Program Decompilation: From MinMIPS to MinC

Sara Moreira
(up201404984@fc.up.pt)

Departamento de Ciências de Computadores,
Faculdade de Ciências da Universidade do Porto

Abstract

Decompilation is a form of reverse engineering based on the reconstruction of a high-level language program from a machine language code. It is a growing field that has has a lot of applications, mainly in areas like security and software maintenance. In this work, we present a decompiler that reconstructs a MinC program from a MinMIPS input. MinMIPS and MinC are subsets of MIPS assembly language and C, accordingly. This decompiler deals with the recovery of high-level types from untyped assembly code following constraints and type inference rules, and the reconstruction of basic C instructions, flow-control structures, and functions.

1 Introduction

Decompilation is the name given to the process of reconstructing a high-level language program from a low-level code. It is a form of reverse engineering that does the opposite of what a compiler does.

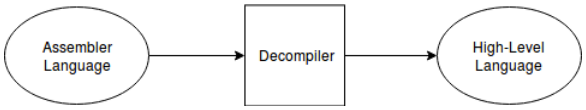

Figure 1: Decompiler

There are several legitimate and useful applications for decompilation, as this process provides an unquestionable support in areas such as software maintenance and security [1]. A decompiler can be a great help when some software containing legacy code needs contact with more recent software and the original source code has been lost. In addition, a decompiler that can generate reliable high-level code, can be a valuable tool in searching for malicious code and understanding how it works, and can provide great support in looking for code errors and vulnerabilities.

The steps involved in the compilation and decompilation processes are similar. However, a decompiler has its work hampered. Unlike the code provided to the compiler, the code provided do the decompiler doesn't carry clear and direct information about variables, types, functions, structures, etc. That said, it is important to mention that, however sophisticated a decompiler is, it is virtually impossible to guarantee that it will not have any imperfections, or even that it could produce an incorrect output.

In this work we studied program reconstruction techniques, from an assembler language in a MIPS arquitecture, and implemented a small decompiler based on those techniques, for a subset of a high-level language. An example of the application of the decompiler can be seen in Figure 2.

```
add $a0 $t0 $t1
ola:
        sub $a1 $a0 $t0
        bneq $a1 $a1 ola
jal hm
sub $t2 $t1 $t1
hm:
        add $a0 $a1 $t3
        mult $a0 $t1 0
        jr $ra
```

```
int a0;
int t0;
int t1;
int a1;
int v0;
int t2;
int t3;
a0 = t0+t1;
while (!(a1==a1)) {
a1 = a0 t0;
}
int v0=hm(a1, a0);
t2 = t1 t1;
hm(int a1, int a0){
a0 = a1+t3;
v0 = a0*t0;
return v0;
}
```

Figure 2: MIPS code and recovered C code

The main challenges presented by decompilation addressed in this work were flow control reconstruction and data type reconstruction. Briefly, this decompiler retrieves the high-level types following a semantic

approach, as described in the article by E. Robbins [8]. The flow control reconstruction follows an approach based on the structural and semantic analysis of the instructions, and uses the information provided by the retrieved types as support.
Our decompiler was implemented in Haskell and the code is available in GitHub (*https://github.com/naosouasara*).
This article is organized as follows: In section 2 we present a brief evaluation of the state of the art on decompilation. Section 3 introduces the decompiler implementation. Section 4 shows some examples of the application of the decompiler. Finally, section 5 presents some final considerations about the work, the conclusions, and some perspectives for future work.

2 State of the art

The decompilation problem has a relatively long history that began around the 60s [4]. There are currently some practical results available, such as Hex-Rays [3], which is integrated into the IDA Pro disassembler (commercial leader).
There have been many different approaches to decompilation, some of which have discussed high-level type recovery. Tie [5] is a decompiler that presents a principled type recovery system. Hex Rays [3], that we mentioned before, uses heuristics to atribuir simple local types. Rewards [6] performs type recovery through dynamic analysis, but is limited because it doesn't deal with flow-control.
In 1990 Mycroft [7] presented a solution based on type inference techniques. It followed an approach that begins by considering a register-based machine language. First, he converts the code to Single Static Assignment (SSA). Then, he creates constraints to assign one or more types to each statement. And finally, uses type inference to identify the most general type for each procedure, where occurs-check errors are repaired by the rebuilding of data structures.
Later, E.Robbins [8] presents a semantics-driven approach to type recovery, which uses a type-correct witness in a high-level language to ensure the type-correctness of inferred types for the low-level language. I put enphasis on these last two because they were the basis of this work, specially the MinX to MinC decompiler [8], in which we took great support, mainly for its MinC's type system and instruction decompilation rules.

