# Adventures in Foreign Function Interfaces

Joel Stanley
jstanley@galois.com

Galois, Inc.

August 19, 2008

©2008 Galois, Inc.

galois

# Overview

- Provide brief description of foreign function interfaces (FFIs)
- Motivate the existence of a Poly/ML $\leftrightarrow$ OCaml conduit
  - Isabelle & Intel's Decision Procedure Toolkit (DPT)
- Demonstrate the FFI usage involved in the integration
- Discuss memory management issues
- Discuss our bridge generation tool
- Q&A

©2008 Galois, Inc.

# Overview of FFIs

"Binding one language to another is a non-trivial task. The binding language needs to understand the calling conventions, type system, data structures, memory allocation mechanisms and linking strategy of the target language, just to get things working.

The task is to carefully align the semantics of both languages, so that both languages can understand the data that passes between them."

– O'Sullivan, Stewart, Goerzen *(RWH)*

©2008 Galois, Inc.

# Overview of FFIs

- Often provided by the RTS and libraries of popular languages
- Examples: Haskell, OCaml, Python, Lua, TCL, etc.
- Typically bridges to C and uses C calling conventions
- Provides extensibility and integration options
  - ...assuming C support is desired
- Used to “import” existing C libraries
- Some automation tools exist (e.g., the Simplified Wrapper and Interface Generator (SWIG)) that support multiple languages

©2008 Galois, Inc.

# Isabelle / DPT Integration

- Isabelle is a well-known generic theorem-proving environment
  - Implemented in Poly/ML
- The Decision Procedure Toolkit (DPT) is an open-source SMT solver from Intel (`http://dpt.sourceforge.org`)
  - Implemented in OCaml
- OCaml and Poly/ML both provide C FFIs
- We want to invoke DPT from an Isabelle tactic
  - SMT solvers can provide counter-examples
  - Useful for formulation of rewrite rules and lemmas
  - ...but this would require a realization of the DPT API in Poly/ML

©2008 Galois, Inc.

# A Tour of the OCaml FFI

- Intended primarily to “import” C libraries into OCaml
- It has somewhat limited support for going the other way
- OCaml’s garbage collector must be dealt with
  - Must deal with relocation as well as reclamation
- The provided FFI is suitable for safe interactions with the OCaml runtime within the C call stack only
  - Suitable for most wrappers, but not good enough for our purposes

©2008 Galois, Inc.

# A Tour of the OCaml FFI (*cont.*)

A trivial binding:

- OCaml:

```
let f x = print_string "ocaml: Inside f, applied to ";
          print_int x;
          print_newline ();;
let _ = Callback.register "test function" f
```

- C:

```
void call_caml_func(int x) {
    value* f = caml_named_value("test function");
    caml_callback(*f, Val_int(x));
}
```

©2008 Galois, Inc.

# A Tour of the OCaml FFI (*cont.*)

A slightly more complex binding:

- OCaml

```
module L = List
class stack_of_ints =
object (self)
  val mutable m_list = ( [] : int list )
  method push x = m_list <- x :: m_list
  method pop    = let result = L.hd m_list in
                  m_list <- L.tl m_list;
                  result
  ...
  end
let push stk x = stk#push x
let peek stk   = stk#peek
let pop stk    = stk#pop
...
let _ =
  (* Callback.register : string -> 'a -> unit *)
  Callback.register "push" push
  Callback.register "pop" pop

  ...
```

©2008 Galois, Inc.

# A Tour of the OCaml FFI (*cont.*)

A slightly more complex binding:

- C

```
#define DECL_CAML_CLOSURE(clos_name)
    static value* clos_name##_closure = 0;
    if (!clos_name##_closure) {
        clos_name##_closure = caml_named_value(#clos_name);
        assert(clos_name##_closure != 0 &&
               "Obtained valid closure value");
    }
#define APPLY1(clos_name, x) (caml_callback(*clos_name##_closure, x))
...
int pop(value stk) {
  CAMLparam1(stk);
  CAMLlocal1(rslt);
  DECL_CAML_CLOSURE(push); /* OCaml RT entry point */
  rslt = APPLY1(pop, stk); /* OCaml RT entry point */
  CAMLreturnT(int, Int_val(rslt));
}
```

- The OCaml FFI macros create a “shadow stack” via a *local roots* data structure...but we require longer lifetimes

©2008 Galois, Inc.

