CompSpec

abhishekkmr18

April 2018

1 Language Specifications

1.1 Lexical Specifications

- Identifier → [a-z]([A-Z] U [a-z])*
- Constructor → [A-z]([A-Z] U [a-z])*
- Lambda → ?
- Dereference → |

All other lexical tokens like =, ¡, (,), , , have the standard meaning.

1.2 Sytax Specifications

- Program → (exports) type_declarations computationalUnit
- exports → Identifier : Type, exports
- type_declarations → Identifier = Constructor types_list | Constructor types_list ...
- computationalUnit → term | let x = term in computationalUnit
- term →
  - ?x.term (Lambda Abstraction)
  - term1 term2 (Lambda Application)
  - Ref term (Reference to a term)
  - | term
  - = term1 term2 (Equality check for two terms)
  - n (metavariable for integers)
  - term : type
  - term + term

  - (term, term) (Pairs)
  - Unit

- type →
  - x: type — phi
  - Ref type
  - type -¿ type
  - (type, type)
  - Un

2 Implementation Specifications

Source language of the compiler is described in detail in the previous section and target language is Web Assembly [?].

Based on the type system described in Chapter 3, we have a source language with **Low Values** and **High Values**, we need to come up with a **Compilation Algorithm** which preserves this distinction in source language throughout the compilation into the target language.

With the above goal we propose the following compiler specifications and algorithm to compile the lambda calculus with the bi-directional type system to web assembly.

2.1 WAST Description:

All the functions have following type:

(func (param i32) (param i32) (result i32))

First arg: *arg to be given*

Second arg: *Closure address of the function being called*

Corresponding to each global declaration there is a global variable of that name which is initialized to the closure address of a function which executes all the code on the right hand side of assignment dynamically and returns the result.

2.2 Compilation Algorithm Overview:

2.3 Global Data Structures Maintained:

A structure "state" is passed to the generate_code method which has following members:

1. funcs_code: Stores the code corresponding to all the web assembly functions. More details in lambda abs compilation section.
2. Global Context : Map from *string* to *ty*, which stores types of all the named definitions.

2.4 Compilation of a *compilationUnit*:

let x = term in compUnit Check all the exports are low

1. Infer type of term (say T)
2. Add (x: T) to the context obtaining a new state
3. Recursively compile compUnit

2.5 Compilation of a Term:

- x → get_local $x
- ?x.term →
  1. Infer type of the function_body, set isUn flag
  2. Allocate a low integrity closure
  3. isUn = True → store the low integrity closure address in the dyanamic sealing table and set tab_index = index in the table
  4. isUn = False → set tab_index = low_integrity memory pointer (static)
  5. Compile term
  6. Generate closure initialization code
- term1 term2 →
  1. Compile term1 followed by term2.
  2. Trivial except, function call depends on type of term1 (Low or High)
- Val n → i32.const
- Ref t | Deref t → Trivial except store_high (load_high) or store_low (load_low) called based on type of t (high or low)
- Fix f → compile recursively (App (f, (Fix f)))
- Constructor (name, params) →
  1. Hash name to i32, say hName
  2. Compile params recursively as:
     (a) param1, params →
        i. Compile (Ref param1) as val1
        ii. Compile params recursively as val2
        iii. Return (leftShift val1) + val2
- Pair (term1, term2) → Compile each part separately as a reference.

2.6 Linking

- Stitch functions from all the linking modules together
- Remove all high exports from from the linked intermediate module
- Generate a low module with low memory, exposing store_low and load_low functions to be used by all the high modules