

# Type-Based Decompilation<sup>1</sup>

(or program reconstruction via type reconstruction)

Alan Mycroft

Computer Laboratory, Cambridge University

[www.cl.cam.ac.uk/users/am](http://www.cl.cam.ac.uk/users/am)

**Draft of 28-Sep-98: please reference final version only**

## 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 40 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 [3] which is taken to its logical conclusion in Morrisett et al.’s [6] introduction of ‘Typed Assembly Language’. A related strand is Necula and Lee’s [7] 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 when an invalid transformation has occurred instead of merely generating buggy code; and
- 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

---

<sup>1</sup>A preliminary form of this work was presented at the APPSEM’98 workshop in Pisa.

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).<sup>2</sup> 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 [8] 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. Furthermore BCPL semantics required pointers to adjacent (typically 32-bit) words to differ by one. This 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. We plan to show how the concept of abstract data-type (here represented by C `structs` with a unique name) can be used to eliminate many of these in a more systematic method than previously.

Clearly extensions of this work to decompile machine code or C to higher-level languages like Haskell would be most interesting; we have chosen C because it provides a good balance of problem-statement and tractability.

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 flow-graph is fairly easy but since a good compiler will often translate a “`while (e) C`” loop to a loop of the form

```
if (e) { do C 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 expect to have types [6] or assertions [7] 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 6.

Justification for 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

---

<sup>2</sup>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.

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 that type-inference will be the central driver of this process.

## 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 `ri` 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 * r0b;
    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]`.<sup>3</sup> 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

---

<sup>3</sup>Note the possibility that `r0` alternatively has a type like `(struct { int m0, m4, m8; })` which would then yield `int r0a = r0->m4;`. There is a notion of polymorphism here and we return to this point later.

```

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. [1]) 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 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

$$v_7 = 3; v_8 = v_7+1; v_9 = v_8+w_3; w_4 = v_9*2;$$

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 arcs. Because flowgraph nodes (rather than edges) contain code this is conventionally represented by a invoking a so-called  $\phi$ -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 and  $y$  if it arrived from the right arc; the value of the  $\phi$ -function is used to define a new singly-assigned variable. Thus consider

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

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

$$\{ \text{if } (p) \{ v_4 = v_3+1; v_5 = v_4+w; \} \text{ else } v_6=v_3-1; \} v_7 = \phi(v_5, v_6); w = v_7*2;$$

```

;   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

```

Figure 1: Iterative summation of a list

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  $W[5]$  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.

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

```

;   int g(struct A *x)
;   {   return x==0 ? 0 : x->hd + g(x->t1);
;   }
;
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

```

Figure 2: Recursive summation of a list

```

f:
    mov     r0,r0a            $tf = t0 \rightarrow t99$ 
    mov     #0,r1a           $t0 = t0a$ 
    cmp     #0,r0a           $t1a = int \vee t1a = ptr(\alpha_1)$ 
    beq     L4F2             $t0a = int \vee t0a = ptr(\alpha_2)$ 
L3F2:
    mov      $\phi(r0a,r0c)$ ,r0b   $t0b = t0a, t0b = t0c$ 
    mov      $\phi(r1a,r1c)$ ,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     L4F2
L4F2:
    mov      $\phi(r1a,r1c)$ ,r1d   $t1d = t1a, t1d = t1c$ 
    mov     r1d,r0d          $t0d = t1a$ 
    ret                                 $t99 = t0d$ 

```

Figure 3: Iterative sum in SSA form with generated type constraints

<b>g:</b>		$tg = t0 \rightarrow t99$
	<b>mov</b> r0,r0a	$t0 = t0a$
	<b>push</b> r8	
	<b>mov</b> r0a,r8a	$t8a = t0a$
	<b>cmp</b> #0,r8a	$t8a = \text{int} \vee t8a = \text{ptr}(\alpha_2)$
	<b>bne</b> L4F3	
	<b>mov</b> #0,r0a	$t0a = \text{int} \vee t0a = \text{ptr}(\alpha_1)$
	<b>br</b> L8F3	
<b>L4F3:</b>	<b>ld.w</b> 4[r8a],r0b	$t8a = \text{ptr}(\text{mem}(4 : t0b))$
	<b>jsr</b> g	$t0 = t0b, t0c = t99$
	<b>ld.w</b> 0[r8a],r1a	$t8a = \text{ptr}(\text{mem}(0 : t1a))$
	<b>add</b> r1a,r0c,r0d	$t1a = \text{ptr}(\alpha_3), t0c = \text{int}, t0d = \text{ptr}(\alpha_3) \vee$ $t1a = \text{int}, t0c = \text{ptr}(\alpha_4), t0d = \text{ptr}(\alpha_4) \vee$ $t1a = \text{int}, t0c = \text{int}, t0d = \text{int}$
<b>L8F3:</b>	<b>mov</b> $\phi(\text{r0a},\text{r0d}),\text{r0e}$	$t0e = t0a, t0e = t0d$
	<b>pop</b> r8	
	<b>ret</b>	$t99 = t0e$

