr(a))\j <- mem(mem(addr(a))\j+1)
     ; }
 else {}
      ; j:= j+1
      ; }
  ; i := i-1
  ; }
; } i
```

4. A random program!

```
var a,b,c :real
var pb :ptr(real)
var arr [10], noarr :ptr(ptr(int))

function print(var t:real):int
[ const(1) ]

function get():real
[
  {t1 := mem(addr(a)\2);
   t2 := mem(addr(b));
   tp := t1 + t2;
   tmp := mem(addr(a)) * mem(addr(b)) + mem(mem(addr(pb)));
   t2 := mem(addr(c));
   t7 := mem(deref(mem(addr(arr)))\7);
   t9 := ref(addr(b));
   t3 := mem(addr(b));
   t3 := { t4 := mem(addr(a));
           addr(a) <- t4 + t2;
              } t4;
   t6 := t2 + t3;
  } t6
]

{t := get();} print(t)
```

1.4 The abstract syntax tree

A CIRL program consists of data definitions, function definitions and a CIRL expression:

```
data Cirlprog = Cirlprog [DataDef] [FunctionDef] Expression
```

A data definition consists of

```
data DataDef = Var String CirlType
             | Array String Int CirlType
```

A function specification contains a name, a list of parameters, a type specification and the function body. The body minimally consists of an expression, but may contain multiple local data and function declarations.

```
data FunctionDef = Function String [DataDef] CirlType Cirlprog
```

There are four types of command in CIRL.

```
data Command = Assign String Expression
             | Store Location Expression
             | Cond Expression [Command] [Command]
             | While Expression [Command]
```

A location (or a pointer) in CIRL can be either a reference to a global declaration (an `Address`), a reference to a local declaration (an address calculated from the frame pointer `Fp`), an expression, or an explicit address. These can be combined with numerical offsets.

```
data Location =  Address String
               | Fp String
               | Ref Expression
               | Offset Location Expression
```

The expression datatypefor CIRL has memory references, function and operation applications, constant values, embedded commands (which may have side effects), and temporaries:

```
 data Expression = Emem  Location
                 | Eref Location
                 | Eapply Op [Expression]
                 | Econst  BaseValue
                 | Eseq [Command] Expression
                 | Etemp  String
data BaseValue = BValint Int | BValreal Double | BValbool Bool
data Op = Add | Sub | Mul | Div | Mod | And | Or
        | Not | Xor | Teq | Tneq | Tlt | Tgt | Tle
        | Tge | Tls | Trs | Tcall String
```

Finally there are the CIRL types

```
data CirlType = CirlValue BCirlType
              | CirlLoc CirlType
data BCirlType = CirlInt| CirlReal | CirlBool
```

2 CIRL's type system

A CIRL program is strongly typed and its introduced variables (by assignment) have defined scopes. The semantic checking determines whether the program is well-typed and whether each variable which is used is in the scope of an appropriate binding (or declaration) of that variable. The purpose of this section is to lay out the type system of CIRL.

2.1 Basic type checking

CIRL comes with some **basic types** which are ultimately determined by the machine architecture which is being targeted. Here very loosely we are aiming at the Motorola chips architecture which we shall abstract using the Jouette machine described further below.

We shall start by assuming the following basic types:

$$\mathsf{Bool}, \mathsf{Int}, \mathsf{Real}.$$

From these basic types we built a larger set of CIRL types. These are the the **value type** $\mathsf{Value}(T)$, where t is a basic type, and the **location type** $\mathsf{Loc}(S)$, where S is an arbitray CIRL type: a location type holds an address to a location in "main memory" which can be used to retrieve a value. The location type can be iterated so one is allowed to have locations of locations etc. The datatype of CIRL types is thus:

```
data CirlType t = CirlValue t
                | CirlLoc (TCIRL t)
```

In addition CIRL has commands with which, in our first description of the language we shall associate the "imaginary" command type Com. To start the ball rolling we provide ourself with sets of temporaries, $\mathsf{Temp}(S)$ and locations $\mathsf{Loc}(S)$ for each CIRL type S. We shall assume that there are, declared at the outset of a CIRL program (or indeed in a function), certain root locations of variables and arrays and some function symbols (with their definitions).

The basic typing of CIRL expressions and commands is displayed in Table 1; one should view this as being performed on the syntax tree of a CIRL program. CIRL has four basic commands: assignments (to a temporary), stores (to a location), conditionals, and while loops. CIRL expressions include basic binary operations (such as addition, multiplication,etc.), function calls, and the ability to load values from locations (in main storage or on the stack). One can take offsets on locations so that for example one can do an array access (bounds checking should be made implicit in this). Finally, expressions can have **embedded commands** which is a convenient feature for translating language features. As we shall see will see shortly, these cause us some trouble as our very first step is to remove them from the code..

Here are some examples of a CIRL expressions:

1. The following CIRL expression

   $$\{t_0 := \mathsf{mem}(\mathsf{fp}(\mathsf{addr}(\mathtt{i}))); \mathsf{fp}(\mathsf{addr}(\mathtt{i})) \leftarrow \mathsf{temp}(t_0) + \mathsf{const}(1)\}\mathsf{addr}(\mathsf{a})\backslash\mathsf{temp}(t_0) \leftarrow \mathsf{mem}(\mathsf{fp}(c))$$

   is intended to be the translation of the C assignment `a[i++]=c`.

2. One may translate the C conditional expression `x= p?e1:e2` as

   $$\mathsf{fp}(x) \leftarrow \{\mathsf{if}[\![p]\!]\mathsf{then}\{t := [\![e_1]\!]\}\mathsf{else}\{t := [\![e_2]\!]\}\}\mathsf{temp}(t)$$

$$\frac{a \in \mathsf{Address(S)}}{\mathsf{addr}(a) : \mathsf{Loc}(S)} \qquad \frac{n \in \mathsf{Const}(T)}{\mathsf{const}(n) : \mathsf{Value}(T)}$$

$$\frac{t \in \mathsf{Temp}(S)}{\mathsf{temp}(t) : S} \qquad \frac{l : \mathsf{Loc}(S)}{\mathsf{fp}(l) : \mathsf{Loc}(S)}$$

$$\frac{l : \mathsf{Loc}(S) \quad x : \mathsf{Value(Int)}}{l \backslash x : \mathsf{Loc}(S)}\ \text{offset} \qquad \frac{l : \mathsf{Loc}(S)}{\mathsf{mem}(l) : S}\ \text{load}$$

$$\frac{\mathsf{op} : \mathsf{T_1}, \mathsf{T_2} \longrightarrow \mathsf{T} \quad x : \mathsf{Value}(T_1) \quad y : \mathsf{Value}(T_2)}{\mathsf{op}(x, y) : \mathsf{Value}(T)}$$

$$\frac{f \in \mathsf{Functions}(S_1, .., S_n; S) \quad e_i : S_i}{f(e_1, ..., e_n) : S}\ \text{call}$$

$$\frac{l : \mathsf{Loc}(S) \quad v : S}{l \leftarrow v : \mathsf{Com}}\ \text{store} \qquad \frac{t \in \mathsf{Temp}(S) \quad v : S}{t := v : \mathsf{Com}}\ \text{assign}$$

$$\frac{}{\mathsf{Null} : [\mathsf{Com}]} \qquad \frac{C : \mathsf{Com} \quad Cs : [\mathsf{Com}]}{C; Cs : [\mathsf{Com}]}$$

$$\frac{C : [\mathsf{Com}] \quad e : S}{\{C\}e : S}\ \text{eseq}$$

$$\frac{C_1, C_2 : [\mathsf{Com}] \quad e : \mathsf{Value(Bool)}}{\mathsf{if}\ e\ \mathsf{then}\ \{C_1\}\ \mathsf{else}\ \{C_2\}}\ \text{conditional}$$

$$\frac{C : [\mathsf{Com}] \quad e : \mathsf{Value(Bool)}}{\mathsf{while}\ e\ \mathsf{do}\ \{C\}}\ \text{while}$$

Table 1: CIRL Types

We shall usually write just t instead of $\mathsf{temp}(t)$, n instead of $\mathsf{const}(n)$, a instead of $\mathsf{addr}(\mathsf{a})$ when the fact that t is a temporary, that n is a constant, or a is an address is clear.

2.2 Temporary binding and scope in CIRL

The other main task of the semantic checking is to check that whenever a variable is used that it has been correctly introduced. This means that we must understand how variables are bound and what the scope of that binding is. In CIRL there are two mechanisms present for binding names which are independent:

(a) The explicit declaration of names for data locations and functions at the beginning of the main program or a function body;

(b) The introduction of temporary names through an assignment.

The use of declarations to introduce location names is largely standard. CIRL regards all the declarations which are made in the main or a function body to have been made *simultaneously*. Thus, one is *not* allowed to declare items which have the same name in such a declaration sequence. Simultaneous declarations make it easy to declare recursive and mutually recursive functions.

A commands in CIRL can have the effect of changing the variables which are available to be used in expressions. This set of variables with associated types is called a **binding context** or simply a **context** and is written as

$$\Gamma = x : \mathsf{int}, y : \mathsf{real}, p : \mathsf{ptr}(\mathsf{prt}(\mathsf{real})), ...$$

where each variable in the list has an associated type. Expressions are constructed in a particular binding context. Thus,

$$\Gamma \vdash e$$

means that the construction of e can only use, as free variables, those present in the context Γ.

Commands, by contrast, both use the context and can alter the context. This we shall denote by

$$\Gamma \| C \rightsquigarrow \Gamma'$$

which means that in the context Γ the sequence of commands C is valid and, after they are executed they modify the context to Γ'. This allows us now to write down, see table 2, a more sophistiacted version of the type system of CIRL which, in particular, takes into account the dynamics of the introduction of temporaries through assignments.

2.3 Free variables and bindings in CIRL

It is useful to turn the above specification of binding and scope on its head and to ask, for an expression, what variables are needed in the context. There may be variables in the actual context which are not needed to build the expression. The needed variable are called the free variables of the expression and determine the minimal binding context required to build the expression.

The calculation for free variable, see 3 is not complete by itself: it turns out one also needs to know the bindings introduced by commands. This latter computation is displayed in table 4. An expression never introduces a binding. Conditionals and while loops introduce bindings although in a manner which requires a moments thought.

$$\frac{a \in \mathsf{Address(S)}}{\Gamma \vdash \mathsf{addr}(a) : \mathsf{Loc}(S)} \qquad \frac{n \in \mathsf{Const}(T)}{\Gamma \vdash \mathsf{const}(n) : \mathsf{Value}(T)}$$

$$\frac{}{\Gamma, x : S, \Gamma' \vdash \mathsf{temp}(x) : S} \qquad \frac{\Gamma \vdash e : \mathsf{Loc}(S)}{\Gamma \vdash \mathsf{fp}(e) : \mathsf{Loc}(S)}$$

$$\frac{\Gamma \vdash l : \mathsf{Loc}(S) \quad \Gamma \vdash e : \mathsf{Value(Int)}}{\Gamma \vdash l \backslash e : \mathsf{Loc}(S)}\ \text{offset} \qquad \frac{\Gamma \vdash l : \mathsf{Loc}(S)}{\Gamma \vdash \mathsf{mem}(l) : S}\ \text{load}$$

$$\frac{\mathsf{op} : \mathsf{T_1}, \mathsf{T_2} \longrightarrow \mathsf{T} \quad \Gamma \vdash e_1 : \mathsf{Value}(T_1) \quad \Gamma \vdash e_2 : \mathsf{Value}(T_2)}{\Gamma \vdash \mathsf{op}(e_1, e_2) : \mathsf{Value}(T)}$$

$$\frac{f \in \mathsf{Functions}(S_1, .., S_n; S) \quad \Gamma \vdash e_i : S_i}{\Gamma \vdash f(e_1, ..., e_n) : S}\ \text{call}$$

$$\frac{\Gamma \| C \rightsquigarrow \Gamma' \quad \Gamma' \vdash e : S}{\Gamma \vdash \{C\} e : S}\ \text{eseq}$$

$$\frac{\Gamma \vdash l : \mathsf{Loc}(S) \quad \Gamma \vdash v : S \quad \Gamma \| C \rightsquigarrow \Gamma'}{\Gamma \| l \leftarrow v; C \rightsquigarrow \Gamma'}\ \text{store} \qquad \frac{\Gamma \vdash e : S \quad \Gamma \backslash \{x\}, x : S \| C \rightsquigarrow \Gamma'}{\Gamma \| x := e; C \rightsquigarrow \Gamma'}\ \text{assign}$$

$$\frac{}{\Gamma \| \rightsquigarrow \Gamma}$$

$$\frac{\Gamma \vdash e : \mathsf{Value(Bool)} \quad \Gamma \| C_1 \rightsquigarrow \Gamma_1' \quad \Gamma \| C_2 \rightsquigarrow \Gamma_2'; C \quad \Gamma_1' \cap \Gamma_2' \| C \rightsquigarrow \Gamma'}{\Gamma \| \mathsf{if}\ e\ \mathsf{then}\ \{C_1\}\ \mathsf{else}\ \{C_2\}; C \rightsquigarrow \Gamma'}\ \text{conditional}$$

$$\frac{\Gamma \vdash e : \mathsf{Value(Bool)} \quad \Gamma \| C_0 \rightsquigarrow \Gamma'' \quad \Gamma \| C \rightsquigarrow \Gamma'}{\Gamma \| \mathsf{while}\ e\ \mathsf{do}\ \{C_0\}; C \rightsquigarrow \Gamma'}\ \text{while}$$

Table 2: Type rules for temporaries

$$\frac{}{\mathsf{Fv}(\mathsf{addr}(a)) = \{\}} \qquad \frac{}{\mathsf{Fv}(\mathsf{const}(k)) = \{\}}$$

$$\frac{}{\mathsf{Fv}(\mathsf{fp}(a)) = \{\}} \qquad \frac{}{\mathsf{Fv}(\mathsf{temp}(t)) = \{t\}}$$

$$\frac{\mathsf{Fv}(l) = V_1 \quad \mathsf{Fv}(e) = V_2}{\mathsf{Fv}(l \backslash e) = V_1 \cup V_2} \qquad \frac{\mathsf{Fv}(l) = V}{\mathsf{Fv}(\mathsf{mem}(l)) = V}$$

$$\frac{(\mathsf{Fv}(e_i) = V_i)_{i=1...n}}{\mathsf{Fv}(\mathsf{f}(e_1, ..., e_n)) = \bigcup_{i=1...n} V_i}$$

$$\frac{\mathsf{Fv}(C) = V \quad \mathsf{Fv}(e) = V_0 \quad \mathsf{Bnd}(C) = U}{\mathsf{Fv}(\{C\}e) = V \cup (V_0 \backslash U)}$$

$$\frac{\mathsf{Fv}(l) = V_1 \quad \mathsf{Fv}(e) = V_2 \quad \mathsf{Fv}(C) = V_3}{\mathsf{Fv}(l \leftarrow e; C) = V_1 \cup V_2 \cup V_3} \qquad \frac{\mathsf{Fv}(e) = V_1 \quad \mathsf{Fv}(e) = V_2}{\mathsf{Fv}(t := e; C) = V_1 \cup (V_2 \backslash \{t\})}$$

$$\frac{\mathsf{Fv}(e) = V \quad \mathsf{Fv}(C_1) = V_1 \quad \mathsf{Fv}(C_2) = V_2 \quad \mathsf{Fv}(C) = V_3 \quad \mathsf{Bnd}(C_1) = U_1 \quad \mathsf{Bnd}(C_2) = U_2}{\mathsf{Fv}(\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C) = V \cup V_1 \cup V_2 \cup (V_3 \backslash (U_1 \cap U_2)}$$

$$\frac{\mathsf{Fv}(e) = V \quad \mathsf{Fv}(C_0) = V_0 \quad \mathsf{Fv}(C) = V_1}{\mathsf{Fv}(\mathsf{while}\ e\ \{C_0\}; C) = V \cup V_0 \cup V_1} \qquad \frac{}{\mathsf{Fv}(\mathsf{NULL}) = \{\}}$$

Table 3: Calculation for free variables

$$\frac{\mathsf{Bnd}(C) = V}{\mathsf{Bnd}(t := e; C) = \{t\} \cup U} \qquad \frac{\mathsf{Bnd}(C) = V}{\mathsf{Bnd}(l \leftarrow e; C) = U}$$

$$\frac{\mathsf{Bnd}(C_1) = U_1 \quad \mathsf{Bnd}(C_2) = U_2 \quad \mathsf{Bnd}(C) = U}{\mathsf{Bnd}(\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C) = (U_1 \cap U_2) \cup U}$$

$$\frac{\mathsf{Bnd}(C) = U}{\mathsf{Bnd}(\mathsf{while}\ e\ \{C_0\}; C) = U} \qquad \frac{}{\mathsf{Bnd}(\mathsf{NULL}) = \{\}}$$

Table 4: Calculation of bindings

Some comments are worth making: after a conditional, strictly speaking, only the variables which are bound in both branches branch (or before) are bound. Thus, strictly speaking, the variables which are not common down both branches are local. In practise, especially if one has not semantically checked this condition, this will make it seem that some variables which should be bound are not. Also, in practise, one does not want these local bindings to be captured when code is moved so one should rename them. A "while" loop binds does not bind any new variable! This is because it may be skipped altogether! However, the body of a while loop may use some new variables which must be treated, therefore, as local.

2.4 Typed internal representation for CIRL programs

The type information obtained from the semantic checking described above is crucial information for the compilation process. For example the type of a temporary determines to which register can be used to hold it. It is therefore important not to loose this information. Thus, after semantic checking it is useful to change the datatype so that the type information is explicit.

Below is a suggestion for a datatype which serves this purpose. Note that two things have happened: type information occurs everywhere but also the distinction between expressions and locations has been removed (although it is still present in the type information) and this allows operations such as the "offset" to be handled more uniformly.