3 Implementation

Figure 3 illustrates how the system components work together.
At the center of this system is the decompilation relation. This process takes a MinMIPS input and produces its MinC equivalent.
Our decompilation relation actually consists on two main steps, type decompilation and instruction decompilation. The first step is to recover the high-level

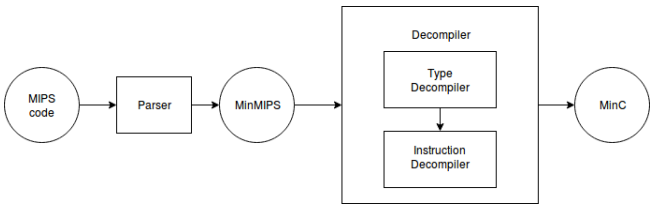

Figure 3: Overview

types. This process happens before the actual instruction decompilation because it provides helpful information for it.
To generate our MinMIPS structure we parse the MIPS code provided, using a parser generated by Haskell's Happy parser generator [2].

3.1 MinMIPS

Our MinMIPS language is a MIPS assembly language subset, that considers only a few of the operations that can be found in MIPS. We define our MIPS structure in terms of three categories, instructions i , operands *ops*, and operations *op*.

$\langle i \rangle$ ::= l
| *addi* $r1$ $r2$ i
| *add* $r1$ $r2$ $r3$
| *sub* $r1$ $r1$ $r3$
| *div* $r1$ $r2$ $r3$
| *mult* $r1$ $r2$ $r3$
| *beq* $r1$ $r2$ l
| *bneq* $r1$ $r2$ l
| *jal* l
| *jr* r
| *lw* r a
| *li* r i
| *sw* r a

The Haskell data to represent abstract syntax of MinMIPS is presented in Figure 4.
Our structure does not consider data declarations, al-

```
data MIPSInstruction =
 MInst Operation [Operand]
 | MLabel Label
 deriving (Show)

data Operation = Op String
 deriving(Show)

data Operand= Reg Register
 | Addr Int Register
 | Immdt Int
 | Lbl Label
```

Figure 4: MinMIPS data

though this is something that could be interesting to work with in the future, mainly to gather information on variable size, since it could be helpful to generate

```
li $t2, 25
lw $t3, 0($t2)
add $t4, $t2, $t3
sub $t5, $t2, $t3
```

Figure 5: Simple MinMIPS code

type constraints. The set of instructions we decided to consider can be divided in three groups, arithmetic instructions *add, sub, mult, div, addi*, branch and jump instructions *beq, bneq, jal, jr*, and data transfer instructions *lw, sw, li*.
There are four different types of operands, registers r, immediates i, addresses a, and labels l.
As we can see, we have two types of use for labels, one as instructions, and the other as branch and jump labels. It is important to consider both uses, because a branch label will serve as a reference to the instruction label of the same name, and this will be very useful for the decompilation of flow-control structures and functions.
Our registers follow the MIPS register convention, as it is relevant for our compiler to know that registers *$f0 – $f31* are floating point, *$a0 – $a3* are arguments for a call, and *$v0 and $v1* are return values of a call.
An address a is used by instructions that refer to memory, such as *lw* and *sw*. An address is formed by a base register and an offset.

3.2 MinC

Our MinC language is a C subset and its sintax was partially based on E.Robbins' [8] MinC language, however we keep a very C-like syntax, so it is easier to convert it into C.
MinC syntax consists of two categories, instructions i and expressions e.

$\langle i \rangle$ ::= $\langle e \rangle$
| $\langle e \rangle$ '=' $\langle e \rangle$
| 'if' $\langle e \rangle$ [$\langle i \rangle$]
| 'else' [$\langle i \rangle$]
| 'while' $\langle e \rangle$ [$\langle i \rangle$]
| 'return' $\langle e \rangle$
| $\langle label \rangle$ [$args$] $(x : t)$

$\langle e \rangle$ ::= c
| x
| [x]
| $\langle e \rangle$ '+' $\langle e \rangle$
| $\langle e \rangle$ '-' $\langle e \rangle$
| $\langle e \rangle$ '*' $\langle e \rangle$
| $\langle e \rangle$ '/' $\langle e \rangle$
| $\langle e \rangle$ '==' $\langle e \rangle$
| $\langle e \rangle$ '!=' $\langle e \rangle$
| $\langle label \rangle$ [$(x : t)$] [$\langle i \rangle$]

Where [] represents a list and $(x : t)$ represents a variable x and its type t.
The Haskell data to represent abstract syntax of MinMIPS is presented in Figure 6.

