Sophia Compiler

Trip over the functional compiler

General Pipeline

---

1. File -> Source code
2. Lexer -> Token stream
3. Parser -> AST
4. Typechecker -> Typed AST
5. IR Compiler -> Intermediate representation
6. Final compiler -> Bytecode
   + optimizations in between

Compilation – source

“

```
function f(n, acc) =
  if(n < 2) acc
  else f(n - 1, acc * n)
```

“

Compilation – token stream

```
---
[ {keyword, function}, {id, “f”}, paren_open, {id, “n”}, comma
, {id, “acc”}, paren_close, {operator, ‘=’}, {keyword, ‘if’}, paren_open
, {id, “n”}, {operator, ‘<’}, {int, 2}, paren_close, vclose, vsemi,
, {keyword, else}, {id, “f”}, paren_open, {id, “n”}, {operator, ‘-’}
, {int, 1}, comma, {id, “acc”}, {operator, ‘*’}, {id, “n”},
, paren_close, vclose, vclose
]
```

Compilation – AST

```
---
{ function,
, “f”,                                                  % name
, [{id, “n”}, {id, “acc”}]                              % args
, { if, {ap, {op, ‘<’}, [{id, “n”}, {int, 2}]}    % body
  , {id, “acc”}
  , {ap, {id, “f”}, [ {ap, {op, ‘-’}, [{id, “n”}, {int 1}]}
                    , {ap, {op, ‘*’}, [{id, “n”}, {id, “acc”}]}]}]}
  }}
```

Compilation – typed AST

———

```
{ function,
, “f”,                                                  % name
, {fun, [int, int], int}                                % type
, [{id, “n”, int}, {id, “acc”, int}]                    % args
, { if, int, {ap, {op, ‘<’, {fun, [int, int], bool}} % body
           , [{id, “n”, int}, {int, 2, int}], bool}
  , {id, “acc”, int}
  , {ap, {id, “f”, {fun, [int, int], int}}, ...
```

Compilation – intermediate representation (example)

```
{ def, “f”, {lambda, 2,
  {case, {apply, int_lesser, [$0, 2]}
  [ {true, $1}
  , {false, {apply_tail, {fun, “f”},
      [{apply, int_minus, [$0, 1]}, {apply, int_mult, [$0, $1}]}}
  ]}
}}
```

Compilation – bytecode

———

```
f0:
  LT stack arg0 2
  JUMPIF stack f2
f1:
  MUL arg1 arg1 arg0
  DEC arg0
  JUMP f0
```

```
f2:
  RET arg1
```

Sophia – syntax

Files:

- aeso_syntax - type definitions: AST, Sophia types, annotations, etc
- aeso_syntax_utils - general usage utilities that work on AST

Sophia – lexer

Files: aeso_scan, aeso_scan_lib, aeso_parse_lib

vsemi, vclose - virtual semicolon, virtual block end

Sophia – parser

Files: aeso_parser, aeso_parse_lib

Beside creating raw AST, the parser takes care of include inlining, first preprocessing (ie. collapsing ‘if’ sequences)

Annotations: store information about token localization, files and some other metadata

Sophia – typechecker

Files: aeso_ast_infer_types

- Infer the types
- Validate types
- Validate constraints
- Decorate AST with type annotations
- Check the overall program structure
- Forbid some illegal cases

Type system is a superset of classic Hindley-Milner.

Sophia – typechecker

Phases:

- Typedef kindcheck
- Function binding
- SCC dependency analysis
- Type inference per SCC block
- (optional) type unfolding

Sophia – typechecker

Upgraded Hindley-Milner:

- Parametric polymorphism
- Mutual recursion (toplevel only)
- No ad-hoc polymorphism
- Inference of records
- Only first rank types

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \text{[Var]}$$

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'} \quad \text{[App]}$$

$$\frac{\Gamma,\ x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\ x\ .\ e : \tau \to \tau'} \quad \text{[Abs]}$$

$$\frac{\Gamma \vdash e_0 : \sigma \qquad \Gamma,\ x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1 : \tau} \quad \text{[Let]}$$

$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma} \quad \text{[Inst]}$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin \mathrm{free}(\Gamma)}{\Gamma \vdash e : \forall\ \alpha\ .\ \sigma} \quad \text{[Gen]}$$

Sophia – typechecker – algorithm

Type inference is done per contract independently.