```
data ICirlprog a = ICirlprog [DataDef] [IFunctionDef a] (IExpression a)
data IFunctionDef a = IFunction String [DataDef] CirlType (ICirlprog a)

data ICommand a
         = IAssign CirlType a (IExpression a)
         | IStore CirlType (IExpression a) (IExpression a)
         | ICond (IExpression a) [ICommand a] [ICommand a]
         | IWhile (IExpression a) [ICommand a]

data IExpression a
         = Imem   CirlType (IExpression a)
         | Iapply  CirlType Ip [IExpression a]
         | Iseq  CirlType [ICommand a] (IExpression a)
         | Itemp  CirlType a
         | Iconst CirlType  BaseValue
         | IFp CirlType Int String
         | IAddr CirlType String

data Ip  = IBinop CirlType Binop
         | IUniop CirlType Uniop
         | ICall Int String
         | IOffset CirlType
```

3 Removing Embedded Commands

The embedded commands have a vital role in the translation from program code. They appear in the intermediate representations of programs for various reasons:

(a) Some special language features require them (as in the examples above).

(b) To translate function calls it is usually necessary to fill certain specific registers (say $s_0, .., s_n$) with arguments. Thus, a function call $f(e_1, ..., e_n)$ must be translated to

$$\{s_0 := f(\{s_1 := e_1\}s_1, ..., (\{s_n := e_n\}s_n)\}s_0$$

which introduces embedded commands.

(c) It is convenient to translate function bodies as embedded commands acting on the returned expression.

(d) When one does a register allocation for expressions the assignments (and spilling code) can be viewed as introducing embedded commands.

Before one can do register allocation on expressions one must remove all the embedded commands and perform instruction selection (so the commands are appropriate for the chosen hardware). The commands of an embedded command, $\{C\}e$, introduces bindings for temporary variables by assigning values to them. The assignment, thus, doubles as the declaration of that variable. The scope of these bindings is limited to the expression e. This means that when one moves the embedded command sequence C one must ensure that the bindings it introduces are not captured accidentally by any command one interposes between C and e. This necessitates renaming the bound variables away from variables which could be used by other terms (this is called "α-renaming").

3.1 Rules for removing embedded commands

The embedded command may be removed using the following rules which depend on non-interference which is discussed further below:

[Rule.1] When C does not interfere with $e_1, .., e_{r-1}$ and the bindings of C are disjoint from the free variables of each e_j with $j \neq r$:

$$f(e_1, ..., \{C\}e_r, ..., e_n) \Rightarrow \{C\}f(e_1, ..., e_r, ..., e_n)$$

[Rule.2] When x is not in the bindings of C and the variables bound in C are disjoint from the current binding context:

$$x := \{C\}e \Rightarrow C; t := e$$

[Rule.3] When the free variables of e are disjoint from the bindings of C and the variables bound in C are disjoint from the current binding context:

$$\{C\}p \leftarrow e \Rightarrow C; p \leftarrow e$$

[Rule.4] When C does not interfere with p, the free variables of p are disjoint from the bindings of C,and the variables bound in C are disjoint from the current binding context:

$$p \leftarrow \{C\}e \Rightarrow C; p \leftarrow e$$

[Rule.5] We can always amalgamate sequential embedded commands:

$$\{C_1\}\{C_2\}e \Rightarrow \{C_1; C_2\}e$$

[Rule.6] When the bindings of C are disjoint from the variables free in C_1 and C_2 and the current binding context:

$$\mathsf{if}\ \{C\}e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \Rightarrow C; \mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$$

[Rule.7] When the bound variable of C are disjoint from the free variable in C' and the current binding context:

$$\mathsf{while}\ \{C\}e\ \mathsf{do}\ C' \Rightarrow C; \mathsf{while}\ e\ \mathsf{do}\ (C'; C).$$

Almost all of these calculations depend on knowing the current binding context. This may be calculated either as the binding context (all variables which are bound prior to that point) or, more ecomonically, as the free variables required subsequent to that point. The advantage potential advantage of using the larger of these two context is that one can avoid repeated renamings as the commands sequences move forward over expressions in which new variables are free.

A further important aspect of this process arises when the order of commands and expressions changes during the removal of embedded commands. If this happens they will be evaluated in a different order and a different behavior *may* result unless they can be guaranteed not to "interfere". This notion of interfering is decribed next.

3.2 Interference

Interference arises in two ways:

(a) The embedded command assigns values to temporaries which are used in an expression in front of which it is being moved. The remedy in this case is to rename (a process called α-renaming) the temporaries involved.

(b) The embedded command and the expression past which it is being moved access memory with one or both writing to memory. The remedy here is to make the expression safe by creating an embedded command which assigns the expression to a new temporary so that the order of evaluation is not changed.

In the former case we may rename the temporary variable in the embedded command to remove the source of the problem (temporary capture). When there is the possibility of interference, as expressions possibly access memory and store value in memory in the second manner it is necessary to keep the computations in the same order: to achieve this it is necessary to introduce new intermediate temporaries to hold the results of these computations. Thus, removing embedded commands generally create new temporaries, especially when one is forced to avoid interference of the latter kind. Recall that the more temporaries we create the more likely we may have to spill. Thus, one reason why we may want to be quite discerning about what interferes and what does not is to relieve this source of register pressure.

We may organize the calculation for non-interference as follows a CIRL expression or command either is:

CLEAN: That is has no loads, store or *any* access to memory.

LDING: That is does (possibly) load something from memory, that is (possibly) has a $\mathsf{mem}(e)$ subexpression somewhere, but definitely has no store instructions.

STING: That is (possibly) stores stuff to memory that is it contains a function call (we assume these are bad news) or a store instruction $e_1 \leftarrow e_2$.

We now have an "interference table":

	CLEAN	LDING	STING
CLEAN	$\surd$	$\surd$	$\surd$
LDING	$\surd$	$\surd$	$\times$
STING	$\surd$	$\times$	$\times$

where there is guaranteed no interference when there is a $\surd$ *and* the two CIRL expressions have no temporaries in common. There still actually may be no interference when there is a $\times$ but our objective is to be conservative (one can be less conservative by tracking what location ranges can be accessed).

3.3 Organizing the computation

A convenient way to organize the embedded command removal (but certainly not the only way) is to remove commands in a "by-value" manner (functionally this is a fold over the datatype – with a state) by first removing the embedded command from subexpressions and working the expression from the leaves toward its root. Each expression is replaced by a command sequence and a new expression $e' \mapsto (C, e)$. The interference calculation will sometimes force the current expression to be assigned in the command sequence and to be replaced by a temporary. By arranging that any embedded command which is introduced in this process *never* interferes one can simply collect commands from the leaves toward the root. The only really tricky aspect is that an already existing embedded command may introduce a source of temporary capture! This has to be accommodated by special code to make the predefined embedded command "safe".

3.4 Making embedded commands safe

As one is processing an expression one must build up a list of bound variables, the **context**, which arise from the bindings of embedded commands. If one is to move a command sequence the variables it binds must not conflict with the variables in this context. Thus, given a context Γ of variables and an embedded command $\{C\}e$ one must go through the variables that the command list C binds and, when they clash with the variables in the context Γ, change their names, by substitution, from where they are assigned. This then produces a substitution list which must then be applied to the term e. There are some important variations on this general pattern to accommodate "while" loops and the conditionals which we discuss further below.

The process for making an embedded command safe given a set V of variables to avoid (this will include the binding context) is, more precisely, as follows:

- To make $\{x := e; C\}e'$ safe given the variables in V to avoid there are two cases:
  - (a) $x \in V$ then choose a fresh variable y, substitute y for x everywhere in C and e' and continue to make $\{C\}e'$ safe adding y to the variables V to be avoided.
  - (b) $x \notin V$ then one makes no change to the assignment but continue to make $\{C\}e'$ safe avoiding V.
- A store does not bind any variables so one can pass over these commands making the remainder of the commands safe.
- When the conditional is the first command one must make both its command lists safe. However, one must be careful to keep its joint bindings the same! To do this one alters the bindings of the one condition while collecting substitutions. Then one modifies the second list using the the substitutions already established from the first condition to rename the bindings which are in common. Note this is very different from a blanket substitution because one must only use the already established substitutions subsequent to an assignment on the variable.
- When a "while" loop is the first command one must first rewrite its body to ensure its bindings avoid the variables. This gives a list of substitutions. One must now use this list to substitute the body (as the code will cycle through the body) noting which substitutions are used. The substitutions which are used must be collected into a list of assignments which become a preamble to (i.e. are placed immediately in front of) the while loop.

3.5 Examples

1. Here is an example which illustrates how the embedded commands on arguments of a function can be removed. It is, as has been mentioned, possibly a little artificial as often one simply wishes to always call functions on a fixed (precolored) sequence of registers. However, it does illustrate the general manner in which arguments must be treated.