We can see that, as mentioned before, MinC uses

```
data CInstruction = Atrib Expr Expr
  | If Expr [CInstruction]
  | Else [CInstruction]
  | While Expr [CInstruction]
  | Impl Expr
  | Return Expr
  | Call Operand [Args] (Return,CType)

data Expr = Var String
  | Const String
  | Not Expr
  | Add Expr Expr
  | Sub Expr Expr
  | Mult Expr Expr
  | Div Expr Expr
  | Eq Expr Expr
  | Func Label [(Args,CType)][CInstruction]
  | Array String Int
```

Figure 6: MinC data

a very simplified C-like syntax. We consider constants c, variables x, arrays of variables, a small set of arithmetic, assignment, and relational operations, flow-control instructions, and functions. For our flow-control we use branch structure *if-else* or loop structure *while*. A function declaration considers the function name *label*, its arguments [$args$] and the return variable and its type $('return' : t)$. A function definition considers the function name *label*, its arguments and their types $(x : t)$, and the body of the function, which is a set of instructions [i]. Finishing the body of the funtion is the return value $'return'$.

We also borrow from E.Robbins' [8] MinC its type system.
This language features three kinds of types.

$$\theta = Int \mid Float$$
$$\beta = \theta \mid \alpha*$$
$$\alpha = \beta \mid [\beta]$$

The basic type θ can be either an int or a float, the compact type β can be a basic type or a pointer to a general type, the general type α can be a compact type, or an array of compact types.
Notice how we only distinguish between basic types *integer* and *float*, and not, for example, between types *int, char, long, short*, unlike E.Robbins that actually differs type *long* from *short*.

3.3 Decompiler

As mentioned above, our decompilation relation relates a MinMIPS input with its MinC equivalent. We translated this relation to a function that has two main steps: type decompilation and instruction decompilation.
After both these steps,we use a *pretty printer* to print our decompiled MinC code as C, which is easily done

because, as we said before, our MinC syntax is very C-like, so it is not necessary to perform any conversions. For simplicity reasons, we decided that our MinC variables would be named after their MinMIPS operand, for example, register *$t1* is translated to variable *t1*.

3.3.1 Type Decompiler

Our type decompiler follows a semantic and syntatic approach for the recovery of types. This process is based on the evaluation of the structure and semantic meaning of each MinMIPS instruction received.
For example, when our decompiler receives an *add $f1 $f2 $f3* instruction, we assume that the type of the MinC variables decompiled will be $Float$, because we know registers *$fi* are floating point. On *lw $t1 j($t2)* instruction, we assume that the MinC variable $t1$ is an Int and that $t2$ is a pointer to an array of Ints $Int[]*$. This is a fairly simple and flawed type recovery technique, to match our simple type system.
Figure 7 presents some of the rules the decompiler applies when receiving a MinMIPS instruction.
These rules are implemented in our main function, *recoverType*, which receives as an argument a list of MIPSInstructions and the list of types that have already been recovered L, L is a list of pairs consisting of variable x and its type t. Before anything else, our function checks if the type of each variable has already been recovered, and if so, it assigns the already recovered type to that variable.

MinMIPS Instruction	MinC Type and Variable
add $t1 $t2 $t3	Int t1, Int t2, Int t3
sub $f1 $f2 $f3	Float f1, Float f2, Float f3
mult $a0 $a1 $a2	Int a0, Int a1, Int a2
div $t1 $t2 $t3	Int t1, Int t2, Int t3
li $t1 i	Int t1
lw $t1 j($t2)	Int t1, Int[]* t2
sw j($t1) $f2	Int[]* t1, Int t2
move $t1 $t2	Type t1 = Type t2

Figure 7: Type decompilation

3.3.2 Instruction Decompiler

Our instruction decompiler takes into consideration some lists of information, it is important to describe these lists formally because of their relevance to the decompilation relation. We have list Ω , which is a list of MinC instructions that have been decompiled since the last instruction label has been checked, this is a very important procedure for the recontruction of *while* structures, list Δ is the list of branch and instruction labels that have been checked, Υ is the list of function labels that have been checked. We translate the decompilation relation to a function that receives these lists as arguments, aswell as the set of MinMIPS instructions do decompile, and a few more relevant information. This function translates a MinMIPS input to a set of MinC instructions, in which we include arithmetic, assignment, and relational operations, function calls and declarations, and flow-control structures.
In *Haskell* this is implemented by the top level function *getCInstruction*.

```
getCInstruction::[MIPSInstruction]
-> [CInstruction]
-> [CheckedBranchLabels]
-> [FunctionInfo]
-> [(CVar,CType)]
->[CInstruction]
```

