Compilation 2024

Dolphin: phase 5

Amin Timany
timany@cs.au.dk

Dolphin: phase 5

- So far:
  - Phase 1: Basic expressions and statements (based on ASTs)
  - Phase 2: Loops (based on ASTs)
  - Phase 3: Frontend, *i.e.*, lexing and parsing
  - Phase 4: Functions and comma expressions

- Today
  - Phase 5:
    - Aggregate types: strings, arrays, and records

Strings: Lexing and Parsing

- Strings are primitive types
  - In Dolphin string literals consist of **ascii** characters
  - We support the usual escape characters (\n, \t, …)
    - Example: “Hello World!\n”
    - Use OCaml’s `Scanf.unescaped` function in **lexer**

- Lexer and Parser need to support string literals
- `length_of` keyword (**not a library function**)
  - Used like a function, *e.g.*, `length_of(s)`
  - Use a separate entry for `length_of` in ASTs

Strings: Semantic Analysis

- A new type: `string`
- Comparison operators support strings:
  - `==` and `!=` (support any non-void type)
  - <, <=, >, >= (support only integers and strings)
  - All operators above compare **contents** of strings
    - String comparison should be implemented in `runtime.c`
- `length_of` keyword
  - Applies to both strings and arrays

Strings are Immutable

- The only way to create strings is through
  - String literals
  - Standard library functions (guarantee immutability):
    - `string substring(s : string, start : int, len : int)`
    - `string string_concat(s1 : string, s2 : string)`
    - `int string_to_int(s : string)`
    - `string int_to_string(i : int)`
    - ect.
- In other words, all functions above returning strings return a newly created string. They do not modify their argument!
- Attention: strings cannot be `nil`!

Arrays: Lexing and Parsing

- The type `[t]` is the type of arrays with `t` (non-`void`) elements
- New expression forms:
  - Creating a new array of length `10` with elements of type `t`:
    - `new t[10]`
    - No explicit initialization; initialized to default values
    - `nil` (null pointer)
- New lval form:
  - Accessing `i`th element of the array `a` (for reading or writing):
    - `a[i]`
- `length_of` keyword (**not a library function**)
  - Used like a function, *e.g.*, `length_of(s)`
  - Use a separate entry for `length_of` in ASTs
  - Same as strings, no new work for arrays compared to strings

Arrays: Semantic Analysis

- Recall: comparison operations == and != are support all non-void types
  - We compare references (pointers)
- The expression `new t[10]` has type `[t]` (`t` cannot be `void`)
- The lval `a[i]` has type `t` if `a` has type `[t]`
- `nil` needs special treatment

Records: Lexing and Parsing

- A program consists of
  - A number of record type declarations
  - A number of functions
- New expression forms:
  - We can declare new record types:
    - `record list {head : int; tail : list;}`
  - Types now include identifiers (names of records)
  - We can create a new record instance:
    - `new list {head = 10; tail = nil;}`
  - `nil` (null pointer) **same as arrays**
- New lval form:
  - Accessing record fields (for reading or writing)
    - `a.head`

Records: Semantic Analysis for Declaring New Record Types

- How do we check that the following is a valid record type declaration?
  - `record list {head : int; tail : list;}`

Records: Semantic Analysis for Declaring New Record Types

- How do we check that the following is a valid record type declaration?
  - `record list {head : int; tail : list;}`
- How about the following declarations?
  - `record A {a : A; b : B;}`
  - `record B {a : A; b : B;}`

Records: Semantic Analysis for Declaring New Record Types

- How do we check that the following is a valid record type declaration?
  - `record list {head : int; tail : list;}`
- How about the following declarations?
  - `record A {a : A; b : B;}`
  - `record B {a : A; b : B;}`
- The above is also valid if other record/function declarations come between them.

