UNIVERSITEIT UTRECHT, DEPARTMENT OF INFORMATION AND COMPUTING SCIENCES

Theory of Programming and Types

Mathijs Baaijens, 3542068
Nico Naus, 3472353

10 juni 2014

1 ABSTRACT

In this paper we will describe an extension to the type-correct, stack-safe, provably correct expression compiler described in the paper "A type-correct, stack-safe, provably correct expression compiler in Epigram". Our extension adds 'let' bindings to this compiler. We will describe the following components of our extension:

- evaluation semantics
- compiler
- interpreter
- correctness proof

2 INTRODUCTION

"A type-correct, stack-safe, provably correct expression compiler in Epigram"by McKinna and Wright is a nice, practical paper about proving the correctness of a simple compiler in a dependently typed programming language. In their paper, the authors use two semantics, one being this compiler, and then proof that both semantics result in the same value for every possible program. Their language consists only of boolean and integer values, plus and if-then-else. We will extend their work to show that this proof can also be done when we extend the language with let-bindings.

3 TYPED STACKS AND REFERENCES

Both the **eval** and the **exec & compile** evaluation mechanism use typed stacks. Simple type expressions are encoded in the `TyExp` datatype. The value datatype encodes values on the stack and is indexed by the type of the value.

```
data TyExp :  Set where
  TyNat :  TyExp
```

```
  TyBool :  TyExp

data Val :  TyExp → Set where
  nat :  ℕ → Val TyNat
  bool :  Bool → Val TyBool
```

The definition of tuples is straightforward and given here for clarity

```
data _x_ (A B : Set) :  Set where
  <_,_> :  A → B → A x B

fst :  A B : Set → A x B → A
fst < x , y > = x

snd :  A B : Set → A x B → B
snd < x , y > = y
```

A context is a list of type expression paired with a boolean indicating whether the stack value is bound to a `let` statement. This boolean will become important in section 3 when converting between stacks used in the **eval** semantics and the **exec & compile** semantics.

```
Γ = List (Bool x TyExp)
```

A stack is a list of values. Elements can only appended at the start of a list. The stack datatype is indexed by a context.

```
data Stack :  Γ → Set where
  empty :  Stack []
  _▷_ :  ∀ b t s → (v:Val t) → (xs :  Stack s) → Stack (<b,t> ::  s)
```

The `Ref` datatype defines references to values on the stack. The datatype is indexed by the context of the stack the reference refers to and the type of the value the reference refers to.

```
data Ref :  Γ → TyExp → Set where
  Top :  ∀ G u → Ref (u ::  G) (snd u)
  Pop :  ∀ G u v → Ref G u → Ref (v ::  G) u
```

Now we are able to define a function to lookup values in a stack. The reference context is equal to the context of the stack, making it impossible to pass references pointing to invalid locations (or locations storing the wrong type of value). The lookup function therefore always returns a value.

```
slookup :  ∀ S t → Stack S → Ref S t → Val t
slookup (v ▷ xs) Top = v
slookup (v ▷ xs) (Pop b₁) = slookup xs b₁
```

4 The First Semantics : eval

We start with defining expressions. The expression datatype is indexed by the following arguments:

- The type of the result of the expression.
- A context that defines the layout of the stack the required by `var` expressions to look up values.
- A boolean value indicating whether the result of this expression will be pushed on the stack as a value bound by a let expression.

```
data Exp :  TyExp → Γ → Bool → Set where
  var :  ∀ ctx t b → Ref ctx t → Exp t ctx b
  let1 :  ∀ ctx t1 t2 b → Exp t1 ctx true → Exp t2 (<true,t1> ::ctx) b →
Exp t2 ctx b
```

The `var` constructor takes a reference to a value on the stack. The let constructor takes two expressions. The evaluation context of the second expression is extended with a value of the return type of the first expression.

Now we can write the evaluation function. The evaluation functions takes a expression with result type t and produces a value of type t. The evalation function also takes as argument a stack of all values bound in preceding ĺet' statements.

```
eval :  ∀ t1 ctx b → (e :  Exp t1 ctx b) → Stack ctx → Val t1
```

Evaluation a `var` expression is straightforward, we simply look up the reference in the stack and return it.

```
eval (var x) env = slookup env x
```

When we evaluate a let-binding, we first evaluate e_1, and push it into the environment. We use this updated environment to evaluate e_2.

```
eval (let1 e1 e2) env = eval e2 ((eval e1 env) ▷ env)
```

5 The Second Semantics : compile & exec

The semantics we are about to define is the actual compiler we want to proof to be correct. Only the extensions that are needed for let-bindings are shown here. For the full compiler and evaluator, refer to the source code.

