Type-Based Decompilation[⋆]
(or Program Reconstruction via Type Reconstruction)

Alan Mycroft

Computer Laboratory, Cambridge University
New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK
`http://www.cl.cam.ac.uk/users/am`

Abstract. We describe a system which decompiles (reverse engineers) C programs from target machine code by type-inference techniques. This extends recent trends in the converse process of compiling high-level languages whereby type information is preserved during compilation. The algorithms remain independent of the particular architecture by virtue of treating target instructions as register-transfer specifications. Target code expressed in such RTL form is then transformed into SSA form (undoing register colouring etc.); this then generates a set of type constraints. Iteration and recursion over data-structures causes synthesis of appropriate recursive C `structs`; this is triggered by and resolves occurs-check constraint violation. Other constraint violations are resolved by C's casts and `unions`. In the limit we use heuristics to select between equally suitable C code—a good GUI would clearly facilitate its professional use.

1 Introduction

Over the last forty years there has been much work on the compilation of higher-level languages into lower-level languages. Traditionally such lower-level languages were machine code for various processors, but there has been growing widening of the concept of compilation on one hand to permit the lower-level language to be a language like C (often viewed as a 'universal assembler language') and on the other to accompany the translation of terms by a corresponding translation of types—good exemplars are many internal phases of the Glasgow Haskell Compiler [4] which is taken to its logical conclusion in Morrisett et al.'s [8] introduction of 'Typed Assembly Language'. A related strand is Necula and Lee's [9] compiler for proof-carrying code in which user types (including a richer set of types containing value- or range-specification) and compiler-generated types or invariants accompany target code ('proof-carrying code') to enable code to be safely used within a security domain.

Two points which can be emphasised are:

- preserving type information increases the reliability of a compiler by allowing it (or subsequent passes) often to report on internal inconsistency if an invalid transformation occurs instead of merely generating buggy code; and

---

[⋆] A preliminary form of this work was presented at the APPSEM'98 workshop in Pisa.

- compilers are in general many-to-one mappings in which the target code is selected from various equivalent target-code sequences by some notion of efficiency—the more optimising a compiler, in general, the greater the number of source-code phrases that map to a given (generable) target-code sequence.

We consider the use of types and type-inference for the reverse process of decompilation, often called reverse engineering. For the purposes of this paper we take the higher-level code to be C and the lower-level code to be register transfer language (RTL).[1] RTL can be used to express various machine codes in architecture independent manner, but in examples we often use a generic RISC-like instruction set. Another important application which drove this work was a large quantity of BCPL [10] legacy code. BCPL was an untyped fore-runner of C popular at Cambridge for low-level implementation until its replacement with ANSI C around 10 years ago. Being untyped it has a single notion of *vector* which conflates the notions of array and record types in the same way that assembler code does. BCPL is easily translatable to RTL code (and indeed source names can be preserved within RTL as annotations) but the challenge was to invent appropriate structure or array types for data-structures just represented by pointers to vectors.

One might wonder where the RTL code comes from. It can be obtained by simple disassembly (and macro-expansion of instructions to RTL form) of code from assembler files, from object files, directly from compiler output or even from DLL's. Note that currently we assume that code is reasonably identified from data and in particular the current system presumes that procedure boundaries (and even—but less critically—procedure names) are available.

Now we turn to one of the central issues of decompilation—that compilation is a many-to-one map means that we must choose between various plausible alternative high-level representations of given RTL code. This is instantly obvious for the names of local variables which are in general lost in compilation and need to be regenerated; although in general we can only give these rather boring names, we can also recover information from a relocation or symbol table (e.g. in a ELF executable) or from a GUI-driven database to aid serious redevelopment of legacy code. However, there are more serious issues. Identifying loops in a reducible flowgraph is fairly easy but since a good compiler will often translate a "`while` (e) C" loop to a loop of the form

$$\texttt{if}\ (e)\ \{\ \texttt{do}\ C\ \texttt{while}\ (e);\ \}$$

we must be prepared to select between or offer the user a choice between various alternatives much like names above.

Note that we do not expect to have types [8] or assertions [9] in the machine code (but if we do these may significantly aid decompilation—it seems unfortunate if aids to program reliability and security make the code-breakers task easier too!). See section 7.

---

[1] Note the notion of source- and target-language is slightly tangled for a decompiler and so we will stick to C and RTL for concreteness.

We briefly justify decompilation. Apart from the obvious nefarious uses, there are real desires (e.g. in the telecoms industry) to continue to exploit legacy code with guaranteed equivalence to its previous behaviour. Additionally, we expect this project to cast further light on the uses of types of various forms in compilation, including proof-carrying code.

