A C backend for Why3

Jacques-Pascal Deplaix - VALS

Abstract: WhyML is a programming language syntactically close to OCaml but without higher-order features. It allows the user to prove programs using theorem provers. I will present my internship work on a compiler that translates WhyML to C.

Introduction: What is WhyML?

WhyML is a high level language being part of the Why3 platform which allows to verify programs giving goals and conditions, automatically using automated provers. Is is a rich language for specifications and programming.
The Why3 platform allows you to verify a given program using multiple provers and even allows you to prove manually using Coq or Isabelle.

WhyML: An example

```
module Demo
  use import int.Fact
  use import int.Int

  let rec fact x
    requires { x >= 0 }
    variant  { x }
    ensures  { result = fact x }
  = if x = 0 then 1 else x * fact (x-1)
end
```

A C backend: Motivations

First of all the motivation for extracting from WhyML is to have a program correct by construction.

Motivations for my work:

- Allows to be used in a C program
- Allows to avoid the use of GC in certain cases via the region system

Difficulties

Differences between C and WhyML introduce some well known difficulties:

Difficulties

Differences between C and WhyML introduce some well known difficulties:

- Polymorphism

Difficulties

Differences between C and WhyML introduce some well known difficulties:

- Polymorphism
- Variants

Difficulties

Differences between C and WhyML introduce some well known difficulties:

- Polymorphism
- Variants
- GC

Difficulties

Differences between C and WhyML introduce some well known difficulties:

- Polymorphism
- Variants
- GC
- Pattern matching

Difficulties

Differences between C and WhyML introduce some well known difficulties:

- Polymorphism
- Variants
- GC
- Pattern matching
- Exceptions

Difficulties

Differences between C and WhyML introduce some well known difficulties:

- Polymorphism
- Variants
- GC
- Pattern matching
- Exceptions
- Partial applications & Inner functions

Polymorphism & value represatation

Polymorphism requires a unified representation of data.
Solution: All values are pointers (but you can avoid boxing for int32 for example).

```
typedef void* value;
```

Example:

```
let ite (b : bool) (x : 'a) (y : 'a) : 'a = if b then x else y
```

```
value F_M_demo__Demo__ite(value b, value x, value y, value* Env) {
    value X__183 = NULL;
    if(b == T_why3__Bool__True)
        X__183 = x;
    else
        X__183 = y;
    return X__183;
}
```

Variants

Constructors with no arguments (constant) are global (like T_why3__Bool__True in the previous slide) Constructors with arguments are allocated.

Each variant constructor contains a tag (an int) which is used to discriminate.

```
struct variant {int key; value* val;};
```

Note: Records are a special case of variants

Example:

```
type t 'a = Nil | Cons 'a (t 'a)
```

```
let cons l = Cons True l
```

Variants

Gives:

```
struct variant X__22 = {0, NULL};
struct variant* T_demo__Demo__Nil = &X__22;

value F_M_demo__Demo__cons(value l, value* Env)
{
    value* X__186 = GC_malloc(sizeof(value) * 2);
    X__186[0] = T_why3__Bool__True;
    X__186[1] = l;
    struct variant* X__187 = GC_malloc(sizeof(struct variant));
    X__187->key = 1;
    X__187->val = X__186;
    return X__187;
}
```

GC

We need a GC in order to collect data which are allocated on the heap and which get out of the scope.

Solution: Boehm GC (conservative GC)

A conservative is the easiest way if you need a GC. You just need to replace calls to malloc by GC_malloc (for boehm) and it works. It scans the heap and the stack searching for pointers. If a pointer is not referenced, it can be freed.

WhyML has, internally, a region system which is used in order to track aliases. It can be used to avoid some allocations.

Pattern matching

Very close to OCaml pattern matching but without WHEN clauses.
They are compiled using an internal function which compiles complicated patterns into simple patterns. Compiled in C using switches on the tag.