Figure 4: Recursive 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:

$$t0c = t0b = \text{ptr}(\text{mem}(4 : t0c))$$

Breaking cycle with:

```
struct G { t2a m0; t0c m4; ... }
```

(Note that  $\text{ptr}(\text{struct } G)$  is a concrete representation of  $\text{ptr}(\text{mem}(0 : t2a, 4 : t0c))$  an instance of both  $t0b = \text{ptr}(\text{mem}(0 : t2a))$  and  $t0b = \text{ptr}(\text{mem}(4 : t0c))$ .) Two solutions:

$$\begin{aligned} t0 = t0a = t0b = t0c &= \text{ptr}(\text{struct } G) \\ t99 = t1a = t1b = t1c = t1d = t2a = t0d &= \text{int} \\ tf &= \text{ptr}(\text{struct } G) \rightarrow \text{int} \end{aligned}$$

and

$$\begin{aligned} t0 = t0a = t0b = t0c &= \text{ptr}(\text{struct } G) \\ t99 = t1a = t1b = t1c = t1d = t2a = t0d &= \text{ptr}(\alpha_3) \\ tf &= \text{ptr}(\text{struct } G) \rightarrow \text{ptr}(\alpha_3) \end{aligned}$$

The second solution is a parasitic solution which is caused by the fact that a 32-bit integral value containing ‘22’ can only really be ascribed type **int** whereas the value ‘0’ can also be a NULL pointer. This is compounded by the fact that C’s addition operator can act on integers and pointers—hence the disjunction of constraints for

```
add    r2,r1,r1
```

(The unwanted solution would then correspond to creating the variable `r` in the original code as

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

and then adding on the 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 (c.f. `void *` in C); however this awaits experiment or the final version of this paper!

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

```
struct G { t2a m0; t0c 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<sup>4</sup> padding type `Tpad`. It is now simple to translate the above code into the following C by re-constituting 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 functions `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;
}
```

---

<sup>4</sup>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.



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)? The issue is that ANSI C allows the compiler to choose, given the expression

```
x->hd + g(x->t1)
```

whether to evaluate `x->hd` or the call to `g` first. Clearly the compiler made the sensible decision to evaluate first `g(x->t1)`. Hence we obtain the above decompilation. However, if we could show that the call `jsr g` could not 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);
```

- then, given that both consequents assign to `r`, the whole body simplifies to the original

```
return x==0 ? 0 : x->hd + g(x->t1);
```

This shows a short-coming of the present type-based approach which can be remedied by using a *type and effect system* [Jouvelot] (also known as an annotated type system) instead of our simple types. Whether the benefits would justify the extra cost is currently unclear.

## 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 : 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. [2] 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 ^ 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 *unions* to within generated *structs* 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; }
int h3(struct { Tpad1 m0; int m; Tpad2 m8; } *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 (except that the *struct* will be named and generated out-of-line so it can be also used by callers), leaving array creation to be triggered by non-constant indexing (or user GUI interaction); the next section investigates the choice between arrays and *structs* 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.

### 4.3 Arrays versus structs

The approach we have taken so far has been to use *structs* 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  $(r0) + (r5)$ ) generates constraints

$$\begin{aligned} \text{ld.w } (r5)[r0],r3 \quad t0 = ptr(array(\alpha)), t5 = int, t3 = \alpha \vee \\ t0 = int, t5 = ptr(array(\alpha)), t3 = \alpha \end{aligned}$$

where  $\alpha$  is constrained to be a 32-bit type, i.e. *int* or *ptr*( $\beta$ ).

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 [7] 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 5.1.

## 5 The type algebra for pointers, structs and arrays

This is complicated by the fact that C deliberately merges the notions of array and pointer. Consider the instructions