Records: Semantic Analysis for Declaring New Record Types

- How do we check that the following is a valid record type declaration?
  - `record list {head : int; tail : list;}`
- How about the following declarations?
  - `record A {a : A; b : B;}`
  - `record B {a : A; b : B;}`
- The above is also valid if other record/function declarations come between them.
- Is the following valid?
  - `record A {a : A; a : A;}`

Records: Semantic Analysis for Declaring New Record Types

- How do we check that the following is a valid record type declaration?
  - `record list {head : int; tail : list;}`
- How about the following declarations?
  - `record A {a : A; b : B;}`
  - `record B {a : A; b : B;}`
- The above is also valid if other record/function declarations come between them.
- Is the following valid?
  - `record A {a : A; a : A;}`
- How about the following declarations?
  - `record A {a : A; b : B;}`
  - `record A {a : A; c : B;}`

Records: Semantic Analysis for Declaring New Record Types

- Add a mapping for records to the environment:
  - maps identifiers to record bodies (list of pairs of field names and their types)
  - Record names are disjoint from function/variable names
    - It is valid to have both a function and a record named `f`

- We check record type declarations as follows:
  - **First**: Add all record names for the entire program to the **environment** without their bodies
    - Record type names must be unique in the program
  - All fields should have valid non-`void` types
  - No field duplication in the same record
  - **Needs to be done before semantic analysis of functions**

Records: Semantic Analysis for Creating New Record Instances

- How do we check that the following record creation?
  - `new list {head = 10; tail = nil;}`
- What should we pay attention to?

Records: Semantic Analysis for Creating New Record Instances

- How do we check that the following record creation?
  - `new list {head = 10; tail = nil;}`
- What should we pay attention to?
  - No field redefinitions
  - **All** fields (of `list` in this example) must be present
    - They must all have the correct type
  - **Only** fields (of `list` in this example) must be present

Records: Semantic Analysis for Field Access

- How do we check that the following record creation?
  - `a.f`
  - We infer the type of `a`
  - It must be a record type `r`, otherwise, we issue an error
  - We lookup declaration of record `r` in the environment
  - The field `f` of some type `t` must be present in the declaration of `r`
  - `a.f` is an lval of type `t`

Records: Semantic Analysis

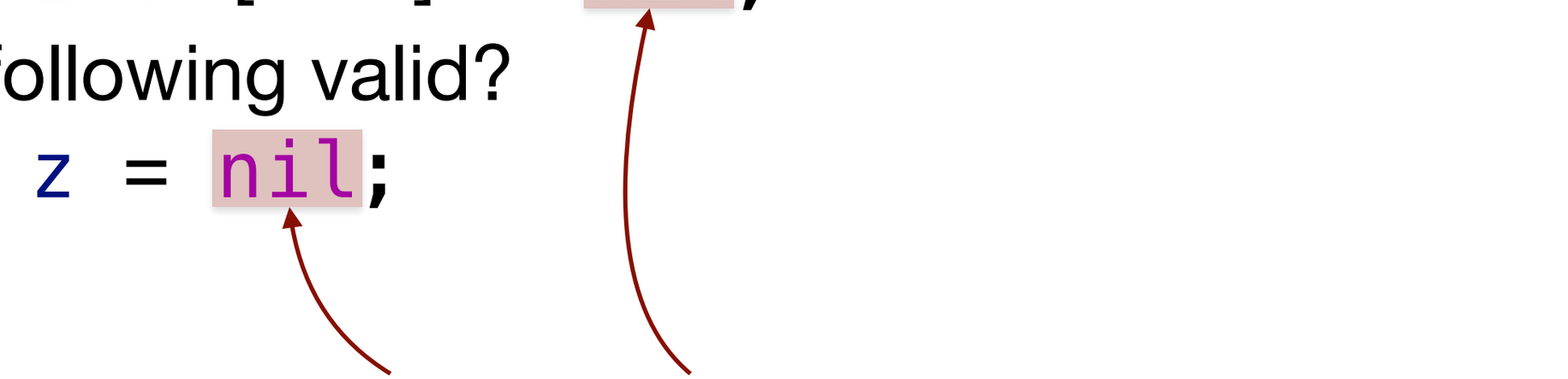
- Recall: comparison operations == and != are support all non-void types
  - We compare references (pointers)
- Record declarations (see earlier slide)
- Record instance creation (see earlier slide)
- Field access (see earlier slide)
- `nil` needs special treatment

Why do we need `nil`?

