UNIVERSITEIT UTRECHT, DEPARTMENT OF INFORMATION AND COMPUTING SCIENCES

Theory of Programming and Types

Mathijs Baaijens
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 THE FIRST SEMANTICS : EVAL

We start by defining our language. It consists of a simple, typed labda calculus. This language is based on the one in the paper by McKinna, but extended with let-bindings.

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

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

In order to make these let-bindings work, we will first construct a context. This context is represented by a list of tuples holding a boolean and a TyExp

```
- ** Tuples **

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

- ** Context **

Γ = List (Bool x TyExp)

- ** Stack **

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

Next, we define references.

```
- ** References **

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

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

Lastly we need to add the let binding to the expressions.

```
- ** Exp **

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
```

Now the evaluation function. An environment is now passed around. When evaluating a var expression, we can look it up in this environment and return it. 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 **

eval :  ∀ t₁ ctx b → (e :  Exp t₁ ctx b) → Stack ctx → Val t₁
eval (var x) env = slookup env x
eval (let₁ e₁ e₂) env = eval e₂ ((eval e₁ env) ▷ env)
```

4 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.

4.1 Specifying intermediate code

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

```
- ** Code **
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)
```

4.2 Implementing an Interpreter for Intermediate Code

The execution function is extended accordingly.

```
- ** Exec **
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
```

4.3 Converting between contexts

```
- ** Converting between contexts **
trimEnv :  Γ → Γ
trimEnv [] = []
trimEnv (< true , x₁ > ::  s) = < true , x₁ > ::  trimEnv s
trimEnv (< false , x₁ > ::  s) = trimEnv s

convertRef :  ∀ S t → Ref (trimEnv 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)
```

4.4 Implementing the compiler to intermediate code

For the actual compiler, we compile a variable expression to loading it's value from stack. Let-bindings require for the bound expression to be evaluated, then popped, and then for evaluation of the body, now that the boud expression is available.

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

5 Compiler Correctness

```
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

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

correct : ∀ b S t → (e : Exp t (trimEnv S) b) → (s : Stack S) → ((eval
e (trimStack s)) ▷ s) âĽą (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
```

6 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 labda-calculus, 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.

7 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.