# Pointer Stabilization

- We need stable C pointers to OCaml heap entities, which permit:
  - OCaml heap entities to stay live when the OCaml GC would normally dispose of them
  - Acquisition of persistent references that can be passed as opaque handles to other language runtimes (e.g., Poly/ML)
- We do this via promotion of each OCaml heap entity to the C heap and registration of the resulting C pointer in the *global roots* table
- Referent heap entities can still be relocated by the GC
  - ...must re-dereference after every OCaml RT invocation
  - ...must not cache any dereferenced values
- This approach is only safe as-is because the OCaml RT and GC are single-threaded

©2008 Galois, Inc.

|galois|

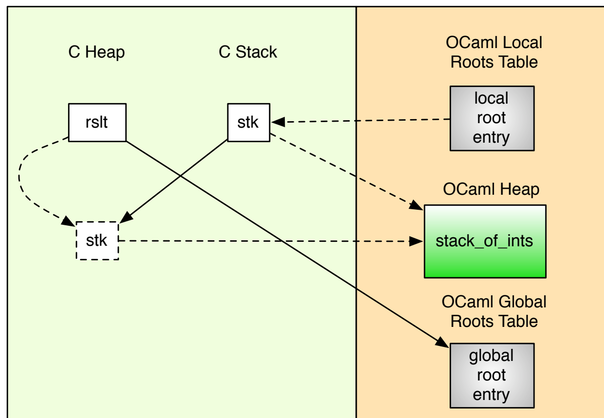
# Pointer Stabilization (*cont.*)

```
value* new_stack () { /* Stable pointer def */
  CAMLparam0();
  CAMLlocal1(stk);
  DECL_CAML_CLOSURE(new_stack);

  stk = APPLY1(new_stack, Val_unit);
  value* rslt = malloc(sizeof(value));
  caml_register_global_root(rslt);
  *rslt = stk;
  CAMLreturnT(value*, rslt);
}

int pop(value* stk_p) { /* Stable pointer use */
  CAMLparam0();
  CAMLparam2(stk, rslt);
  DECL_CAML_CLOSURE(pop);
  stk = *stk_p;
  rslt = APPLY1(pop, stk);
  CAMLreturnT(int, Int_val(rslt));
}
```

©2008 Galois, Inc.

# Pointer Stabilization (*cont.*)

```
stk = APPLY1(new_stack, Val_unit);
value* rslt = malloc(sizeof(value));
caml_register_global_root(rslt);
*rslt = stk;
```




©2008 Galois, Inc.

galois

# The Poly/ML FFI

- Similar in complexity, but has type safety advantages over C (surprise!)
- Interacts with C code via a dynamically loaded library
- Permits abstract type wrappers around C types:

```
(* mkConv : (vol -> 'a) -> ('a -> vol) -> Ctype -> 'a Conv *)
abstype stack_of_ints = stackofints of vol
with val STACK_OF_INTS = mkConv
    stackofints
    (fn (stackofints v) => v)
    (Cpointer Cvoid)
end

(* c2 : string -> 'a Conv * 'b Conv -> 'c Conv -> 'a * 'b -> 'c *)
(* OCAML METHOD: push : stack_of_ints -> int -> unit *)
fun push (stk : stack_of_ints) (x : int) : unit =
  let val ffi_func =
      c2 "push" (STACK_OF_INTS, INT) (VOID);
  in
      ffi_func (stk, x)
  end
```

©2008 Galois, Inc.

# “We Should Automate This”

- Writing per-function bindings by hand is unpleasant and horribly tedious
- Memory management across three language runtimes is error-prone (to say the least!)
- Hard to keep bindings current if the wrapped API changes
- ...but all of the binding code is easy to generate!

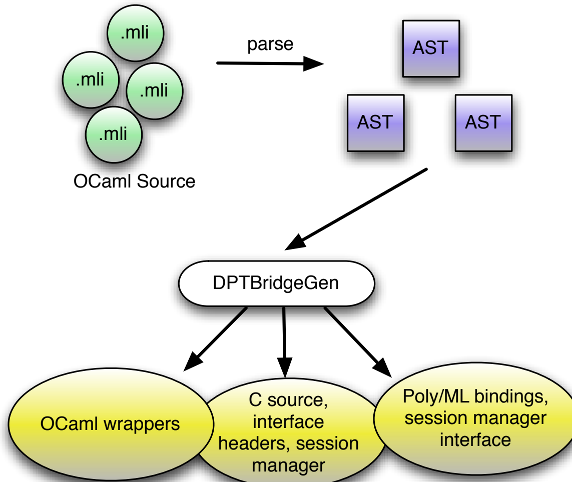
©2008 Galois, Inc.