Apart from the glib statement that we decompile RTL to C, certain things do need to be made more precise. We will assume that the RTL has 8-, 16-, 32- and (possibly) 64-bit memory accesses and additionally a push-down stack for allocating temporaries and locals via `push` and `pop`. Moreover, the generated C will assume that `char`, `short`, `int` and `long` will represent these types. `unsigned` can be used as a qualifier as demanded by the code (e.g. triggered by unsigned division, shift or comparison, or by user GUI interaction) but otherwise `signed` forms of these types are generated. Pointers are currently assumed to be 32-bit values and again can only be distinguished from `int` values by their uses. We will see type-inference as the central driver of this process.

This work represents a position intermediate between traditional reverse engineering viewpoints. On one hand, there is decompilation work which has mainly considered control restructuring and tended to leave variables as `int`, to be type-cast on use as necessary (Cifuentes [1] is a good example). On the other hand, the formal methods community has tended to see reverse engineering as reconstructing invariants and specifications (e.g. by Hoare- or Dijkstra-style weakest precondition or strongest postcondition techniques—see for example Gannod and Cheng [5]) from legacy code so that it may be further manipulated. It is claimed that a type-based approach can be used for gross-level structuring automatically (possibly with a GUI driver for major choice resolution) whereas exact formal methods techniques are more limited in the size of acceptable problem (e.g. due to the need to prove theorems).

2 Intuitive example

Consider the following straight-line code

```
f:      ld.w  4[r0],r0
        mul   r0,r0,r0
        xor   r0,r1,r0
        ret
```

and a procedure calling standard which uses `r`i as argument and result registers. It is apparent that `f` has (at least) two arguments—see later—but for now we assume exactly two arguments. It is clear that `f` could be expressed as

```
int f(int r0, int r1)
{    r0 = *(int *)(r0+4);
     r0 = r0 * r0;
     r0 = r1 ^ r0;
     return r0;
}
```

However, if we break register uses into *live ranges* and give each a separate name we get:

```
int f(int r0, int r1)
{    int r0a = *(int *)(r0+4);
     int r0b = r0a * r0a;
     int r0c = r1 ^ r0b;
     return r0c;
}
```

Now it is apparent here that argument `r0` could be written as type (`int *`) instead of (`int`) which allows `*(int *)(r0+4)` to be replaced by `*(r0+1)` or its syntactically equivalent form `r0[1]`.[2] Moreover (modulo taking care not to violate any of C's rules concerning side-effects and *sequence points*), variables only used once can be folded into their referencing expressions yielding

```
int f(int *r0, int r1)
{    int r0a = r0[1];
     return r1 ^ (r0a * r0a);
}
```

There is now a further issue of stylistic choice as to whether the above code is preferred or the alternative:

```
int f(int *r0, int r1);
{    return r1 ^ (r0[1] * r0[1]);
}
```

which simply may have generated the original code as a result of a compiler's common sub-expression phase.

We recall the discussion in the introduction in which we observed that the more optimising a compiler the more pieces of code are mapped into a given, possibly optimal, form. A good correctness-preserving heuristic will select one (hopefully readable) form (a maximum-valued solution to various rules). A GUI user interface could select between wide-scale revision (i.e. seeking alternative local—to the constraint solver—maximum) or by demanding a choice between syntactic forms on a local—to the generated source code—basis.

3 SSA—Single Static Assignment

The Single Static Assignment (SSA) form (see e.g. [2]) is a compilation technique to enable repeated assignments to the same variable (in flowgraph-style code) to be replaced by code in which each variable occurs (statically) as a destination exactly once. We use the same technique for decompilation because we wish

---

[2] Note the possibility that `r0` could be given a type (`struct { int m0, m4, m8; }`) which would then lead to `int r0a = r0->m4;`. There is a notion of polymorphism here and we return to this point later.

to undo register-colouring optimisations whereby objects of various types, but having disjoint lifetimes, are mapped onto a single register.