This function receives as arguments a list of MinMIPS instructions, a list of MinC instructions that have already been decompiled (this will be explained when we talk about decompiling flow-control), a list of branch labels that have already beem checked, a list of information about functions, a list of MinC variables and their types, and returns a list of MinCInstructions.
The decompilation of instructions follows a set of rules, presented in Figure 8. As examples of cases implementing rules in Figure 8, we have:

```
getCInstruction
((MInst (Op "addi") ([Reg r1, Reg r2, Immdt i])):xs)
cinstr blbls ((args,label,called,inf):xs1) types=
(Atrib (Var r1) (Add (Const r2) (Const (show i))))
(getCInstruction xs cinstr blbls ((args,label,called,inf):
xs1) types)

getCInstruction
((MInst (Op "move") ([Reg r1, Reg r2])):xs)
cinstr blbls finfo types=
(Atrib (Var r1) (Var r2)):
(getCInstruction xs cinstr blbls finfo types)

getCInstruction
((MInst (Op "lw") ([Reg r1, Addr i r2])):xs)
cinstr blbls finfo types=
(Atrib (Var r1) (Array r2 i)):
(getCInstruction xs cinstr blbls finfo types)
```

These three cases are the implementation of the *addi*, *move* and *lw* instruction decompilation.

Flow control decompilation Our decompiler distinguishes beween two control-flow structures, *while* and *if-else*. A control-flow structure is detected when we receive a branch MinMIPS instruction. But how do we distinguish between branch stucture *if-else* and loop structure *while*? We have a function that keeps track of the labels that have already been checked.
When we receive a branch MinMIPS instruction, our main function checks if the label has already been checked. If so, we translate this instruction to a *while* loop. The body of the loop are the MinC instructions that have been decompiled since the label of the same name was checked.
If the label has not been checked, we translate it to a *if-else* structure. The decompilation of the body of the *if* structure is implemented by top level function *getLabelInstructions*.

Function Decompilation To decompile a function we need to consider two important parts, the function declaration and the function definition. We know a

```
getLabelInstructions::([MIPSInstruction],Label)
-> [CheckedCInstruction]
-> [CheckedBranchLabels]
-> [FunctionInfo]
->[(CVar, CType)]
->[CInstruction]
```

function is being called when we receive a *jal label* instruction, so when this happens, we save the label as a function label, so we know it has been checked and it is associated to a function. Later, when we receive a label instruction of the same name, we know we have to define the function of that name. The definition of a function is implemented by top level function *getFunctionInstruction*. For simplicity reasons, we assume that

```
getFunctionInstruction:: [MIPSInstruction]
-> [CInstruction]
-> [CheckedBranchLabels]
-> [FunctionInfo]
-> [(CVar, CType)]
->[CInstruction]
```

the return value of the function is saved on register *$v0* (and not in registers *$v0* and *$v1* like in MIPS), and since we know this, we assign the function to a new variable *v0*, and because we have already recovered the high-level types, we simply have to look for variable *v0* and its type in our list of recovered types. Another important step in decompiling a function is knowing which arguments we have to pass, and what are their types. We know that arguments are passed in registers *$a0* to *$a3*, so everytime we find one of those registers, we "check" them. When a function is called, we assume that all "checked" arguments until then are arguments for that function, and pass them in that call. To know their types we simply have to follow the same procedure we follow for the return value.