# FFI Bridge Generation: The Idea

- Devise an AST for representing source language function signatures
  - Must have a way to distinguish between primitive types and “heap entities” (e.g. int vs. stack_of_int)
  - Representation needs to account for nuances of source language, e.g.:
    - Method invocations
    - Module scoping
- Process the AST and generate the bindings and wrappers in each target language
- Provide a lightweight library for memory management, (un)marshaling, etc; we call this the “session manager”

©2008 Galois, Inc.

# DPTBridgeGen Data Flow




©2008 Galois, Inc.

galois

# A Simple AST for OCaml Functions

```
type modpath = string (* format is Mod.SubMod.Etc *)
type ffi_type =
  | Unit | Int | Float | String | Bool
  | Fcn of ffi_type list * ffi_type
  | List of ffi_type
  | HeapEntity of modpath * string
  | SessionMgr

type ocaml_ffi_signature = {
  modpath : modpath;
  name    : string;
  inputs  : ffi_type list;
  ret     : ffi_type
}

type ffi_wrapper =
  | Function of ocaml_ffi_signature
  | MethodInv of ffi_type * ocaml_ffi_signature
  | HeapAlloc of ocaml_ffi_signature
```

©2008 Galois, Inc.

# AST for the Integer Stack

- Stack constructor:

```
HeapAlloc {
  modpath =""; (* visible at top level *)
  name    = "stack_of_ints";
  inputs  = [];
  ret     = HeapEntity("","stack_of_ints")
};
```

- Method invocation:

```
MethodInv (
  HeapEntity("", "stack_of_ints"),
  {
    modpath = ""; (* visible at top level *)
    name    = "push";
    inputs  = [Int];
    ret     = Unit
  }
);
```

©2008 Galois, Inc.

# Transforming a DPT Term Constructor

- The term data type is in the Term module
- Standard term constructors are in the Term_logic.Standard module
- Let's transform the implication constructor:

```
Term_logic.Standard.implication : Term.term -> Term.term -> Term.term
```

- The AST representation of the type signature is:

```
Function {
  modpath = "Term_logic.Standard";
  name    = "implication";
  inputs  = [HeapEntity("Term","term"), HeapEntity("Term","term")];
  ret     = HeapEntity("Term","term")
};
```

©2008 Galois, Inc.

|galois|

# Transforming a DPT Term Constructor: OCaml

```
let term_logic_standard__implication
        (a0 : Term.term)
        (a1 : Term.term)
   : Term.term =
    Term_logic.Standard.implication a0 a1
...
Callback.register "term_logic_standard__implication"
                  term_logic_standard__implication;
```

©2008 Galois, Inc.

|galois|

# Transforming a DPT Term Constructor: C

```
value* dpt_term_logic_standard__implication(
    SessionMgr_dpt* mgr,
    value* a1,
    value* a2)
{
    CAMLparam0();
    CAMLlocal1(_t);
    value* rslt;

    DECL_CAML_CLOSURE(term_logic_standard__implication);

    _t = APPLY2(term_logic_standard__implication, *a1, *a2);

    rslt = register_with_SessionMgr_dpt(mgr, _t);
    CAMLreturnT(value*, rslt);
}
```

©2008 Galois, Inc.

# Transforming a DPT Term Constructor: Poly/ML

```
fun dpt_term_logic_standard__implication
    (mgr : DPT_SessionMgr)
    (a1 : DPT_Term_term)
    (a2 : DPT_Term_term)
 : DPT_Term_term =
    let
        val ffi_func = dpt_c3 "dpt_term_logic_standard__implication"
                (DPT_SESSIONMGR, DPT_TERM_TERM, DPT_TERM_TERM)
                (DPT_TERM_TERM);
    in
        ffi_func (mgr, a1, a2)
    end
```

©2008 Galois, Inc.