In straight-line code the transformation to SSA is straightforward, each variable v is replaced by a numbered instance v_i of v. When an update to v occurs this index is incremented. This results in code like

```
v = 3; v = v+1; v = v+w; w = v*2;
```

(with next available index 4 for `w` and 7 for `v`) being mapped to

$$\texttt{v}_7\ \texttt{= 3; v}_8\ \texttt{= v}_7\texttt{+1; v}_9\ \texttt{= v}_8\texttt{+w}_3\texttt{; w}_4\ \texttt{= v}_9\texttt{*2;}$$

On path-merge in the flowgraph we have to ensure instances of such variables continue to cause the same data-flow as previously. This is achieved by placing a logical (single static) assignment to a new common variable on the path-merge node, which captures the effect of two separate assignments on the arcs leading to the path-merge node. This is conventionally represented by a so-called ϕ-function at entry to the path-merge node. The intent is that $\phi(x, y)$ takes value x if control arrived from the left arc or y if it arrived from the right arc; the value of the ϕ-function is used to define a new singly-assigned variable. Thus consider

```
if (p) { v = v+1; v = v+w; } else v=v-1;
w = v*2;
```

which would map to (only annotating `v` and starting at 4)

$$\begin{array}{l}\texttt{if (p) \{ v}_4\ \texttt{= v}_3\texttt{+1; v}_5\ \texttt{= v}_4\texttt{+w; \} else v}_6\texttt{=v}_3\texttt{-1;}\\ \texttt{v}_7\ \texttt{=}\ \phi(\texttt{v}_5\texttt{,v}_6)\texttt{; w = v}_7\texttt{*2;}\end{array}$$

In examples our variable names will be based on those of machine registers `r0`, `r1`, etc.—instances of these will be given an alphabetic suffix, thus `r0a`, `r4e`, etc.

4 Type reconstruction

Our type reconstruction algorithm is based on that of Milner's algorithm W [7] for ML; it shares the use of unification but involves a rather more complicated type system and delays unification until all constraints are available. Unification failure is used to trigger reconstruction of C types in a way which enables the constraint resolution failure to be repaired.

The C algebra of types does not neatly express the type concepts needed during type reconstruction[3] so we use an internal type algebra (for types t, `struct` members s and register types r) given by:

$$\begin{array}{l} t ::= \mathit{char} \mid \mathit{short} \mid \mathit{int} \mid \mathit{ptr}(t) \mid \mathit{array}(t) \mid \mathit{mem}(s) \mid \mathit{union}(t_1, \ldots, t_k) \\ s ::= n_1 : t_1, \ldots, n_k : t_k \\ r ::= \mathit{int} \mid \mathit{ptr}(t) \end{array}$$

---

[3] Indeed sometimes user-interaction may be desirable to select between C alternatives.

where the n_i range over natural numbers and $k > 0$. α and β are respectively also used to range over t and r (to highlight 'new' type variables generated during unification). The notation $\mathit{mem}(s)$ represents a storage type known to contain various types at identified offsets (i.e. it represents C's `struct`s and `unions` and, as we will see, may also represent arrays only accessed by constant subscripts). While the above type grammar allows user interaction to select all C types, for automatic inference it is convenient to require that all $\mathit{ptr}(t)$ types are of the form $\mathit{ptr}(\mathit{mem}(s))$ e.g. by selecting $s = 0 : t$. However, we will still feel free to write (e.g.) $\mathit{ptr}(\mathit{int})$ to be understood as shorthand. Finally, for the purposes of this paper, $\mathit{union}(t_1, \ldots, t_k)$ is not used as it can be simulated by $\mathit{mem}(0 : t_1, \ldots, 0 : t_k)$.

Each machine code instruction now generates constraints on the types of its operands in a straightforward manner. For example, adopting the notation that tk is the type ascribed to register rk, we have

instruction		generated constraint
`mov`	`r4,r6`	$t6 = t4$
`ld.w`	n`[r3],r5`	$t3 = \mathit{ptr}(\mathit{mem}(n : t5))$
`xor`	`r2a,r1b,r1c`	$t2a = \mathit{int}, t1b = \mathit{int}, t1c = \mathit{int}$
`add`	`r2a,r1b,r1c`	$t2a = \mathit{ptr}(\alpha), t1b = \mathit{int}, t1c = \mathit{ptr}(\alpha) \vee$
		$t2a = \mathit{int}, t1b = \mathit{ptr}(\alpha'), t1c = \mathit{ptr}(\alpha') \vee$
		$t2a = \mathit{int}, t1b = \mathit{int}, t1c = \mathit{int}$
`ld.w`	`(r5)[r0],r3`	$t0 = \mathit{ptr}(\mathit{array}(t3)), t5 = \mathit{int} \vee$
		$t0 = \mathit{int}, t5 = \mathit{ptr}(\mathit{array}(t3))$
`mov`	`#42,r7`	$t7 = \texttt{int}$
`mov`	`#0,r7`	$t7 = \texttt{int} \vee t7 = \mathit{ptr}(\alpha'')$

Note that overloaded C operators such as `+` naturally lead to disjunctive type constraints—compare `add` with `xor`. This also applies to indexed load and store instructions[4] and the constant zero, which is conventionally used to implement the null pointer constant.