```
var a,b,c: bool
fun f(var x,y,z:bool):int
  [ .... ]

fun g(var x,y,z:bool):bool
  [ .... ]

{ a := mem(addr(a));
  b := mem(addr(b));
  c := mem(addr(c));
  t := f( {t := g(a,b,c);}t&&a
        , {a := g(b,c,a);} a||g(c,b,a)
        , {t := c&&mem(addr(b));} not t
        );
} t+t
```

Note that the middle argument of the call to `f` has a function call (which is assumed to be storing. The last argument has a memory reference and this means that the middle argument must be replaced by a variable. The body then gets translated as:

```
{ t0:bool := mem(addr(a))
; t1:bool := mem(addr(b))
; t2:bool := mem(addr(c))
; t4:bool := g(t0, t1, t2)
; t5:bool := g(t1, t2, t0)
; t7:bool := (t5||g(t2, t1, t5))
; t6:bool := (t2&&mem(addr(b)))
; t3:int := f((t4&&t0), t7, not t6)
; } (t3+t3)
```

2. To illustrate the removal of embedded commands from a store command consider:

```
var b[40],c,d :int

{ { x:= mem(addr(c))+3
  ; x:= x*x
  ; } mem(addr(b)\x)
     <- { y:= mem(mem(addr(b))\2)+mem(addr(d))
         ; x:= 2* y
         ; mem(addr(b))\2 <- x
         ; } x
; x := 9
; } x
```

Here the embedded command for the expression `x` contains a store command necessitating the assigning of the location calculation to a temporary. If this store is removed from the embedded command this transfer should not happen.

```
{ t0:int := (mem(addr(c))+3)
; t0:int := (t0*t0)
; t3:int := mem((addr(b)\t0))
; t1:int := (mem((mem(addr(b))\2))+mem(addr(d)))
; t2:int := (2*t1)
; (mem(addr(b))\2):int <- t2
; t3:int <- t2
; t0:int := 9
; } t0
```

3. Here is and example which removes embedded commands from stores and offsets:

```
var p[12] :int
fun f(var a, b:int):int [ ... ]

{ mem(addr(p))\7 <-
    {x:= f( mem(mem(addr(p))\1)
          , mem(mem(addr(p))\ ({x := f(1,1)
                                ; }x))
          )
    ; } mem(mem(addr(p))\3)*7
; y := 1
; }y
```

Note the function call in the embedded command for the second argument of the offset. This causes all the arguments at both levels to have to be assigned to temporaries. Note that the variable numbering gives away the order in which my algorithm discovers the arguments which have to be displaced!

```
{ t4:int := (mem(addr(p))\7)
; t3:int := mem((mem(addr(p))\1))
; t2:ref(int) := mem(addr(p))
; t1:int := f(1,1)
; t0:int := f(t3, mem((t2\t1)))
; t4:int <- (mem((mem(addr(p))\3))*7)
; t1:int := 1
; } t1
```

4. Here is an example of a conditional in an embedded command:

```
var a,b[20] :int

{ z := mem(mem(addr(b))\10)
; mem(addr(b))\0 <- { if {z := ref(mem(addr(b))\z)
                          ; } mem(z) <12
                      then {z := { x := mem(addr(a))
                                 ; x := x*x
                                 ; } x
                           ; }
                      else {z := { addr(a) <- mem(addr(a))+1
                                 ; } 10
                           ; }
                    ; } z
; z := mem(addr(a))
; } z
```

Notice that `t3`, originally `z` is renamed the same way down each branch:

```
{ t0:int := mem((mem(addr(b))\10))
; t4:int := (mem(addr(b))\0)
; t1:ref(int) := (mem(addr(b))\2)
; if (mem(t1)<12)
   then { t2:int := mem(addr(a))
        ; t2:int := (t2*t2)
        ; t3:int := t2
        ; }
   else { addr(a):int <- (mem(addr(a))+1)
        ; t3:int := 10
        ; }
; t4:int <- t3
; t0:int := mem(addr(a))
; } t0
```

5. This example illustrates the removing embedded commands from the test of a while loop. Here we repeat the code in the loop. There are two ways of avoiding this (a) by not removing the embedded command from the condition of the "while" loop and translating this form (b) by changing the looping construct to a more fundamental one (which we shall do shortly).

   ```
   var a[2]: int

   { x := mem(mem(addr(a))\0)
   ; y := mem(mem(addr(a))\1)
   ; while { b:= x < y; } b
      { x := x +1
      ; y := y-1
      ; }
   ; x := x+y
   ; } x
   ```

   Here is the result:

   ```
   { t0:int := mem((mem(addr(a))\0))
   ; t1:int := mem((mem(addr(a))\1))
   ;  t2:bool := (t0<t1)
   ;  while t2
      { t0:int := (t0+1)
      ; t1:int := (t1-1)
      ; t2:bool := (t0<t1)
      ; }
   ; t0:int := (t0+t1)
   ; } t0
   ```

6. To illustrate the problems with making while loop code in an embedded command safe consider the following example:

```
var a[3] : int

function g(var z:int):int
[ mem(fp(z)) ]

{ t:= mem(mem(addr(a))\0)
; x:= mem(mem(addr(a))\1)
; y:= mem(mem(addr(a))\2)
; t := { while x=<y { t := t+1
                    ; x:= x+1
                    ; y:= y-1
                    ; }
       ; } g(t)
; } t
```

Here we must introduce a preamble which renames the variables to the intended output:

```
{ t0:int := mem((mem(addr(a))\0))
; t1:int := mem((mem(addr(a))\1))
; t2:int := mem((mem(addr(a))\2))
; t3:int := t0
; t4:int := t1
; t5:int := t2
; while (t4=<t5)
    { t3:int := (t3+1)
    ; t4:int := (t4+1)
    ; t5:int := (t5-1)
    ; }
; t0:int := g(t3)
; } t0
```

3.6 Function calls and more on interference

In the above we have left the function calls in the expressions. Function calls, however, in the compiling process have a special status. Not only do we always assume the worst in regards interference but they play no role in the next step of the compiling process which is the instruction selection. It is therefore sensible to break down the function calls in the code at the outset and this has a further important benefit. When embbed commands have been removed from expressions the only source of a storage is from a function call. If we remove function calls as well expressions can no longer contain storage commands at all! This means that expressions can no longer interfere. This is important as we are then free to apply the Sethi-Ullman algorithm to break down expressions into a series of assignments in which we can freely swap the order of evaluation.

This then raises the issue of how to break up the function calls. The trick is to encapsulate them in an embedded command in a preprocesing step *before* applying the general algorithm for removing the embedded commands. Thus an expression `f(exp1,...,expn)` can be rewritten as:

```
    { s1 := exp1;...;sn := expn;s_0=f(s1,...,sn) } s0
```

Note this form is equivalent to that described earlier, athough we have to assume that `si` is not free in any `expj` with *tti* < `j`. This means that when the embedded commands are removed the function calls will all be at the top level. This is desirable as function calls will have some special translation requirements specific to the architecture.

Here is the first example above with the function calls broken up in this manner:

```
{ t0:bool := mem(addr(a))
; t1:bool := mem(addr(b))
; t2:bool := mem(addr(c))
; t3:bool := g(t0, t1, t2)
; t3:bool := (t3&&t0)
; t5:bool := g(t1, t2, t0)
; t4:bool := g(t2, t1, t5)
; t4:bool := (t5||t4)
; t6:bool := (t2&&mem(addr(b)))
; t5:bool := not t6
; t3:int := f(t3, t4, t5)
; } (t3+t3)
```

4 Instruction Selection

When all embedded commands have been removed from a command sequence the code becomes a series of assignments, stores, conditionals, and while loops. Using instruction selection techniques the aim is then to translate this into the operations provided by the actual machine. We shall do this using a dynamic algorithm which (for expression trees) provides an optimal solution. This is why, in the process of removing embedded commands, we wished to keep the tree structure as intact as possible.

We shall start by describing an instruction set, the Jouette instructions[1], which are loosely based on Motorola instruction sets.

4.1 Jouette instructions

A general jouette instruction consists of an operation code followed by one or more operands. They can be broken in to two categories, *expression* instructions and command instructions. Expression instructions produce a value and place it into either a data or address register.

A typical expression instruction is written as:

$$\mathsf{opn}\ \ [i_0], i_1, \ldots, i_n$$

where the register in square parentheses, $[i_0]$, indicate the output and the list $i_1 \ldots, i_n$, the inputs. This is important information to the instruction selection stage as such an instruction will be viewed as an operation on its inputs combinaed with an assignment. An instruction which has no output arguments is a command and is a side-effecting operation, either changing main memory or program flow. It is typically written as:

$$\mathsf{opn}\ \ L, i_1, \ldots, i_n$$

or

$$\mathsf{opn}\ \ i_1, \ldots, i_n$$

where L is a label and the $i_1, \ldots, i_n$ are input to the command.

Registers in the Jouette machine are grouped into data registers and address registers.

Data registers	$d_0, d_1, ..., d_M$
Address registers	$a_0, a_1, ..., a_N$

One can only access main memory using an address register. The instructions are displayed in table 5.

Table 5: Jouette Instruction list

Instruction	Cst	Effect/CIRL	Description
`add` $[d_i], d_j, d_k$	2	$d_i := d_j + d_k$	Expression: Adds the contents of two data registers and places it in a third.
`mul` $[d_i], d_j, d_k$	5	$d_i := d_j * d_k$	Expression: Multiplies the contents of two data registers and places it in a third.

Continued on next page

[1]These instructions were collected by Brett Giles

Instruction	**Cst**	**Effect/CIRL**	**Description**
`sub` $[d_i], d_j, d_k$	2	$d_i := d_j - d_k$	Expression: Subtracts the contents of two data registers and places it in a third.
`div` $[d_i], d_j, d_k$	10	$d_i := d_j / d_k$	Expression: Divides the contents of two data registers and places it in a third.
`addi` $[d_i], d_j, c$	1	$d_i := d_j + c$	Expression: Adds a constant to a data register and places it in another.
`subi` $[d_i], d_j, c$	1	$d_i := d_j - c$	Expression: Subtracts a constant from a data register and places it in another.
`movea` $[d_i], a_j$	2	$d_i := ref(a_j)$	Expression: Moves contents of an address register to a data register.
`moved` $[a_i], d_j$	2	$a_i := deref(d_j)$	Expression: Moves contents of a data register to an address register.
`load` $[d_i], a_j, c$	10	$d_i := mem[a_j \backslash c]$	Expression: Moves memory pointed to by an address register + constant to a data register.
`store` a_i, c, d_j	10	$a_i \backslash c \leftarrow d_j$	Command: Store data register to memory pointed to by an address register + constant.
`movem` a_i, a_j	20	$a_i \leftarrow mem[a_j]$	Command: Moves memory pointed to by one address register to memory pointed to by another address register.
`bge` L, d_j	3	JUMP to L if $d_j \geq 0$	Command: Jump to label (change program counter) if data register greater than or equal to 0.
`blt` L, d_j	3	JUMP to L if $d_j < 0$	Command:
`bz` L, d_j	3	JUMP to L if $d_j = 0$	Command:

Continued on next page

Instruction	Cst	Effect/CIRL	Description
`bnz` L, d_j	3	JUMP to L if $d_j \neq 0$	Command:
`br` L	2	JUMP to L	Command: Jump to label.
`call` $[d_0], L, d_1, \ldots, d_n$	5	JUMP to L, remember return location	Expression: Jump to label and remember address of next instruction. d_0 is the output register and $d_1, \ldots, d_n$ are the input arguments.
`return` d	3	Return from last `call`	Return to instruction just after last call.
L:	0	Label next instruction	Provide destination for jump, branch and call.

4.2 Translation

A **translation system** for terms consists of

- Two sets of types the **source types**, S, and the **target types**, T, together with a map $t : S \longrightarrow T$ which turns source types into target types;
- Three algebraic signatures:

$$\begin{aligned} \mathsf{type} &: \mathsf{Target} \longrightarrow \mathsf{list}(T) \times T \\ \mathsf{type} &: \mathsf{Internal} \longrightarrow \mathsf{list}(T) \times T \\ \mathsf{type} &: \mathsf{Source} \longrightarrow \mathsf{list}(S) \times S \end{aligned}$$

  respectively the **target**, **internal**, and **source** signatures.

- A set of translation rules either of the form:
  - A translation to an internal pattern:

$$c(x_1, .., x_n) \Leftarrow t_1 \mid ...| t_n$$

    where c is a function in the internal signature signature and $c(x_1, ..., x_n)$ and $t_1, ..., t_n$ are well-type terms in the signature $\mathsf{Internal} + \mathsf{Source}$ using all the the variables $\{x_1 : X_1, ..., x_n : X_n\}$. The typing is with respect to the target types with source types being coerced by t into target types.
  - A translation to a target term:

$$t_0 \Leftarrow t_1 \mid ...| t_n$$

    where t_0 is a well typed term in the target signature, $t_1, .., t_n$ are well-typed terms in the signature $\mathsf{Internal} + \mathsf{Source}$, as above, such that each t_i uses all the variables used in t_0.

Example 4.1 For two type systems, $S = T = \{E, \mathbb{Z}\}$ where $\mathbb{Z}$ is the integers, let the target signature be $\mathsf{ADDI} : E, \mathbb{Z} \longrightarrow E$ and $0 :\longrightarrow E$, the internal signature be empty, and the source signature be $\mathsf{add} : E, E \longrightarrow E$ and $\mathsf{const} : \mathbb{Z} \longrightarrow E$.

$$\begin{array}{rcl} \mathsf{ADDI}(x,n) & \Leftarrow & \mathsf{add}(x, \mathsf{const}(n)) \mid \mathsf{add}(\mathsf{const}(n)), x) \\ x & \Leftarrow & \mathsf{add}(x, \mathsf{const}(0)) \mid \mathsf{add}(\mathsf{const}(0), x) \end{array}$$

A first question to ask is whether every source tree can be translated into a target tree. In the above example it is clear that we cannot translate arbitrary source trees into target expressions (e.g try translating $\mathsf{add}(x, y)$). If a tree can be translated, the next problem is whether one translation is better than another. Indeed, more importantly, is there an optimal, translation into a target tree.

The form of the rules guarantees it is decidable whether a translation of a source term is possible: each rule *cannot* increase the size of the non-target portion of the tree. It may be possible to enter a finite cycle, but one can easily detect this. A simple approach to determining whether a tree is translatable is to "munch it" non-deterministically from the root (watching for repetition). This means eating a right hand side of a rule and, when a terminal constructor appears at the root, working recursively on its arguments.

It is quite possible that every source tree is a translation of a starting tree. In particular, this means that the function symbols of the source signature must be the translation of something in the start signature. This is necessary and also sufficient; once one can translate the basic functions one can translate all the terms. In instruction selection problems it is usual that the intermediate language can always be translated! In this case the issue is to find the best translation.

A bad translation of $\mathsf{add}(\mathsf{const}(m), \mathsf{const}(n))$ in the above example is as

$$\mathsf{ADDI}(\mathsf{ADDI}(\mathsf{ADDI}(0, m), 0), n)$$

a better translation would be $\mathsf{ADDI}(\mathsf{ADDI}(0, m), n)$. Here we have not defined what we mean by "better": it may be given by the size, the register requirement, or, indeed, some other (possibly rather complex) requirement.

We shall focus on **monotone cost criteria** which satisfy if $\mathsf{size}(s_i) \leq \mathsf{size}(s'_i)$ then we shall require that $\mathsf{size}(c(s_1, ..., s_n)) \leq \mathsf{size}(c(s'_1, ..., s'_n))$. Usually such a cost criteria is given by associating costs with the operations (but register need for example is such a criterion). This allows us to optimize the representation of the subtrees before determining the representation of the root and this allows a dynamic algorithm to solve the problem as we can "remember" the best solution for all subtrees.

4.3 Types for instruction selection

Our aim is now to develop here a general instruction selection "module" which allows translation from CIRL into various instruction sets – including, in particular, the Jouette machine outlined above. The instruction selection phase will use a "rules" file which specifies the translation from CIRL instructions into real code. Each translation rule will have an associated cost which will then be used in the dynamic algorithm.

However, before we can fully understand the rules file (see table 8) it is necessary to understand the general form of the instructions we wish to produce and the various datatypes associated with the matching process.

4.3.1 Instruction datatype

The datatype for instructions is supposed to embody a rather general form of machine instruction which may be applicable to various architectures. Ignoring, for the moment, the structure of programs and function definitions it should be clear that expressions (outside the adressing modes) are as expected (an operation applied to arguments). Embedded commands are still present and so may have to be removed (again) after translation. The command structure deserves for comment:

```
 data InstrProg = InstrProg [DataDef] [InstrFDef] InstrExp

data InstrFDef = InstrFDef String [DataDef] CirlType InstrProg

data InstrExp   = InstrOp Instrop [InstrExp]
                | InstrK BaseValue
                | InstrAd String
                | InstrFp Int String
                | InstrReg MachineType InstrR
                | InstrSeq [InstrCom] InstrExp

data InstrCom   = InstrCom String [InstrExp]
                | InstrLabel Int
                | InstrJump String [InstrExp] Int
                | InstrAssign MachineType InstrR InstrExp

data InstrR = InstrRTemp Int
            | InstrRFix String

data Instrop = InstrCall Int String
             | Instrop String

type MachineType = String
```

The translation of the Jouette command **ADDI** $[d3]$ $e1$ k into this structure, becomes the combination of an assignment and an expression:

$$\texttt{InstrAssign Data 3 (ADDI [e1, InstrKk])}.$$

An architecture may support various jump instructions: for example the Jouette **bge** L, d_j is translated as:

$$\texttt{InstrJump BGE [dj] L}$$

where a new instruction which consists of a label is introduced:

$$\texttt{InstrLabel L}.$$

The store command, $e_1 \longleftarrow e_2$ gets translated as an "ordinary" command:

$$\mathtt{InstrCom\ STORE}\ [\mathtt{e1}, \mathtt{e2}]$$

4.3.2 Datatypes for matching

In a rules file the right hand side of a rule, also called the pattern, is matched against a (typed) CIRL expression or command sequence. These right hand sides are stored into a special datatype which facilitates matching and parallels the structure of CIRL expressions and commands. The matching structure for CIRL expressions is:

```
data MExpression
       = MMem MExpression
       | MApply Mop [MExpression]
       | MConst MBaseValue
       | MEVar MachineType Int

data MBaseValue
       = MVal BaseValue
       | MBVar MachineType Int

data Mop = MBinop Binop
         | MUniop  Uniop
         | MCall Int String
         | MOffset
```

Note that one cannot match against an embedded command. Also note that it is always possible to have a variable (represented by the last constructor in the datatype here) in a match expression. Constants must be matched (and can be variables. Note also that there is type information carried by these patterns but it is the machine type. As one matches one must make sure the CIRL type translates to the specified machine type.

The matching structure for command sequences is mostly carried by the matching structure for sequences (lists) which allow for prefix matching (the second constructor is always substituted by the rest of the list):

```
data MList a = MCons a (MList a)
             | MLVar Int

data MCommand
       = MAssign  MachineType Int MExpression
       | MStore MExpression MExpression
       | MCond (MExpression) (MList MCommand) (MList MCommand)
       | MWhile MExpression (MList MCommand)
```

4.4 The datatype for translations

The lefthand sides of rules are translated into slightly different datatypes which match closely the the structure of instructions. These are the target instruction expressions:

```
data MInstrExp  = MInstrOp  MachineType String [MInstrExp]
                | MInstrK MachineType BaseValue
                | MInstrKV MachineType Int
                | MInstrReg MachineType MReg
                | MInstrExpVar MachineType Int
                | MInstrSeq (MIList MInstrCom) MInstrExp

data MReg = MRegVar Int -- register variable
          | MRegTemp Int    -- temporary
          | MRegFix String -- fixed/precolored
```

Note that registers have no less than three forms: variable being matched and transfered from the right hand side, temporaries special to the target code, and precolored registers. These rules are typed although only to indicate the sort of register required in the machine.

Target commands again rely heavily on the structure of the list translations:

```
data MIList a = MICons a (MIList a)
              | MIAppend Int (MIList a)
              | MILVar Int
              | MINil

data MInstrCom  = MInstrCom String [MInstrExp]
                | MInstrLabel Int
                | MInstrJump String [MInstrExp] Int
                | MInstrAssign MReg MInstrExp
```

Note that the translation of a list has two extra possibilities: `MIAppend c rest` in which the command sequences substituting `c` is appended to the front of the command seqence `rest`, and `MILVar c` in which the command sequence c is appended at the end. When translating the command rules the outermost target sequence is regarded as a prefix (so uses this constructor). On the otherhand embedded commands in target expressions (which are only allowed in the expression rules) are alway terminated by `MINil`.

4.5 Datatypes for rules

Each rule is held together with its cost:

```
type ISExpRule = (MExpression,Int,MInstrExp)
type ISComRule = (MList MCommand,Int,MIList MInstrCom)
type ISRules = ([ISExpRule],[ISComRule])
```

Note that each rule contains five(!) different types of variable: for expressions, for command sequences, for temporaries, for constants, and for labels. On matching the first four are instantiated by substitution with the matched components. The labels names, by contrast are generated in the translation process.

4.6 Dynamic types

The last data types used in the instruction selection phase are the so called “dynamic types”. They basically mimic the CIRL datatypes but also at each stage also hold the best translation so far. Notice for commands that, rather than holding the best translation of a single command, the best translation of a sequence of commands is held.

The dynamic structure for expressions is:

```
data DExpression dexp dcoms
       = DMem  dexp CirlType (DExpression dexp dcoms)
       | DApply dexp CirlType Ip [DExpression dexp dcoms]
       | DConst dexp CirlType  BaseValue
       | DSeq dexp CirlType (DList dcoms (DCommand dexp dcoms))
                            (DExpression dexp dcoms)
       | DTemp dexp CirlType Int
       | DFp dexp CirlType Int String
       | DAddr dexp CirlType String
```

and the dynamic structure for commands is:

```
data DList d a = DNil d
               | DCons d a (DList d a)

data DCommand dexp dcoms
       = DAssign CirlType Int (DExpression dexp dcoms)
       | DStore CirlType (DExpression dexp dcoms) (DExpression dexp dcoms)
       | DCond (DExpression  dexp dcoms)
                    (DList dcoms (DCommand  dexp dcoms))
                    (DList dcoms (DCommand  dexp dcoms))
       | DWhile (DExpression  dexp dcoms)
                     (DList dcoms (DCommand  dexp dcoms))
```

The dynamic strucure for programs and functions has the optimal code translation selected for the functions and the program:

```
data DProg
   = Dprog [DataDef] [DFunt] (SF(Int,[InstrCom]))

data DFunt
   = DFunt String [DataDef] CirlType DProg
```

These are used with the following instantiation of types in which the generated code is wrapped in the state monad which is dependent on the label number:

```
type DExp = DExpression (SF(Int,SM Int InstrExp)) (SF(Int,SM Int [InstrCom]))
type DCom = DCommand (SF(Int,SM Int InstrExp)) (SF(Int,SM Int [InstrCom]))
type DComs = DList (SF(Int,SM Int [InstrCom])) DCom
```

Where `SF` is the “success or fail” datatype (exception monad):

```
data SF a = SS a | FF
```

Note that it is quite possible to fail to find a translation! SM is the state monad. Command and expression code depends on the label numbers chosen, thus, these must be threaded round the code when it is generated.

4.7 The syntax of the rules file

An example of a rules file is in table 8. The grammar for the rules is in table 6 and the lexemes for this grammar are in table 7. There follows a brief description of the grammar of a rules file.

A rules file is split into two parts: the expression rules and the command rules

```
rules -> "Expressions" exp_rules "Commands" com_rules .
```

Each rules is preceded by a cost (a natural number). An expression rule has a single lefthand side target machine instruction expression (which can have an embedded command – which is viewed as a terminated list pattern (rather than a prefix) – and possibly multiple right hand side pattern expressions which translate into the given left hand side.

```
exp_rules -> exp_rule exp_rules | .
exp_rule -> NUM ":" minstr_exp "<==" pattern_exps .
minstr_exp -> "{" instr_coms "}" pure_instr_exp | pure_instr_exp .
pattern_exps -> pattern_exp more_pattern_exp .
```

A pattern expression is a CIRL operation applied to arguments, and expression variable, a constant variable, or a constant.

```
pattern_exp ->
       ADD pattern_exp_args | SUB pattern_exp_args | DIV pattern_exp_args
     | MOD pattern_exp_args | EQUAL pattern_exp_args | GT pattern_exp_args
     | AND pattern_exp_args | OR pattern_exp_args | XOR pattern_exp_args
     | NEQ pattern_exp_args | GE pattern_exp_args | LE pattern_exp_args
     | LT pattern_exp_args | NOT "(" pattern_exp ")" | NEG "(" pattern_exp ")"
     | MEM "(" pattern_exp ")"  | EXPV  ":" machine_type
     | CONSTV ":" machine_type | NUM | REAL | BOOL
pattern_exp_args -> "(" pattern_exp "," pattern_exp ")".
```

The target instruction expression can have embedded commands: a pure target instuction expression could be a (typed) expression variable, a register (variable, fixed/precolored, or temporary) a constant variable, a constant, or a general instruction expression.

