Compiler Construction - Supervision 3

Nandor Licker

October 2019

1 Past Papers

- 2009 Paper 5 Question 2
- 2011 Paper 3 Question 4

2 Practical

2.1 Enum Type Declarations

Now that the compiler can parse some type declarations, it is time to add more complex data types, namely algebraic data types. The first step is adding syntax to define new types and constructors:

```
enum List<T> {
    Nil();
    Cons(T, List<T>)
};
```

The data constructors take a number of arguments, whose types are specified in the syntax. You should reuse and extend the type parsing mechanisms implemented in the previous exercise. At this point, it might be worthwhile introducing a restriction on identifiers, similar to Haskell: uppercase identifiers denote types and constructors, lowercase identifiers denote functions and outher bound variables.

2.2 Constructors

- Constructors need to be type checked and types for constructors must be built. In a first pass, you must identify all types in the program, ensuring names and the type variables are not duplicated. A data structure should save type names, along with the number of type parameters accepted by the data structure. Note that types are polymorphic: the list can store any type inside of it.
- The enumeration of types `ty` must be extended with `TyEnum of string * ty array`. The string referes to the type name, while the array represents all the types through which the polymorphic type is instantiated with. For example, the type of a list of integers is `TyEnum [| TyInt |]`.
- Each constructor needs to be assigned a type during the type checking phase. For example, the constructors in the previous ADT are typed as:
  - $Nil :: \forall\alpha.() \to List(\alpha)$

    `TyArr [| |] (TyEnum [| TyAbs 0 |])`
  - $Cons(T, List < T >) :: \forall\alpha.(\alpha, List(\alpha)) \to List(\alpha)$

    `TyArr [| TyAbs 0, TyEnum [| TyAbs 0 |] |] (TyEnum [| TyAbs 0 |])`

  Types must be checked for each constructor and saved in a context. During this phase, errors must also be emitted if constructor names are not unique. Evaluate each constructor in the context of its type: create a mapping from parameter names to `TyVar`s, then construct the arrow types (`TyArr`) in that context, as shown in the prior examples. These types will not be polymorphic: the `generalise` method must be used to take the free type variables and turn them into `TyAbs` polymorphic parameters.

- Constructors should be assigned numeric tags in order to distinguish the variants: for example, `Nil` and all objects built using Nil should have tag 0, while all objects build using the `Cons` constructor should have tag 1. This mapping should be saved - tags can be reused between different enumerations.

2.3 Constructor Calls

Objects should be allocated using the following syntax:

```
func make_list(n) {
    return Cons(n, Cons(n, Cons(n, Nil())));
}
```

Depending on your choice of naming, at this stage the AST should either distinguish constructors and function calls in separate nodes or provide a way to distinguish call targets: similarly to function calls, the arguments should be unified with the constructor type, loaded from the previously built mapping and instantiated: the method replaces all `TyAbs` ocurrences with fresh type variables.

Polymorphism relies on the instantiation method. When a name with a polymorphic type is used, the free parameters are replaced with fresh variables, which are then unified with a specific type, such as `TyInt` in the case of `List(1, Nil)`. When the list is used again, with lists of lists for example, `TyAbs` is replaced with another fresh variable, which is unified with the inner list type. This allows `List` to be generic.

Your task is to typecheck these constructor calls, then lower them to IR and assembly. In the IR lowering, you will probably need a `Cons` instruction which takes a tag an the arity of the constructor. When lowered to assembly, it should allocate a block similarly to closures, setting individual fields to values popped from the stack. Unlike closures, the header of the allocated heap block should also encode the constructor.

2.4 Pattern Matching

Variant types are not particularly useful without a way to peek inside their contents. In this task, you are to implement pattern matching: First, a very limited mechanism, then a more general method which supports deeply nesting patterns, as implemented in OCaml. To start out, add a match statement:

```
match list {
    List(x, xs) => {
        ...
    }
    Nil => {
        ...
    }
}
```

The patterns should be limited - only constructor names should be allowed, with fields captured by variable names. The grammar should prohibit nested patterns at this point. Extend the parser, type checker and code generator, choosing a suitable instruction for the tag selection. What happens if a pattern is not exhaustive? Clearly describe your strategy for dealing with that scenario.

If you wish, implement nested patterns as well, using the method outlined in the cited paper [1].

References

[1] Lennart Augustsson. Compiling pattern matching. In *Conference on Functional Programming Languages and Computer Architecture*, pages 368–381. Springer, 1985.