# Final Example: DPT (OCaml)

```
module L = Term_logic.Standard
module T = Term
module C = Constant
module S = Symbol

let p_term = T.variable (Variable.named "p")
let q_term = T.variable (Variable.named "q")
let impl   = L.implication p_term q_term
let notq   = L.negation q_term;;

let dpll   = new Dpll.make () in
let theory = new Cc.make () in
let solver = new Solver_dpllt.make dpll theory in

Printf.printf "run solver 1: satisfiable=%B" solver#solve;
solver#add_clause [p_term];
Printf.printf "run solver 2: satisfiable=%B" solver#solve;
solver#add_clause [impl];
Printf.printf "run solver 3: satisfiable=%B" solver#solve;
solver#add_clause [notq];
Printf.printf "run solver 4: satisfiable=%B" solver#solve;
```

©2008 Galois, Inc.

# Final Example: DPT (OCaml vs. Poly/ML)

The name for an OCaml function `A.B.C.name` replaced is by a Poly/ML name `dpt_a_b_c__name`. If an OCaml function does heap allocation, a session manager is also required.

```
let p_term = T.variable (Variable.named "p")
let q_term = T.variable (Variable.named "q")
let impl   = L.implication p_term q_term
let notq   = L.negation q_term;;
```

```
fun dpt_test () =
  let val mgr    = dpt_init_ffi ();
      val pt     = dpt_term__variable mgr (dpt_variable__named mgr "p");
      val qt     = dpt_term__variable mgr (dpt_variable__named mgr "q");
      val impl   = dpt_term_logic_standard__implication mgr pt qt;
      val notq   = dpt_term_logic_standard__negation mgr qt;
```

©2008 Galois, Inc.

galois

# Final Example: DPT (OCaml vs. Poly/ML)

```
let solver = new Solver_dpllt.make dpll theory in
...
solver#add_clause [p_term];
solver#add_clause [impl];
...
```

```
...
val solver = dpt_new_solver_dpllt__make mgr dpll theory
val mtl    = (fn ts => marshal_ocaml_list mgr DPT_TERM_TERM ts)
(* marshal_ocaml_list : DPT_SessionMgr ->
                        'a Conv -> 'a list ->
                        'a ocaml_list *)

...
dpt_solver_api_solver__add_clause solv (mtl [p_term]);
dpt_solver_api_solver__add_clause solv (mtl [impl]);
```

©2008 Galois, Inc.

|galois|

# Final Example: DPT (Poly/ML)

```
fun dpt_test () =
  let val mgr    = dpt_init_ffi ();
      val pt     = dpt_term__variable mgr (dpt_variable__named mgr "p");
      val qt     = dpt_term__variable mgr (dpt_variable__named mgr "q");
      val impl   = dpt_term_logic_standard__implication mgr pt qt;
      val notq   = dpt_term_logic_standard__negation mgr qt;

      val dpll   = dpt_new_dpll__make mgr;
      val theory = dpt_new_cc__make mgr;
      val solver = dpt_new_solver_dpllt__make mgr dpll theory;

      fun run s n =
        let val sat = dpt_solver_api_solver__solve s in
          print ("run " ^ Int.toString n ^ " : satisfiable=" ^
                 (Bool.toString sat))
        end;

      (* marshals a list of terms to an ocaml list *)
      val mtl = (fn ts => marshal_ocaml_list mgr DPT_TERM_TERM ts)
    in
```

©2008 Galois, Inc.

# Final Example: DPT (Poly/ML)

```
    run solv 1;

    (* dpt_solver_api_solver__add_clause :
       DPT_Solver_api_solver -> DPT_Term_term ocaml_list -> unit *)

    dpt_solver_api_solver__add_clause solv (mtl [pt]);
    run solv 2;

    dpt_solver_api_solver__add_clause solv (mtl [impl]);
    run solv 3;

    dpt_solver_api_solver__add_clause solv (mtl [notq]);
    run solv 4;

    dispose_SessionMgr_dpt mgr
end
```

©2008 Galois, Inc.

# Conclusions

- Existing automated binding generators aren't bidirectional enough, nor do they facilitate "language hopping"
- Better integration tools are needed for applications written in different FPLs
- DPTBridgeGen available online (`http://dpt.sourceforge.net`)
- Alpha version - many improvements could be made
  - Still need an interface parser
  - Make it more general purpose
  - Mirror OCaml module structure (vs. flattened naming)
  - Leverage weak pointer mechanisms in Poly/ML
  - Perform quantitative performance analysis
- Questions?

©2008 Galois, Inc.