```
pure_instr_exp -> EXPV ":" machine_type | register_instr
                | CONSTV ":" machine_type
                | NUM | BOOL | REAL
                | ID "(" pure_instr_exps ")" ":" machine_type
register_instr -> REGV | REGT | REGF
pure_instr_exps -> pure_intr_exp more_pure_instr_exp | .
more_pure_instr_exp -> "," pure_instr_exp more_pure_instr_exp | .
```

The command rules each have an associated cost and consist of a left hand side, the target command sequence, and a right hand side which could be multiple command sequence patterns.

```
com_rules -> com_rule com_rules | .
com_rule -> NUM ":" instr_coms "<==" instr_patterns .
instr_patterns -> coms_pattern more_instr_patterns
more_instr_patterns -> "|" coms_pattern more_instr_patterns | .
```

A command sequence pattern consists a series of command patterns. A command pattern can be an assignment, a store instruction, a conditional or a while command.

```
coms_pattern -> com_pattern more_com_pattern
more_com_pattern -> ";"  com_pattern more_com_pattern | .
com_pattern -> REGV ":" cirl_type ":=" pattern_exp
            | pattern_exp "<-" pattern_exp
            | "if" pattern_exp "then" COMSV "else" COMSV
            | "while" pattern_exp "do" COMSV
```

The target instruction command sequence are a sequence of target instruction commands. These are either jumps, labels, assignmments, or general commands (which will cover various storing modes for example).

```
instr_coms -> COMSV more_instr_coms | instr_com more_instr_coms |.
more_instr_coms -> ";" instr_coms | .
instr_com -> JUMP ID "(" pure_instr_exps ")"  LABV
           | ASSIGN register_instr ASS pure_instr_exp
           | LABEL LABV
           | ID "(" pure_instr_exps ")" .
machine_type -> ID
```

There are no less than five different sorts of variable: expression variable `$n`, command sequence variables `!n`, temporary variables `#n`, constant variables `^ n`, and lastly label variables `n`. These variables are matched into the CIRL expression and so must be instantiated in the pattern matching stage. One can also use "precolored" temporaries, such as `<d0>` and in the embedded commands associated with expressions general temporaries `Tn`. Note in all the above `n` stands for a positive number. When a match is found for a right hand side all the variables, except for the label variables, must be instantiated: thus, every such variable on the left must already occur on the right (and in fact linearly – that is at most once).

4.8 Examples of instruction selection

What has happened to the original code?

(a) The CIRL program has been type checked.

(b) The function calls have been broken up.

(c) Instruction selection has been undertaken using the rules file below (a pseudo-Jouette).

```
rules -> "Expressions" exp_rules "Commands" com_rules .
exp_rules -> exp_rule exp_rules | .
exp_rule -> NUM ":" minstr_exp "<==" pattern_exps .
minstr_exp -> "{" instr_coms0 "}" pure_instr_exp | pure_instr_exp .
instr_coms0 -> COMSV more_instr_coms0 | instr_com more_instr_coms0 |.
more_instr_coms0 -> ";" instr_coms0 | .
pattern_exps -> pattern_exp more_pattern_exp .
more_pattern_exp -> "|" pattern_exp more_pattern_exp | .
pattern_exp ->
           ADD pattern_exp_args | SUB pattern_exp_args | DIV pattern_exp_args
         | MOD pattern_exp_args | EQUAL pattern_exp_args | GT pattern_exp_args
         | AND pattern_exp_args | OR pattern_exp_args | XOR pattern_exp_args
         | NEQ pattern_exp_args | GE pattern_exp_args | LE pattern_exp_args
         | LT pattern_exp_args | NOT "(" pattern_exp ")"
         | NEG "(" pattern_exp ")" | MEM "(" pattern_exp ")"
         | EXPV  ":" machine_type | CONSTV ":" machine_type
         | NUM | REAL | BOOL
pattern_exp_args -> "(" pattern_exp "," pattern_exp ")".
pure_instr_exp -> EXPV ":" machine_type | register_instr
               | CONSTV ":" machine_type
               | NUM | BOOL | REAL
               | ID "(" pure_instr_exps ")" ":" machine_type
register_instr -> REGV | REGT | REGF
pure_instr_exps -> pure_intr_exp more_pure_instr_exp | .
more_pure_instr_exp -> "," pure_instr_exp more_pure_instr_exp | .
com_rules -> com_rule com_rules | .
com_rule -> NUM ":" instr_coms "<==" instr_patterns .
instr_patterns -> instr_pattern more_instr_patterns
more_instr_patterns -> "|" instr_pattern more_instr_patterns | .
coms_pattern -> com_pattern more_com_pattern
more_com_pattern -> ";"  com_pattern more_com_pattern | .
com_pattern -> REGV ":" cirl_type ":=" pattern_exp
            | pattern_exp "<-" pattern_exp
            | "if" pattern_exp "then" COMSV "else" COMSV
            | "while" pattern_exp "do" COMSV
instr_coms -> COMSV more_instr_coms | instr_com more_instr_coms |.
more_instr_coms -> ";" instr_coms | .
instr_com -> JUMP ID "(" pure_instr_exps ")"  LABV
          | ASSIGN register_intr ASS pure_instr_exp
          | LABEL LABV
          | ID "(" pure_instr_exps ")" .
machine_type -> ID
```

Table 6: Grammar for the rules file

```
IF  <== "if"            THEN <== "then"       ELSE <== "else"
WHILE <== "while"       COMMA <== ","         COLON <== ":"
SEMICOLON <== ";"       ASS <== ":="          ASSIGN <== "ASSIGN"
BARROW <== "<=="        ADD <== "add"         SUB <== "sub"
NEG <== "~"             MUL <== "mul"         DIV <== "div"
AND <== "and"           OR <== "or"           XOR <== "xor"
MOD <== "mod"           NOT <== "not"         EQUAL <= "eq"
GTQ <= "gt"             GEQ <= "ge"           LTQ <== "lt"
LEQ <== "le"            NEQ <== "neq"         LARR <== "<-"
OFFSET <== "offset"     MEM <== "mem"         LPAR <== "("
RPAR <== ")"            LCPAR <== "{"         RCPAR <== "}"
SLASH <== "|"           LABEL <== "LABEL"     JUMP <== "JUMP"

EXPRESSIONS <== "Expressions"          COMMANDS <== "Commands"
NUM Int <== <natural number>           EXPV Int <== "$"<natural number>
COMSV Int <== "!"<natural number>      LABV Int  <== "@"<natural number>
CONSTV Int  <== "^"<natural number>    REGV Int  <== "#"<natural number>
ID String <== <Identifier>             REGF String  <== "<"<identifier>">"
REGT Int  <== "T"<natural number>
```

Table 7: Lexemes for rules grammar

(d) The embedded commands on the instructions have been removed.

(e) The result has been “pretty printed” to make it look like Jouette code!

What remains is to:

A. Put the code in static single assignment form – so no registers which do not have to be colored the same are actually colored the same. This allow the register allocation more freedom to find an acceptable colouring. This step is “optional” as register allocation can be performed on the code as is. This should be performed after (or at the same time as removing embedded commands from CIRL.

B. The expressions have yet to be proken up: here the the Sethi-Ullman algorithm should be used to ensure that a minimal number of registers is used to achieve this. This should be performed before removing embeedded commands from the instructions as it is most effecient to use the algorithm to produce embedded commands.

C. A general register allocation phase has yet to be done. Note that this phase may get rid of some of the numerous `MOVE` instructions which get introduced by these transformations. This should be performed after the all the steps above.

Here are the translations of (the first three) programs from the last section:

1. Here is the form with the functions removed:

```
Expressions

2: ADDI(<d0>:Data,^1:Data):Data  <== ^1:Data
2: ADDI($2:Data,^1:Data):Data  <== add ($2:Data,^1:Data)
                                 | add(^1:Data,$2:Data)
2: ADD($1:Data,$2:Data)   <== add ($1:Data,$2:Data)
2: OFFSET ($1:Addr,$2:Data):Addr <== offset ($1:Addr,$2:Data)
2: {ASSIGN T1:Data := $1:Data ;
    JUMP BNZ (T1:Data) @1;
    ASSIGN T1:Data := $2:Data;
    LABEL @1} T1:Data  <== or ($1:Data, $2:Data)
2: {ASSIGN T1:Data := $1:Data ;
    JUMP BZ (T1:Data) @1 ;
    ASSIGN T1:Data := $2:Data  ;
    LABEL @1} T1:Data  <== and ($1:Data, $2:Data)
3:  LOADI($1:Addr,^2:Data):Data <== mem(offset($1:Addr,^2:Data))