```
ld.w    4[r0],r1
xor     #1,r1,r2
```

where the `xor` exists largely to restrict the possible types of `r1` (and hence what `r0` points to) to (possibly `unsigned`) `int`. In this situation `r0` can be a pointer to an array of `ints` or to a `struct` containing an `int` at offset 4 as we saw above. It cannot be a pointer to a single object of type `int`. However, were the first instruction to have been

```
ld.w    0[r0],r1
```

then a third possibility arises that `r0` can simply be a pointer to an `int`. We observe that the type of `r0` can make parametric polymorphic sense at the machine code level (describing the field at given offset from `r0`) but results in overloading between various parametric types at the conceptual high-language level in a manner which is imperfectly expressible in C (e.g. ANSI C does not legitimise `(int (*) [])` as giving a pointer to an array of unknown size).

So, given

```
ld.w    n[r0],r1
```

where `r1` is constrained to be integer we generate a constraint that the type of `r0` is one of:

- `ptr(array(int))`, i.e. `int (*) []`;

- $ptr(mem(n : int))$  i.e. `struct { char pad[n]; int m; ... } *`; and
- $ptr(int)$ , i.e. `int *`, provided  $n = 0$ .

where the second form expresses partial information about a `struct`—here that only offset  $n$  is known—such `structs` are mapped to C inventing dummy padding types and members. Because of the difficulty in C in maintaining a proper separation of pointers to arrays and pointers to their elements (and having no type-based form to express restrictions on array size) we map the first form to the third on output of C.

The internal type algebra (for types  $t$ , `struct` members  $s$  and register types  $r$ ) is given by:

$$\begin{aligned}
 t & ::= \text{char} \mid \text{short} \mid \text{int} \mid ptr(t) \mid array(t) \mid mem(s) \mid union(t_1, \dots, t_k) \text{ with } k > 0 \\
 s & ::= n_1 : t_1, \dots, n_k : t_k \text{ with } k > 0 \\
 r & ::= \text{int} \mid ptr(t)
 \end{aligned}$$

where the  $n_i$  range over natural numbers. Although all of these types are necessary to express the full richness of C's types (and as such need to be used by a GUI), the automated type-unification is Herbrand-based with the following additional rules:

- $mem(s_1)$  and  $mem(s_2)$  unify to  $mem(s_1 \cup s_2)$ ; note that keeping conflicting items (e.g.  $mem(0 : int, 1 : char)$ ) is not an error since this may later be used to replace the member at offset zero with a `union` in the generated C.
- $array(t)$  and  $mem(n_1 : t, \dots, n_k : t)$  unify to  $array(t)$  when only a single type  $t$  occurs.
- $union$  is associative-commutative and can only occur immediately inside a  $ptr$  context;  $ptr(union(t_1, t_2))$  is the result of unifying  $ptr(t_1)$  and  $ptr(t_2)$  when this would fail due to having different type constructors.

## 5.1 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` 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[49 /* or more */]; int b[17 /* or more */] } *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];
```

## 6 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 [6] 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 [7], 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 `structs` 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 explicitly 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 have chosen C because of its ability to capture most sequences of machine instructions naturally; casts can also represent type cheating in reconstructed source form. 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 an 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.

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

Thanks to Pete Glasscock (a student on the Diploma in Computer Science at Cambridge in 1997–98) for illustrating, in his dissertation [4], what could be done in the way of decompilation without type-based reconstruction of source.

## References

- [1] 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.
- [2] 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.
- [3] Glasgow Haskell Compiler.
- [4] Glasscock, P.E. An 80x86 to C Reverse Compiler. Diploma in Computer Science Dissertation, Computer Laboratory, Cambridge University, 1998.
- [5] Milner, R. A Theory of Polymorphism in Programming, *JCSS* 1978.
- [6] 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.
- [7] Necula, G.C. and Lee, P. The Design and Implementation of a Certifying Compiler. Proc. ACM conf. on Programming Language Design and Implementation, 1998.
- [8] Richards, M. and Whitby-Stevens, C. BCPL—The Language and its Compiler, *CUP* 1979.