UNIVERSITEIT UTRECHT, DEPARTMENT OF INFORMATION AND COMPUTING SCIENCES

Theory of Programming and Types

Mathijs Baaijens
Nico Naus, 3472353

9 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

3 THE FIRST SEMANTICS : EVAL

```
- ** Types & values **

data TyExp : Set where
TyNat : TyExp
TyBool : TyExp

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

- ** 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)

- ** 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

- ** Ref **

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

- ** 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

- ** 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)
```

3.1 Type preservation is the type of the interpreter

4 The Second Semantics : compile & exec

4.1 Specifying intermediate code

```
- ** 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

```
- ** 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

```
- ** 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

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.