- Hint: think about recursive records

Semantic Analysis of `nil`

- `nil` complicates type checking/inference
- How do we do semantic analysis for the following?
  - `var z : [int] = nil;`
- Is the following valid?
  - `var z = nil;`



Semantic Analysis of `nil`

- We introduce a new kind of type (undetermined)
  - Before:

```
type typ =
| Void of {loc : Loc.location}
| Int of {loc : Loc.location}
| Bool of {loc : Loc.location}
| ErrorType
```

  - After (suggestion):

```
type typ =
| Void of {loc : Loc.location}
| Int of {loc : Loc.location}
| Bool of {loc : Loc.location}
| Str of {loc : Loc.location}
| Byte of {loc : Loc.location}
| Array of {typ : typ; loc : Loc.location}
| Record of {recordname : recordname}
| ErrorType

type gentyp =
| Determined of typ
| Undetermined of int
```

Each `nil` encountered gets a fresh id.

Semantic Analysis of `nil`

- In typed AST, all expressions (old and new) have type `typ` except for `nil`

```
type expr =
|…
| Nil of {typ : gentyp; loc : Loc.location}
|…
```

Semantic Analysis of `nil`

- Type inference
  - Takes an expression
  - Returns a pair of a typed expression and a `gentyp`
  - Implemented (**almost**) as before and extended as explained earlier for strings, records, and arrays
  - When encountering `nil`, we produce a typed `nil` with type `Undetermined id` for some fresh `id`

Semantic Analysis of `nil`

- Type checking
  - Takes an expression and a `typ`
  - returns a typed expression
  - As before, implemented by running type inference
  - With the following exception
    - To see if an expression `e` has type `t`:
      - **Before**: infer the type `s` of `e` to obtain a typed version of `e`
        - If `s` is **equal** `t`, then return typed version of `e`
        - Otherwise issue an error
      - **Now**: infer the type `s` of `e` to obtain a typed version of `e`
        - If `s` is **equal** `t`, then return typed version of `e`
        - If `s` is `Undetermined id` and `t` is an array or record
          - Substitute `Undetermined id` with `t` in the typed version of `e` and return it
        - Otherwise: issue error

Semantic Analysis of `nil`

- How do we do semantic analysis for the following?
  - `var z : [int] = nil;`
- Or rather, for an arbitrary expression `e`
  - `var z : [int] = e;`

- Run inference on `nil` to obtain a typed version of `e`
- Returns a typed `nil` with type `Undetermined id`
- Substitute `Undetermined id` with [int] in the typed version of `e` and use that to construct the typed version of the entire command

Semantic Analysis of `nil`

- Question: does this mean that after semantic analysis there are no unresolved `nil`s in the program. That is, when we perform semantic analysis, does the end result have a `nil` with `Undetermined id` type?

- Hint: think about all places where we run inference that is not done via type checking …

Semantic Analysis of `nil`

- Let us revisit type inference
- How do we infer the type of
  - `if(nil == nil) return nil;`
  - `if(x != nil) return x;`
  - `var x = (nil, 3); // (comma expression)`
- For (in)equality, we need to **unify** the types of two sides
  - Unification:
    - If the two types are equal, do nothing
    - If one side is `Undetermined id` and the other is a record or array type, or another undetermined type `t`
      - Substitute `Undetermined id` with the `t`
    - Otherwise, issue error
- We **ignore** cases where the expression on left of comma is undetermined

LLVM -- for translating aggregates

- Structure types
- Fixed-size arrays
- Named types
- Global Variables
- String variables
- Casting
- Pointer to integer conversion
- Computing physical size of type
- getelementptr (Gep)