Type unification is deferred until section 5, but for now it suffices to consider it as Herbrand unification where occurs-check failures are repaired rather than causing premature termination.

4.1 Inventing recursive data-types from loops or recursion

Consider the C recursive data type

```
struct A { int hd; struct A *tl; };
```

and the iterative and recursive procedures for summing its elements given in Figs. 1 and 2. (Note that for convenience the assembler code is given as a compiler might produce, with the original C code as comment and with generated label names, but note that code and type reconstruction only depends on the machine instructions.) Figs. 3 and 4 show the example assembler code in

---

[4] Here we assume such instructions do no automatic scaling.

```
;   int f(struct A *x)
;   {   int r = 0;
;       for (; x!=0; x = x->tl) r += x->hd;
;       return r;
;   }
;
f:
        mov     #0,r1
        cmp     #0,r0
        beq     L4F2
L3F2:
        ld.w    0[r0],r2
        add     r2,r1,r1
        ld.w    4[r0],r0
        cmp     #0,r0
        bne     L3F2
L4F2:
        mov     r1,r0
        ret
```

Fig. 1. Iterative summation of a list

```
;   int g(struct A *x)
;   {   return x==0 ? 0 : x->hd + g(x->tl);
;   }
;
g:
        push    r8
        mov     r0,r8
        cmp     #0,r8
        bne     L4F3
        mov     #0,r0
        br      L8F3
L4F3:
        ld.w    4[r8],r0
        jsr     g
        ld.w    0[r8],r1
        add     r1,r0,r0
L8F3:
        pop     r8
        ret
```

Fig. 2. Recursive summation of a list

SSA form and with generated type constraints. We now turn to the process of resolving the type constraints for `f`.

`f:`		$tf = t0 \to t99$
	`mov  r0,r0a`	$t0 = t0a$
	`mov  #0,r1a`	$t1a = int \vee t1a = ptr(\alpha_1)$
	`cmp  #0,r0a`	$t0a = int \vee t0a = ptr(\alpha_2)$
	`beq  L4F2`	
`L3F2:`	`mov  `ϕ`(r0a,r0c),r0b`	$t0b = t0a, t0b = t0c$
	`mov  `ϕ`(r1a,r1c),r1b`	$t1b = t1a, t1b = t1c$
	`ld.w 0[r0b],r2a`	$t0b = ptr(mem(0 : t2a))$
	`add  r2a,r1b,r1c`	$t2a = ptr(\alpha_3), t1b = int, t1c = ptr(\alpha_3) \vee$ $t2a = int, t1b = ptr(\alpha_4), t1c = ptr(\alpha_4) \vee$ $t2a = int, t1b = int, t1c = int$
	`ld.w 4[r0b],r0c`	$t0b = ptr(mem(4 : t0c))$
	`cmp  #0,r0c`	$t0c = int \vee t0c = ptr(\alpha_5)$
	`bne  L3F2`	
`L4F2:`	`mov  `ϕ`(r1a,r1c),r1d`	$t1d = t1a, t1d = t1c$
	`mov  r1d,r0d`	$t0d = t1d$
	`ret`	$t99 = t0d$

Fig. 3. Iterative sum in SSA form with generated type constraints

Type reconstruction (for `f` using the constraints in Fig. 3) now proceeds by:

Occurs-check constraint failure:

$$t0c = t0b = ptr(mem(4 : t0c)) = ptr(mem(0 : t2a)$$

Breaking cycle with:

$$\texttt{struct G \{ } t2a \texttt{ m0; } t0c \texttt{ m4; ...\}} \text{ i.e.}$$
$$t0c = ptr(mem(0 : t2a, 4 : t0c)) = ptr(\texttt{struct G})$$

A record is kept that this particular *mem* is represented by `struct G` which can then be used for printing types. Solving gives two solutions:

$$t0 = t0a = t0b = t0c = ptr(\texttt{struct G})$$
$$t99 = t1a = t1b = t1c = t1d = t2a = t0d = \texttt{int}$$
$$tf = ptr(\texttt{struct G}) \to \texttt{int}$$

and

$$t0 = t0a = t0b = t0c = ptr(\texttt{struct G})$$
$$t2a = \texttt{int}$$
$$t99 = t1a = t1b = t1c = t1d = t0d = ptr(\alpha_4)$$
$$tf = ptr(\texttt{struct G}) \to ptr(\alpha_4)$$