Example:

```
let rec length l = match l with
  | Nil -> ...
  | Cons x r -> ...
  end
```

Pattern matching

Gives:

```
value F_M_demo__Demo__length(value l, value* Env) {
  struct variant* X_189 = l;
  value* X_190 = X_189->val;
  value X_191 = NULL;
  switch(X_189->key) {
    case 0: ... break;
    case 1:
      value x = X_190[0];
      value r = X_190[1];
      ... break;
  }
  return X_191;
}
```

Exceptions

WhyML exceptions are Very close to OCaml exceptions too but cannot be used as first-class value.

Solution: setjmp/longjmp

Example:

```
exception E int

let f x raises {E -> true} = raise (E x)
let g () = try f 42 with E x -> x end
```

Exceptions

Gives:

```
typedef char const * exn_tag;
struct exn {exn_tag key; value val;};
__thread struct exn* Exn = NULL;

exn_tag M_demo__Demo__E = "M_demo.Demo.E";

value F_M_demo__Demo__f(value x, value* Env, jmp_buf Exn_buf)
{
    struct exn* X__193 = GC_malloc(sizeof(struct exn));
    X__193->key = M_demo__Demo__E;
    X__193->val = x;
    Exn = X__193;
    longjmp(Exn_buf, 1);
    return NULL;
}
```

Exceptions

```
value F_M_demo__Demo__g(value us, value* Env) {
    jmp_buf X__195;
    if(setjmp(X__195) == 0) {
        value o = int_create_from_str("42", 10);
        struct closure* X__198 = M_demo__Demo__f;
        value X__199 = X__198->f;
        value (*X__200)(value, value*, jmp_buf Exn_buf) = X__199;
        return X__200(o, X__198->env, X__195);
    }
    else {
        exn_tag X__202 = Exn->key;
        if(X__202 == M_demo__Demo__E)
            return Exn->val;
        else
            abort();
    }
}
```

Partial applications & Inner functions

Even if WhyML has no higher order features, it allow to returns local functions which needs an environment. In case of partial applications, a new function will be created dynamically which fully apply the original function with arguments already applied passed through the Env parameter.

Solution: closure

```
struct closure {value f; value* env;};
```

Example:

```
let f1 x =
  let g y = x + y in
  g

let f2 () = (f1 2) 42
```

Partial applications & Inner functions

```
value F_g(value y, value* Env) {
    return int_add(Env[0], y);
}

value F_M_demo__Demo__f1(value x, value* Env) {
    value X__116[1] = {NULL};
    struct closure* X__117 = GC_malloc(sizeof(struct closure));
    value* X__118 = GC_malloc(sizeof(value) * 1);
    X__118[0] = x;
    X__117->f = F_g;
    X__117->env = X__118;
    X__116[0] = X__117;
    return X__116[0];
}

struct closure X__121 = {F_M_demo__Demo__f1, NULL};
struct closure* M_demo__Demo__f1 = &X__121;
```

Partial applications & Inner functions

```
value F_M_demo__Demo__f2(value us, value* Env)
{
    value o = int_create_from_str("42", 10);
    value o1 = int_create_from_str("2", 10);
    struct closure* X__124 = M_demo__Demo__f1;
    value X__125 = X__124->f;
    value (*X__126)(value, value*) = X__125;
    value X__127 = X__126(o1, X__124->env);

    struct closure* X__128 = X__127;
    value X__129 = X__128->f;
    value (*X__130)(value, value*) = X__129;
    value X__131 = X__130(o, X__128->env);
    return X__131;
}
```

The driver system

Example:

```
theory int.Int
  prelude "#include \"int.c\""

  syntax function (+) "int_add(%1, %2)"
  syntax function (-) "int_sub(%1, %2)"
  syntax function (*) "int_mul(%1, %2)"
end

module mach.int.Int32
  syntax val (*) "(value)((int32_t)%1 * (int32_t)%2)"
end
```

Conclusion & Future

Things which can still be done:

- Optimisations using regions
- More precise types in the C code (instead of « value »)
- Transformation using intermediate AST in order to simply/improve the code

Beautiful internship, beautiful team. I had a great time here. Special thanks to Jean-Christophe and Andrei.

Questions?

Thank you !