Commands

2: JUMP BZ ($1:Data) @1;
   !1;
   JUMP BR () @2;
   LABEL @1;
   !2;
   LABEL @2
   <== if $1:Data then {!1} else {!2}
2: LABEL @1; JUMP BZ ($1:Data) @2; !1 ;JUMP BR () @1; LABEL @2
               <==  while $1 {!1}
1: ASSIGN #1:Data := $2:Data <== #1:Data := $2:Data
3: STORE ($1:Addr,$2:Data) <== $1:Addr <- $2:Data
```

Table 8: An example instruction selection rules file

```
var a:bool
var b:bool
var c:bool
fun f(var x:bool var y:bool var z:bool):int
[ { } 1]

fun g(var x:bool var y:bool var z:bool):bool
[ { } false]

{ t0:bool := mem(addr(a))
; t1:bool := mem(addr(b))
; t2:bool := mem(addr(c))
; t3:bool := g(t0, t1, t2)
; t3:bool := (t3&&t0)
; t5:bool := g(t1, t2, t0)
; t4:bool := g(t2, t1, t5)
; t4:bool := (t5||t4)
; t6:bool := (t2&&mem(addr(b)))
; t5:bool := not t6
; t3:int := f(t3, t4, t5)
; } (t3+t3)
```

Here is the function translated into pseudo-Jouette code:

```
 [cost=465]
   LOADI [ d1 ] addr(a) 0
 ; LOADI [ d2 ] addr(b) 0
 ; LOADI [ d3 ] addr(c) 0
 ; CALL [ d4 ] g (d1,d2,d3)
 ; MOVE [ d5] d4
 ; BZ @1 (d5)
 ; MOVE [ d5] d1
 ; LABEL @1
 ; MOVE [ d4] d5
 ; CALL [ d6 ] g (d2,d3,d1)
 ; CALL [ d5 ] g (d3,d2,d6)
 ; MOVE [ d7] d6
 ; BNZ @2 (d7)
 ; MOVE [ d7] d5
 ; LABEL @2
 ; MOVE [ d5] d7
 ; MOVE [ d8] d3
 ; BZ @3 (d8)
 ; LOADI [ d8 ] addr(b) 0
 ; LABEL @3
 ; MOVE [ d7] d8
 ; NOT [ d6 ] d7
 ; CALL [ d4 ] f (d4,d5,d6)
 ; ADD [ d1 ] d4 d4
```

2. Here is the second program with embedded commands removed and functions broken up (there are none!):

```
var b[40]:int
var c:int
var d:int

{ t0:int := (mem(addr(c))+3)
; t0:int := (t0*t0)
; t3:ref(int) := mem((addr(b)\t0))
; t1:int := (mem((mem(addr(b))\2))+mem(addr(d)))
; t2:int := (2*t1)
; (mem(addr(b))\2):int <- t2
; t3:int <- t2
; t0:int := 9
; } t0
```

   Here is the translation into pseudo-Jouette code:

```
var b[40]:int
var c:int
var d:int

 [cost=111]
   ADDI [ d1 ] LOADI(addr(c),0) 3
 ; MUL [ d1 ] d1 d1
 ; LOADAI [ a4 ] OFFSET(addr(b),d1) 0
 ; ADD [ d2 ] LOADI(LOADAI(addr(b),0),2) LOADI(addr(d),0)
 ; MUL [ d3 ] ADDI(<d0>:Data,2) d2
 ; STORE LOADAI(addr(b),0) 2 d3
 ; STORE a4 0 d3
 ; ADDI [ d1 ] <d0>:Data 9
 ; MOVE [ d1] d1
```

   Note here we still need to break up the instruction expressions.

3. Here is the third program with embedded commands removed and functions broken up:

```
var p[12]:int
fun f(var a:int var b:int):int
[ { } mem(fp(0,a))]
```

```
{ t5:ref(int) := (mem(addr(p))\7)
; t0:int := mem((mem(addr(p))\1))
; t4:ref(int) := mem(addr(p))
; t2:int := 1
; t1:int := 1
; t3:int := f(t2, t1)
; t1:int := mem((t4\t3))
; t0:int := f(t0, t1)
; t5:int <- (mem((mem(addr(p))\3))*7)
; t1:int := 1
; } t1
```

Here is the pseudo-Jouette code:

```
var p[12]:int

function f(var a:int var b:int):int
[
 [cost=11]
   LOADI [ d1 ] fp(0,a) 0
]

 [cost=310]
   OFFSETI [ a6 ] LOADAI(addr(p),0) 7
 ; LOADI [ d1 ] LOADAI(addr(p),0) 1
 ; LOADAI [ a5 ] addr(p) 0
 ; ADDI [ d3 ] <d0>:Data 1
 ; ADDI [ d2 ] <d0>:Data 1
 ; CALL [ d4 ] f (d3,d2)
 ; LOADI [ d2 ] OFFSET(a5,d4) 0
 ; CALL [ d1 ] f (d1,d2)
 ; STORE a6 0 MUL(LOADI(LOADAI(addr(p),0),3),ADDI(<d0>:Data,7))
 ; ADDI [ d2 ] <d0>:Data 1
 ; MOVE [ d1] d2
```

Note that the expressions have cleraly yet to be broken up!

4.9 An example of a Jouette rules file

```
Expressions

3: ADD($1,$2):Data <== add ($1:Data,$2:Data)
6: MUL($1,$2):Data <== mul($1:Data,$2:Data)
3: SUB($1,$2):Data <== sub($1:Data,$2:Data)
10: DIV($1,$2):Data <== div($1:Data,$2:Data)
```

```
2: ADDI(<d0>:Data,^1):Data  <== ^1:Data
2: ADDI($2,^1):Data  <== add ($2:Data,^1:Data) | add(^1:Data,$2:Data)
2: SUBI($1,^2):Data  <== sub ($1:Data,^2:Data)

3: OFFSET ($1,$2):Addr <== offset ($1:Addr,$2:Data)
3: OFFSETI ($1,^2):Addr <== offset ($1:Addr,^2:Data)
2: {ASSIGN T1:Data := $1;
    JUMP BNZ (T1:Data) @1;
    ASSIGN T1:Data := $2;
    LABEL @1} T1:Data  <== or ($1:Data, $2:Data)
2: {ASSIGN T1:Data := $1 ;
    JUMP BZ (T1:Data) @1 ;
    ASSIGN T1:Data := $2  ;
    LABEL @1} T1:Data  <== and ($1:Data, $2:Data)
3: {JUMP BLT (SUB($1,$2):Data) @1 ;
    ASSIGN T1:Data := ADDI(<d0>:Data,1):Data;
    JUMP BR () @2;
    LABEL @1 ;
    ASSIGN T1:Data := <d0>:Data ;
    LABEL @2 } T1: Data
     <== lt ($1:Data,$2:Data) | gt ($2:Data,$1:Data)
3: {JUMP BGE (SUB($1,$2):Data) @1 ;
    ASSIGN T1:Data := ADDI(<d0>:Data,1):Data;
    JUMP BR () @2;
    LABEL @1 ;
    ASSIGN T1:Data := <d0>:Data ;
    LABEL @2 } T1: Data
     <== ge ($1:Data,$2:Data) | le ($2:Data,$1:Data)
10: LOADI($1,^2):Data <== mem(offset($1:Addr,^2:Data))
10: LOADAI($1,^2):Addr <== mem(offset($1:Addr,^2:Data))
10: LOADI($1,0):Data <== mem($1:Addr)
10: LOADAI($1,0):Addr <== mem($1:Addr)
4: NOT($1):Data <== not($1:Data)

Commands

2:  JUMP BRZ ($1) @1;
    !1;
    JUMP BR () @2;
    LABEL @1;
    !2;
    LABEL @2
     <== if $1:Data then {!1} else {!2}
4:  JUMP BLT (SUB($2,$1):Data) @1;
    !1;
    JUMP BR () @2;
    LABEL @1;
    !2;
    LABEL @2
     <== if lt($1:Data,$2:Data) then {!1} else {!2}
2: LABEL @1; JUMP BRZ ($1) @2; !1 ;JUMP BR () @1; LABEL @2
               <==  while $1:Data {!1}
2: LABEL @1; JUMP BLT (SUB($2,$1):Data) @2; !1 ;JUMP BR () @1; LABEL @2
               <==  while lt($1:Data,$2:Data) {!1}
                 |  while gt($2:Data,$1:Data) {!1}
1: ASSIGN #1:Data := $2 <== #1:Data := $2:Data
1: ASSIGN #1:Addr := $2 <== #1:Addr := $2:Addr
15: MOVEM ($1,$2) <== $1:Addr <- mem($2:Addr)
10: STORE ($1,0,$2) <== $1:Addr <- $2:Data
10: STOREA ($1,0,$2) <== $1:Addr <- $2:Addr
10: STORE ($1,^3,$2) <== offset($1:Addr,^3:Data) <- $2:Data
10: STOREA ($1,^3,$2) <== offset($1:Addr,^3:Data) <- $2:Addr
```