`g:`		$tg = t0 \to t99$
	`mov  r0,r0a`	$t0 = t0a$
	`push r8`	
	`mov  r0a,r8a`	$t8a = t0a$
	`cmp  #0,r8a`	$t8a = int \lor t8a = ptr(\alpha_2)$
	`bne  L4F3`	
	`mov  #0,r0a`	$t0a = int \lor t0a = ptr(\alpha_1)$
	`br   L8F3`	
`L4F3:`	`ld.w 4[r8a],r0b`	$t8a = ptr(mem(4 : t0b))$
	`jsr  g`	$t0 = t0b, t0c = t99$
	`ld.w 0[r8a],r1a`	$t8a = ptr(mem(0 : t1a))$
	`add  r1a,r0c,r0d`	$t1a = ptr(\alpha_3), t0c = int, t0d = ptr(\alpha_3) \lor$ $t1a = int, t0c = ptr(\alpha_4), t0d = ptr(\alpha_4) \lor$ $t1a = int, t0c = int, t0d = int$
`L8F3:`	`mov  φ(r0a,r0d),r0e`	$t0e = t0a, t0e = t0d$
	`pop  r8`	
	`ret`	$t99 = t0e$

Fig. 4. Recursive sum in SSA form with generated type constraints

The second solution is a parasitic solution which is caused by the effective overloading of addition and the constant zero on both pointers and integers as discussed earlier. It corresponds to creating the variable `r` in the original code as

```
char *r = 0;
```

and then adding on the (`int`) elements `x->hd` by address arithmetic. We believe that this false solution (it corresponds to code which is not strictly ANSI conformant) can be eliminated by enhancing the type system with a *weak pointer* type which is not suitable for arithmetic (cf. `void *` in C); however this awaits experiment.

Having obtained, then, the solution $tf = ptr(\texttt{struct G}) \to \texttt{int}$ with

$$\texttt{struct G } \{ \ t2a \ \texttt{m0; } \ t0c \ \texttt{m4; ...}\}$$

we can set about mapping the assembly code into appropriate C. Note that no information has been derived about the size of `struct G`; the use of the ellipsis above corresponds to the type "record type unknown apart from having field `m`" obtained for a Standard ML function such as

```
fun f(x) = x.m;
```

We model this in concrete C by creating an optional padding type `Tpad`.[5] It is now simple to translate the above code into the following C by re-constituting

---

[5] Unfortunately for us, C does not allow zero-sized types and so we must allow the field to be optional or allow the C pre-processor to macro-expand away `Tpad` if later information (e.g. from uses of the function `f`) indicate its size to be zero.

expressions from variables only used once and by pattern matching (out of the scope of this paper) for commands to obtain:

```
struct G { int m0; struct G *m4; Tpad m8; };
int f(struct G *x)
{   int r = 0;
    if (x != 0)
       do { r += x->m0; x = x->m4; } while (x != 0)
    return r;
}
```

Further pattern matching can reproduce the original `for` loop.

Incidentally, note that the recursive list summation function `g` results in an equivalent set of constraints and therefore can be similarly decompiled into:

```
int g(struct G *x)
{   int r;
    if (x==0)
        r = 0;
    else
    {   int t = g(x->m4);
        r = t + x->m0;
    }
    return r;
}
```

But why is this not nearly so close to the original (even if it is one of the common coding styles for this type of recursion)? Consider the expression

```
x->hd + g(x->tl)
```

which ANSI C declare to be implementation defined if `g()` side-effects `x->hd` and otherwise allows the compiler to choose whether to evaluate `x->hd` or the call to `g` first—sensible compilers would generally evaluate `g(x->tl)` first since this reduces register pressure. However, conversely, we are not in general at liberty to fold a *sequential* call to `g()` and an addition of `x->hd` into the original code and hence the above decompilation is as good as we can obtain under the assumption that procedure calls (e.g. `jsr g`) can affect memory arbitrarily. However, *if* we could determine that the call `jsr g` cannot result in side-effects (on `x->hd`) then the following simplifications are triggered:

- the code for the `else`-part could be reconstructed to

  ```
  r = x->m0 + g(x->m4);
  ```

  which is only valid C if `g()` cannot affect `x->m0`
- then, given that both consequents assign to `r`, the whole body simplifies to the original

  ```
  return x==0 ? 0 : x->hd + g(x->tl);
  ```

This short-coming of the present type-based approach could be remedied by extending function types to include details of side effects—using a *type and effect system* (also known as an *annotated type system*)—see for example [11]

Finally, we observe that all the suggested decompilations of `f` and `g` above yield identical target code when processed by the compiler (ncc) which was used to produce the sample code for decompilation. Of course, we might be able to use a better compiler the second time round!

