μJuniper: Soundness for Array Capacities

Caleb Helbling - Purdue University CS 592 APL - Spring 2023

Juniper

- A functional reactive programming language for the Arduino
- Stack based memory management, minimal heap allocation
- Arrays are stored on the stack, so array sizes must be known at compile time
- Array size (capacities) are part of the type of an array: **uint8[n]**
- Generic functions can be polymorphic over type level integers
- Types and type aliases can be polymorphic over types and type level integers

Example: map for lists

```
alias list<'a; n> = { data : 'a[n]; length : uint32 }
```

```
fun map<'a,'b,'closure;n>(f : ('closure)('a) -> 'b, lst : list<'a; n>) : list<'b; n> =
    let mutable ret = array 'b[n] end;
    for i : uint32 in 0u32 to lst.length - 1u32 do (
        set ret[i] = f(lst.data[i]);
        ()
    ) end;
    {data=ret; length=lst.length}
)
```

Safe String Concatenation

```
alias charlist<;n> = list<uint8;n+1>
```

- Strings in Juniper are lists of ASCII characters
- Strings must have a terminating null character
  - Helps with C/C++ compatibility
- safeConcat function is part of the Juniper standard library
- **∀aCap,bCap.**`(charlist<;aCap>, charlist<;bCap>) -> charlist<;aCap+bCap>`

Dependent Types in Juniper

- Type level arithmetic operations on capacities supported (+, -, *, /)
- Type level integers **can** be demoted to values
- Integer values **cannot** be promoted to types
- Juniper has type inference
  - Solve a system of constraints
  - Constraints can contain arbitrary arithmetic operations
  - Solver needs to know algebra. Juniper uses a CAS (computer algebra system) to do inference and verify type level arithmetic
  - Example: int[m+n] == int[n+m]

µJuniper – Juniper Formalization in Coq

- Subset of Juniper
- No loops, no mutation
- Supports tuples, arrays, value level integers, value level arithmetic
- Array literals, constant arrays, array get, array set, mapi
- Polymorphism handled using Gallina functions
- Type level arithmetic done in Gallina
- Type aliases are Gallina functions
- Type level operations are Gallina → we can use **`lia`** to resolve arithmetic solving

String Concatenation

```
Definition junList (elem : ty) (capacity : nat) := <| Nat * (elem[capacity]) |>.
Definition junStr (capacity : nat) := junList <| Nat |> (capacity + 1).

Definition concatStr (capX : nat) (capY : nat) :=
  <{\ x : <<junStr capX>>, \y : <<junStr capY>>,
    let temp : <|Nat[<<capX + (capY + 1)>>]|> = <<tm_array_con (capX + capY + 1) <| Nat |> <{n 0}>>> in
    let xLen : <|Nat|> = fst x in
    let yLen : <|Nat|> = fst y in
    let xArr : <|Nat[<<capX + 1>>]|> = snd x in
    let yArr : <|Nat[<<capY + 1>>]|> = snd y in
    <
      (xLen + yLen) - n 1,
      <<tm_mapi
        <{
        (\idx : Nat, \c : Nat,
          if idx < (xLen - n 1) then
            <<tm_array_get xArr idx <{n 0}>>>
          else
            <<tm_array_get yArr <{(idx + n 1) - xLen}> <{n 0}>>>)
        }>
        <{ temp }>
        <| Nat |>
      >>
    >
    }>.
```

Type Checker

```
Theorem string_concat_type :
  forall capX capY,
  empty |- <<concatStr capX capY>> \in
  (<<junStr capX>> -> (<<junStr capY>> -> <<junStr (capX + capY)>>)).
```

$$\frac{\Gamma \vdash t_1 : T_2 \to T_1 \qquad \Gamma \vdash t_2 : T_3 \qquad T_2 = T_3}{\Gamma \vdash t_1\ t_2 : T_1}\text{TAPP}$$

```
| T_App :
  forall Gamma t1 t2 T1 T2 T3,
  Gamma |- t1 \in (T2 -> T1) ->
  Gamma |- t2 \in T3 ->
  T2 = T3 ->
  Gamma |- <{t1 t2}> \in T1
```

Main Result

- Proofs that µJuniper is sound (progress & preservation)
  - The world is now a slightly safer place
- Problem: Type level arithmetic and type polymorphism is “outside” of µJuniper
  - Prove that the CAS system is correct?
- Proving type inference is correct would be a lot more work
- Clear connection between CAS used by Juniper and lia used by Coq