$$add\frac{}{\Sigma \vdash add\ r1\ r2\ r3 \rightsquigarrow r1 = r2 + r3}$$

$$addi\frac{}{\Sigma \vdash addi\ r1\ r2\ i \rightsquigarrow r1 = r2 + i}$$

$$sub\frac{}{\Sigma \vdash sub\ r1\ r2\ r3 \rightsquigarrow r1 = r2 - r3}$$

$$mult\frac{}{\Sigma \vdash mult\ r1\ r2\ r3 \rightsquigarrow r1 = r2 * r3}$$

$$div\frac{}{\Sigma \vdash div\ r1\ r2\ r3 \rightsquigarrow r1 = r2/r3}$$

$$move\frac{}{\Sigma \vdash move\ r1\ r2 \rightsquigarrow r1 = r2}$$

$$lw\frac{}{\Sigma \vdash lw\ r1\ j(r2) \rightsquigarrow r1 = r2[j]}$$

$$sw\frac{}{\Sigma \vdash sw\ j(r1)\ r2 \rightsquigarrow r1[j] = r2}$$

$$li\frac{}{\Sigma \vdash li\ r1\ i \rightsquigarrow r1 = i}$$

$$beq\frac{l \in \Delta \quad \Omega}{\Sigma \vdash beq\ r1\ r2\ l \rightsquigarrow' while'(r1 == r2); \Omega}$$

$$bneq\frac{l \in \Delta \quad \Omega}{\Sigma \vdash bneq\ r1\ r2\ l \rightsquigarrow' while'\ 'not'\ (r1 == r2); \Omega}$$

$$beq\frac{l \notin \Delta \quad \Omega}{\Sigma \vdash beq\ r1\ r2\ l \rightsquigarrow' if'(r1 == r2);;' else';}$$

$$bneq\frac{l \notin \Delta \quad \Omega}{\Sigma \vdash beq\ r1\ r2\ l \rightsquigarrow' if''not'(r1 == r2);\mathrm{L(l)};' else'}$$

Where $L(l)$ represents the function *getLabelFunction*, mentioned above.

Figure 8: Instruction decompilation rules

4 Examples

In listings 1, 2, 3 and 4 we present a few examples of the application of our decompiler. These examples cover the main focuses of our decompiler, arithmetic instructions, flow-control structures, and functions.

Listing 1: MinMIPS to MinC arithmetic instruction

```
add $t0 $t1 $t2
```

```
int t0;
int t1;
int t2;
t0 = t1+t2;
```

Listing 2: MinMIPS branch to MinC instruction

```
add $t0 $t1 $t2
beq $t0 $t1 test
sub $t1 $t1 t2
test:
mult $t1 $t1 $t2
```

```
int t0;
int t1;
int t2;
t0 = t1+t2;
if (t0==t1){
t1 = t1*t2;
}
else {
t1 = t1 t2;
}
```

Listing 3: MinMIPS branch to MinC *While* instruction

```
add $t0 $t1 $t2
test:
mult $t1 $t1 $t2
beq $t0 $t1 test
sub $t1 $t1 t2
```

```
int t0;
int t1;
int t2;
t0 = t1+t2;
while (t0==t1) {
t1 = t1*t2;
}
t1 = t1 t2;
```

Listing 4: MinMIPS *jal label* instruction to MinC function *label*

```
add $a0 $t1 $t2
jal func
func:
mult $v0 $t1 $t2
jr $ra
```

```
int a0;
int t1;
int t2;
int v0;
a0 = t1+t2;
int v0=func(a0);
func(int a0){
v0 = t1*t2;
return v0;
}
```

5 Final Considerations

We have described a system that decompiles MIPS assembly code to C. It can sucessfully reconstruct basic instructions, flow-control structures, and function calls and definitions. It can also recover the high-level types described in our type system.

We presented the syntax for our subset languages, MinMIPS and MinC, our system design, our decompilation rules, and demonstrated its different applications with a few relevant examples.

As mentioned, we do not consider data declaration on our MinMIPS structure, which could be a goal for the future, since this part of the code could provide some helpful information on variable size, and knowing variable size could be a key factor to distinguishing between *short, long, char* and *int* types. We also haven't discussed MinMIPS values saved on stack and not on registers, which proves to be a big flaw in our system since we are decompiling functions and have no way of dealing with local variables. We have yet to experiment the decompiler on larger bodies of code, but we're fairly happy with the results, despite its flaws.

References

[1] The decompilation wiki. `http://program-transformation.org/Transform/WhyDecompilation`.

[2] Happy: The parser generator for haskell. `https://www.haskell.org/happy/`.

[3] Hex rays. `https://www.hex-rays.com/`.

[4] Cristina Cifuentes. *Reverse compilation techniques*. Queensland University of Technology, Brisbane, 1994.

[5] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. Internet Society, 2011.

[6] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*, page 5. CERIAS-Purdue University, 2010.

[7] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming*, pages 208–223. Springer, 1999.

[8] Edward Robbins, Andy King, and Tom Schrijvers. From minx to minc: semantics-driven decompilation of recursive datatypes. In *POPL*, pages 191–203. ACM, 2016.