4.2 When `structs` cannot resolve type conflicts

Although our decompiler needs internally a richer set of types than ML (e.g. we have seen that `ld.w 4[r0],r1` leads us to reason that `r0` may be a pointer to any type with a 32-bit component at offset 4, including both structures and arrays) we have exploited constraint gathering and solution by unification much as we might find in an ML compiler. In section 5 we will discuss the additional ordering on types (and non-Herbrand unification) occasioned by code which can reflect either array element or struct member access.

(Herbrand) unification may fail for two reasons. Firstly, a type variable may need to be unified with a term containing it—this is solved as above by synthesising recursive data types. Secondly, we may have a straightforward clash of type-constructors and it is to this case which we now turn.

Consider the code:

```
h:      ld.w    4[r0],r1
        xor     r1,r0,r0
        ret
```

where `r0` is constrained to be an `int` because of its appearance as the source of an `xor` instruction and as a pointer to store (containing an `int` at offset 4) due to the `ld.w` instruction. (Note that all the uses of `r0` except for the destination of the `xor` instruction form a single live range and so the transformation to SSA form used in the introduction does not help here.) So we attempt to unify `int` with $ptr(mem(4 : \texttt{int}))$ and find no solution. Such situations are deferred until the global set of constraint failures are available (here there are no more) and then the application to typing outlined by Gandhe et al. [3] for finding maximal consistent subsets of inconsistent sets is applied.

Here we find a benefit of using C as the high-level language for decompilation in that it can express such code by *casts* or `union` types. C's `union` type can express the solution trivially as

```
int h(union {int i; int *p;} x) { return x.p[1] ^ x.i; }
```

but this is not a very common (nor very readable) form of C and indeed is not strictly conforming in ANSI C (reading a `union` at a different type from what it was written is forbidden). We would prefer to restrict the synthesis of `union`s to within generated `struct`s which contain also a discriminator. Cast-based alternatives seem better in this case and we get three plausible solutions:

```
int h1(int x) { return *(int *)(x+4) ^ x; }
int h2(int *x) { return  x[1] ^ (int)x; }
struct h3arg { Tpad1 m0; int m; Tpad2 m8; };
int h3(struct h3arg *x) { return x->m ^ (int)x; }
```

Note we have suppressed the variant of `h1` in which `x` is cast to a new `struct` type which contains an `int` at offset 4; clearly a skilled program re-constructor might be able to specify the `*(int *)(x+4)` more precisely, but inventing a separate new datatype for each such access would clutter code for no clear benefit. We will prefer option `h3` by default (with the understanding that the generated `struct h3arg` will be unified with arguments of callers), leaving array creation to be triggered by non-constant indexing (or user GUI interaction); the next section investigates the choice between arrays and `struct`s in more detail.

Of course, one justification of using C in this paper is that the above assembler code could not plausibly be generated by any Haskell compiler— C is more expressive in this sense.

4.3 Arrays versus `structs`

The approach we have taken so far has been to use `struct`s whenever possible. While these, together with casts and address arithmetic, would suffice for decompilation, it is more rational to trigger array synthesis when indexing instructions occur, whether they be manifest:

```
ld.w    (r5)[r0],r3
```

or more indirectly coded (a non-constant `int` value being used for addition or subtraction at pointer type) such as

```
add     r5,r0,r1
ld.w    0[r1],r3
```

Such an indexing instruction (for the purposes of this discussion we will assume scaling is not done in hardware, thus the effective address is $(\texttt{r0}) + (\texttt{r5})$) generates constraints (as explained earlier):

$$\begin{aligned} \texttt{ld.w (r5)[r0],r3}\ & t0 = \mathit{ptr}(\mathit{array}(\beta)), t5 = \mathit{int}, t3 = \beta \vee \\ & t0 = \mathit{int}, t5 = \mathit{ptr}(\mathit{array}(\beta)), t3 = \beta \end{aligned}$$

where β is constrained to be a register type, i.e. `int` or $\mathit{ptr}(\alpha)$.

If the constraints for a given pointed-to type are all `struct` types (resulting from constant offsets) then the resulting unified type is also `struct` as in the previous subsection. Otherwise, if all accesses via a pointer are of the same *size*, e.g. all 32-bit accesses, then the unified type is *array*, otherwise a `union` type is generated, e.g. the constraints for

```
ld.b    0[r0],r1
ld.b    48[r0],r2
ld.w    (r5)[r0],r3
```

unify to yield

```
union G { struct { char m0; char pad1[47]; char m48; } u1;
          int u2[];
        } *r0;
```

Inferring limits for arrays requires, in general, techniques beyond those available to our type-based reconstruction. If presented with proof-carrying code [9] then array bounds could be extracted from code proved to be safe. To a large extent however, C programmers do not take great care with array size specifications, especially when passed as arguments since the C standard requires formal array parameters to be mapped to pointers thereby losing size information.

