Type Inference and Polymorphism for C

Jiří Klepl September 2020

The C Language

- Valued for its simplicity and code transparency
- One of the most influential languages in systems development
- The language of kernel development
- No support for polymorphism
- Generic constructions must be simulated (macros, void*)

Generic Programming in C (example from BSD: queue.h)

```
SLIST HEAD (slisthead, entry) head = SLIST HEAD INITIALIZER (head);
struct entry {
 . . .
 SLIST ENTRY(entry) entries: /* Singly-linked List. */
 . . .
} *n1:
n1 = malloc(sizeof(struct entry));
SLIST INSERT HEAD (&head, n1, entries); /* Insert at the head. */
```

Generic Programming in C (example from BSD: queue.h)

```
SLIST HEAD (slisthead, entry) head = SLIST HEAD INITIALIZER(head);
struct entry {
 . . .
 SLIST ENTRY(entry) entries: /* Singly-linked List. */
 . . .
} *n1:
n1 = malloc(sizeof(struct entry));
SLIST INSERT HEAD (&head, n1, entries); /* Insert at the head. */
```

We have to provide type-relevant information (this could be inferred).

Example from BSD: queue.h

```
SLIST HEAD (int slist, int entry) head = SLIST HEAD INITIALIZER (head)
struct int_entry {
    SLIST_ENTRY(int_entry) entries;
    int value:
} n1;
n1 = malloc(sizeof(struct int_entry));
n1 \rightarrow value = 1:
SLIST INSERT HEAD (&head, n1, entries);
```

Our approach

- Minimal language extension
- Hindley-Milner-style polymorphism and inference
- Type classes for overloading

Our approach

- Minimal language extension
- Hindley-Milner-style polymorphism and inference
- · Type classes for overloading

```
a head = empty(); // the type is inferred
insert_head(&head, (int)1);
```

Polymorphic list implementation

```
<a : b ~ struct list : <a> > // type variables and derived types
struct list {
    b *next;
    a value;
};
<a : b ~ struct list : <a> >
void insert_head(b **head, a value) {
    b *node = new(); // required type for new() is inferred
    node->next = *head:
    node->value = value:
    *head = node;
```

Simple helper functions

```
<a, b : c ~ b : <a> >
c *empty() {
    return (c *)NULL;
}

<a> a *new() {
    return (a *)malloc(sizeof(a));
}
```

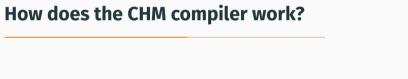
Results

- Proof-of-concept compiler to C
- Functionality partially overlaps with C++ templates
- Practical programs that work with CHM
- Identified future challenges:
 - implicit conversions (subtyping, constantness, ...)
 - \bullet type of struct accessors

Example output

Compiler output: normal C code, can be compiled with gcc

```
typedef struct TA7TC4list3int TA7TC4list3int;
struct TA7TC4list3int { TA7TC4list3int * next: int value: }:
TA7TC4list3int * __new_TF018TP14TA7TC4list3int()
f return (TA7TC4list3int *) malloc(sizeof(TA7TC4list3int)): }
void insert head TF25TP18TP14TA7TC4list3intint7TC4Void(TA7TC4list3int * * head.
                                                        int value)
   TATTC4list3int * node = new TF018TP14TA7TC4list3int():
   node->next = *head:
   node->value = value:
   *head = node: }
TA7TC4list3int * nullptr TF018TP14TA7TC4list3int()
{ return (TA7TC4list3int *) (void *) 0: }
TP14TA7TC4list3int head = nullptr TF018TP14TA7TC4list3int():
insert head TF25TP18TP14TA7TC4list3intint7TC4Void(&head. (int) 1):
```



Implementation overview

- Parsing (uses modified Language.C)
- Conversion to λ_C
- Type Inference (uses modified THIH)
- Monomorphization
 - Instantiation
 - Mangling ('TP', 'TF', structs, primitive types)
- Code output
- Compiling (uses gcc)

Representing C types using $\lambda_{\mathcal{C}}$ conversion

- Primitive types and structs (unions) modeled directly:
 - $[[t]]_{\lambda_c} = t$
- · Pointers and functions:
 - $[[t *]]_{\lambda_c} = (*) [[t]]_{\lambda_c}$
 - $[[t(p_1,\ldots p_n)]]_{\lambda_c}=([[p_1]]_{\lambda_c},\ldots,[[p_n]]_{\lambda_c})\rightarrow [[t]]_{\lambda_c}$

Representing C types using $\lambda_{\mathcal{C}}$ conversion

- Primitive types and structs (unions) modeled directly:
 - $[[t]]_{\lambda_c} = t$
- · Pointers and functions:
 - $[[t *]]_{\lambda_c} = (*)[[t]]_{\lambda_c}$
 - $[[t(p_1,\ldots p_n)]]_{\lambda_c} = ([[p_1]]_{\lambda_c},\ldots,[[p_n]]_{\lambda_c}) \rightarrow [[t]]_{\lambda_c}$

Representing type dependencies inside C constructs using $\lambda_{\mathcal{C}}$ conversion

- An example for C addition (functions and initializations more complex)
 - $[[a + b]]_{\lambda_c} = (+) [[a]]_{\lambda_c} [[b]]_{\lambda_c}$
 - (+): $\forall \alpha. Num(\alpha) \Rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$
- Challenges
 - No implicit casts (we can use explicit casts)
 - · Const semantics with the C syntax do not fit type inference rules
 - C dot operator with a given field syntactically equivalent to a function
 - To preserve syntax we have to make limitations on field types

Extending the language (polymorphic functions and structures)

```
<a>>
a copy(a value) { return value; }
<a>>
struct container { a value; };
<a : b ~ container : <a> >
b wrap(a value) {
    b wrapper;
    wrapper.value = value;
    return wrapper;
```

User-defined type classes and instances

```
<a : Div<a> >
class SafeDiv <a> {
a safe_div(a left,
           a right);
instance SafeDiv<float> {
float safe_div(float left,
               float right) {
    return left / right;
```

```
instance SafeDiv<int> {
int safe_div(int left,
             int right) {
    if (right != 0)
        return left / right;
    else return 0;
```