5.1 Specifying intermediate code

The code needs to accommodate let-bindings. This requires the stack-operations Load from Stack and Pop Stack.

```
data Code :  Γ → Γ → Set where
  LDS : ∀ S t b → (f :  Ref S t) → Code S (< b , t > ::  S)
  POP : ∀ b S t1 t2 → Code (<b,t1> ::  (<true,t2> ::  S)) (<b,t1> ::  S)
```

5.2 Implementing an Interpreter for Intermediate Code

The execution function is extended accordingly.

```
exec :  S S' : Γ → Code S S' → Stack S → Stack S'
exec (LDS f) s = (slookup s f) ▷ s
exec POP (v ▷ (v₁ ▷ s)) = v ▷ s
```

5.3 Converting between contexts

We define a function to convert an execution context to an evaluation context.

```
trimContext :  Γ → Γ
trimContext [] = []
trimContext (< true , x₁ > ::  s) = < true , x₁ > ::  trimContext s
trimContext (< false , x₁ > ::  s) = trimContext s
```

Now we can define a function that converts evaluation stack references to execution stack references. The `trimEnv` function is used to establish a relation between the context of the stacks. Because an evaluation stack is always a subset of an execution stack the conversion is always possible.

```
convertRef :  ∀ S t → Ref (trimContext S) t → Ref S t
convertRef [] ()
convertRef < true , x₁ > ::  S Top = Top
convertRef < true , x₁ > ::  S (Pop s) = Pop (convertRef s)
convertRef < false , x₁ > ::  S s = Pop (convertRef s)
```

5.4 Implementing the compiler to intermediate code

Implementing compilation is straightforward. Just as for `convertRef` the relation between the evaluation stack and the execution stack is expressed using the trimContext function.

```
compile :  ∀ b S t → (e :  Exp t (trimContext S) b) → Code S (<b,t> ::  S)
compile (var x) = LDS (convertRef x)
compile (let₁ e e₁) = compile e ++₁ (compile e₁ ++₁ POP)
```

6 Compiler Correctness

To define the correctness proof we need the equivalent of `trimContext` for stacks. The `trimStack` function removes all values from a stack that are not bound by 'let' expressions.

```
trimStack :  ∀ S → Stack S → Stack (trimEnv S)
trimStack [] x = empty
trimStack < true , x₁ > ::  S (v ▷ x₂) = v ▷ (trimStack x₂)
trimStack < false , x₁ > ::  S (v ▷ x₂) = trimStack x₂
```

We start by proving that looking up an evaluation stack reference in a execution stack converted to an evaluation stack equals looking up the reference converted to an execution stack reference in the execution stack reference.

```
lemma :  ∀ S t → (x :  Ref (trimEnv S) t) → (s :  Stack S)
              → (slookup (trimStack s) x) ≡ (slookup s (convertRef x))
```

```
lemma [] () s
lemma < true , t > ::  S Top (v ▷ s) = refl
lemma < true , x1 > ::  S (Pop e) (v ▷ s) = lemma e s
lemma < false , x1 > ::  S e (v ▷ s) = lemma e s
```

Now we can define the correctness proof of our compiler.

```
correct :  ∀ b S t → (e :  Exp t (trimEnv S) b) → (s :  Stack S)
                   → ((eval e (trimStack s)) ▷ s) ≡ (exec (compile e) s)
correct (var x) s with lemma x s
...  | p with slookup (trimStack s) x | slookup s (convertRef x)
correct (var x) s | refl | .l | l = refl
correct (let1 e e1) s with correct e s
...  | p1 with exec (compile e) s | eval e (trimStack s)
correct (let1 e e1) s | refl | .(p3 ▷ s) | p3 with correct e1 (_▷_ true p3
s)
...  | p4 with exec (compile e1) (_▷_ true p3 s) | eval e1 (p3 ▷ trimStack
s)
correct (let1 e e1) s | refl | .(p3 ▷ s) | p3 | refl | .(p6 ▷ (p3 ▷ s)) |
p6 = refl
```

7 Conclusion

We have indeed seen that it is possible to prove correctness for the defined language. The proof however is not all that trivial. In theory, it should be possible to construct a proof for a far more elaborate imperative language, but the proof gets really complicated, really fast. When comparing our work to previous, it is evident that adding just let-bindings makes matters much worse, proof wise.

8 Related work

A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language [1]. Here the author presents a certified compiler for a language similar to ours, with a machine-checked correctness proof written in Coq.

Referenties

[1] Adam Chlipala, *A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language*. Proceedings PLDI '07, p54-65, New York, 2007.