Although currently not implemented, note that a GUI could be used to direct that the above `union` should instead be decompiled as

```
struct G { char m0; char pad1[3]; int m4[15]; char m48; } *r0;
```

when it is clear to a user that the array is actually part of the struct. We return to this point in section 6.1.

5 Type Unification

Unification of our types is Herbrand-based with the following additional rules, i.e. the cases below are tried in order if Herbrand unification fails.

- type variable α unifies with type t containing α to yield $t[$`struct G`$/\alpha]$ with an auxiliary definition of `struct G` being produced.
- $\mathit{array}(t)$ and $\mathit{mem}(n_1 : t_1, \ldots n_k : t_k)$ unify to $\mathit{array}(t)$ when type $(\forall i) t_i = t$.
- $\mathit{mem}(s_1)$ and $\mathit{mem}(s_2)$ unify to $\mathit{mem}(s_1 \cup s_2)$; note that keeping conflicting items (e.g. $\mathit{mem}(0 : \mathit{int}, 1 : \mathit{char})$) is not an error since this may later be used to replace the member at offset zero with a `union` in the generated C.

6 Selecting C types for generated types

As noted earlier, generated types are more expressive than C types, and this sometimes means that a choice has to be made from among various possibilities. Moreover certain C types are less commonly used that others, e.g. a function parameter is more likely to be described as (`int *`) rather than (`int (*)[10]`), whereas appropriate uses would leave to identical target code. The default method for selecting types which result from unification is as follows:

- translate *char*, *short*, *int* as `char`, `short`, `int`;
- translate $\mathit{ptr}(t)$ to `T *` where `T` translates t;
- translate $\mathit{array}(t)$ to `T []` where `T` translates t;
- translate $\mathit{mem}(s)$ to:
    - `struct G` if `struct G` was generated during unification for this *mem*;
    - `T` if $s = (0 : t)$ and `T` translates t;

- `struct G` where `G` is a new struct definition laying out translated typed members of *s* at appropriate offsets (note this may require `unions` for overlapping members of *s* and may require padding members for unreferenced struct elements).

6.1 Unwelcome choice in reconstructing arrays and `structs`

The main problem which arises is due to the expressiveness (and defined storage layout) of C's `struct`, `union` and array types compared to those of Java (which may only contain another such object via a pointer). As in Fortran it can be hard to distinguish array type `int [10][10]` from `int[100]`. Similarly, arrays or structs containing other arrays or structs cannot in general be uniquely decoded, consider distinguishing objects `x1`, . . . `x3` defined by:

```
struct S1 { int a; int b[4]; int c; int d[4]; } x1;
struct S2 { int a; int b[4]; } x2[2];
struct S3 { struct S2 c,d; } x3;
```

We are exploring various options for constraint resolution when array indexing and `struct` selection occurs. Consider code like that discussed in section 4.3:

```
ld.b    0[r0],r1
ld.b    48[r0],r2
ld.w    (r5)[r0],r3
```

There are several possible ways to approximate this data-structure from the above information, including:

```
union T1 { char a[/*ATLEAST*/49]; int b[/*ATLEAST*/17]} *r0;
struct T2 { char a; char pad[3]; int b[15]; char c; } *r0;
```

The latter is appealing in that additional information, e.g. a

```
ld.b    16[r0],r4
```

instruction could cause natural, fuller-information, revision to

```
struct T2 { char a; char pad[3]; int b[3]; } (*r0)[4];
```

Finally, a current limitation is not exploiting stride information. For example, we could use information about the computation of `r0` to determine restrictions on sizes (and hence types) in the pointed-to values represented by `r5` in instructions like `ld.w  (r5)[r0],r3`.

7 Conclusions and further work

We have described a system which can decompile assembler code at the RTL level to C. It can successfully create `structs` both intra- and inter-procedurally and in doing can generate code close to natural source form.

We have not discussed the use of local variables stored on the stack rather than in registers. A simple extension can manipulate local stacks satisfactorily (essentially at the representative power of Morrisett et al.'s [8] Typed Assembly Language) when local variables are not address-taken. However, there are problems with taking addresses of local stack objects in that it can be unclear as to where the address-taken object ends—a `struct` of size 8 bytes followed by a coincidentally contiguously allocated `int` can be hard to distinguish from a `struct` of size 12 bytes.