After typedef sanity check the functions are grouped in the SCC graph and topologically sorted.

The groups are typechecked in the ascending (reg. the topological ordering) order.

The type env is being updated along the way.

Sophia – typechecker – function inference

Inference of group of functions (`infer_letrec`):

1. Assign a wobbly typevar to each function definition
2. Take the next function, infer its type
3. Unify with the global env (update the substitution)
4. If not finished, goto 2

Sophia – typechecker – body inference

Inference of expressions/blocks (`infer_expr`, `infer_block`):

- Typed expressions: infer the type of expression and unify with the declared type
- Lambdas: assign type vars for arguments and typecheck body
- Records: accumulate field constraints for variables (ie. “has field `kek`”). Assert nonambiguity.
- Bits: constraints, as above
- Application: unify function with a functional type
- Other cases should be easy/trivial

Sophia – typechecker – unification

MGU - most general unifier. Minimal substitution that unifies two types:

- `mgu(uvar(x), T) = {x := T}`
- `mgu(T, uvar(x)) = {x := T}`
- `mgu(F1 -> A1, F2 -> A2) = mgu(F1, F2) + mgu(A1, A2)`
- `mgu(tvar(x), tvar(y)) = if(x == y) {} else error`

uvars (“wobbly” ones) can be substituted. tvars (“rigid”) cannot. Implemented in the `unify` function.

Sophia – typechecker – occurs check

Consider function f(x) = f(f)

1. f : ‘a => ‘b
2. mgu(‘a => ‘b, ‘a)
3. ‘a ~ ‘a => ‘b
4. (((... => ‘b) => ‘b) => ‘b) => ‘b ???

We forbid a situation where we assign a type with a free variable to this variable.

Sophia – typechecker – type schemes

Consider function f(x) = (x, 1).

Officially f : (‘a) => ‘a * int

But actually f : forall ‘a. (‘a) => ‘a * int

“I can be a type `(‘a) => ‘a * int` for whatever ``’a` you want”.

If the `forall` could be nested, we would talk about higher rank types.

Sophia – typechecker – instantiation

Instantiation is the process of making the wobbly variables rigid to protect the generality.

Consider function f(x : ‘a) = x + x

Without the type annotation, ‘a would be unified with int. In this case, this would lead to loss of generality.

While entering the local context we must instantiate the quantified typevars.

Sophia – typechecker – error handling

- Collected during runtime
- Stored in ETS
- Destroyed and reported in the end of typecheck process

Sophia – typechecker – constraints

- Stored in ETS
- Created during inference
- Solved in the end
- Ambiguity is treated as an error

Sophia – compiler

Two backends:

- AEVM
  - IR: icode
  - Stack machine similar to EVM
  - Deprecated
- FATE
  - IR: fcode
  - Stack/register machine with types and some Erlang abstractions
  - The recent one

Sophia – fcode

Files: aeso_ast_to_fcode

- Simplified functional programming language
- All of syntactic sugar is unfolded
- Optimizations:
  - Dead code elimination
  - Unused variable elimination
  - Case specialization
  - Constant propagation
  - Inlining
  - Let floating

Sophia – FATE

Files: aeso_fcode_to_fate, repository aebytecode

- High level (as for VM)
- Typed
- Supports maps, lists, tuples natively
- Gas calculation per instruction
- Memory layout:
  - Chain/call data
  - State
  - Event stream
  - Execution stack (push/pops on calls)
  - Accumulator stack (push/pops on instructions)
  - Register storage

Sophia – FATE

Optimizations:

- Tail recursion realization
- Inlining
- Removal of dead assignments
- Dead code elimination
- Jump optimizations

Questions?

– – –

. . .