Here it is worth remarking on the assistance given by proof-carrying code [9], particularly when the proof has been generated by a compiler, to our decompiler. Many of the questions which gave difficulty for decompilation concerned issues like: where arrays live inside a `struct`, is the data-structure really an array of `struct`s instead, or simply where a given array (determined by variable indexing) begins and ends. In general these are exactly the points which an accompanying proof-of-safety must address. This suggests that we can probably do much better at decompilation given the proof part—perhaps this shows that proof-carrying code is not a good idea for secret (or deliberately obfuscated) algorithms!

Note that the decompilation process does not depend intrinsically on C. We chose C because of its ability to capture most sequences of machine instructions naturally; casts can also represent type cheating in reconstructed source form. It also provides a good balance of problem-statement and tractability for this initial work. Of course, there are instruction sequences which are not translatable to C—the most obvious example is that C does not have label variables and so `jmp r0` cannot be decompiled (except possibly as a tail-recursive call to a procedure variable).

One could imagine a generalisation of this system where compiler translation rules (e.g. for Haskell) are made available to the decompiler to reconstruct rather more high-level languages. Failure of code to match such rules would in general indicate a call to a native (in the Java sense) procedure or that the proffered code cannot be expressed in the source code represented by the translation rules. This clearly links to the "formal methods" view of reverse engineering discussed in the introduction for inventing higher-level (than C) notions for existing code. Our type-based approach clearly can assist in this process in that it is coarser-grain than algebraic semantic approaches yet retains aspects of global understanding. Work in progress concerns identification and replacement of abstract data-types—in BCPL (a fore-runner of C) adjacent words in memory are required to have addresses differing by one which causes current translators for byte-addressed targets (whether via C or direct to target machine code) to generate large numbers of shift-left or shift-right by two instructions. Identifying "BCPLaddress" as an ADT and re-implementing it could eliminate all such shifts with the exception of those caused by users explicitly relying on this part of the standard.

Finally, we turn to performance: we as yet have no experimental results[6] for large bodies of code, but the ability to reconstruct datatypes for both iterative

---

[6] A student project is currently underway.

and recursive procedures is appealing over other techniques. Since the process of data-structure reconstruction depends only on finding cycles which in reality are likely to be quite short even in large programs, we are optimistic about the scalability of the techniques. In common with several type-based systems, interprocedural versions seem to come naturally and without great cost.

Acknowledgments

I would like to thank the anonymous referees for their helpful comments on the draft version of this paper. Thanks are also due to Pete Glasscock (a student on the Diploma in Computer Science at Cambridge in 1997–98) for illustrating, in his dissertation [6], what could be done in the way of decompilation without type-based reconstruction of source.

References

1. Cifuentes, C. *Reverse Compilation Techniques*, PhD thesis, Queensland University of Technology, 1994. Available as `ftp://ftp.csee.uq.edu.au/pub/CSM/dcc/decompilation_thesis.ps.gz`
2. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.W. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451-490, October 1991.
3. Gandhe, M., Venkatesh, G., Sanyal, A. Correcting Errors in the Curry System. In Chandrum V. and Vinay, V. (Eds.): Proc. of 16th conf. on Foundations of Software Technology and Theoretical Computer Science, LNCS vol. 1180, Springer-Verlag, 1996.
4. Glasgow Haskell Compiler.
5. Gannod, G.C. and Cheng, B.H.C. Using Informal and Formal Techniques for the Reverse Engineering of C Programs. Proc. IEEE International Conference on Software Maintenance, 1996.
6. Glasscock, P.E. An 80x86 to C Reverse Compiler. Diploma in Computer Science Dissertation, Computer Laboratory, Cambridge University, 1998.
7. Milner, R. A Theory of Polymorphism in Programming, *JCSS* 1978.
8. Morrisett, G., Walker, D., Crary, K. and Glew, N. From System F to Typed Assembly Language. Proc. 25th ACM symp. on Principles of Programming Languages, 1998.
9. Necula, G.C. and Lee, P. The Design and Implementation of a Certifying Compiler. Proc. ACM conf. on Programming Language Design and Implementation, 1998.
10. Richards, M. and Whitby-Strevens, C. BCPL—The Language and its Compiler, *CUP* 1979.
11. Tang, Y.M., Jouvelot, P. Effect Systems with Subtyping. Proc. ACM symp. on Partial Evaluation and Program Manipulation (PEPM), 1995.