Abstract

An exception compilation scheme that dynamically creates and removes exception handler entries on the stack. A formalization of an article of the same name by Hutton and Wright [1].

1 Compiling exception handling

theory Exceptions
imports Main
begin

1.1 The source language
datatype expr = Val int | Add expr expr | Throw | Catch expr expr

primrec eval :: "expr ⇒ int option"
where
"eval (Val i) = Some i"
"eval (Add x y) = (case eval x of None ⇒ None | Some i ⇒ (case eval y of None ⇒ None | Some j ⇒ Some(i+j)))"
"eval Throw = None"
"eval (Catch x h) = (case eval x of None ⇒ eval h | Some i ⇒ Some i)"

1.2 The target language
datatype instr =
  Push int | ADD | THROW | Mark nat | Unmark | Label nat | Jump nat
datatype item = VAL int | HAN nat
type_synonym code = "instr list"
type_synonym stack = "item list"

fun jump where
"jump 1 [] = []"
lemma size_jump1: "size (jump l cs) < Suc (size cs)"
apply (induct cs)
  apply simp
  apply (case_tac a)
  apply auto
done

lemma size_jump2: "size (jump l cs) < size cs ∨ jump l cs = cs"
apply (induct cs)
  apply simp
  apply (case_tac a)
  apply auto
done

function (sequential) exec2 :: "bool ⇒ code ⇒ stack ⇒ stack"
where
  "exec2 True [] s = s"
| "exec2 True (Push i#cs) s = exec2 True cs (VAL i # s)"
| "exec2 True (ADD#cs) (VAL j # VAL i # s) = exec2 True cs (VAL(i+j) # s)"
| "exec2 True (THROW#cs) s = exec2 False cs s"
| "exec2 True (Mark l#cs) s = exec2 True cs (HAN l # s)"
| "exec2 True (Unmark#cs) (v # HAN l # s) = exec2 True cs (v # s)"
| "exec2 True (Label l#cs) s = exec2 True cs s"
| "exec2 True (Jump l#cs) s = exec2 True (jump l cs) s"
| "exec2 False cs [] = []"
| "exec2 False cs (VAL i # s) = exec2 False cs s"
| "exec2 False cs (HAN l # s) = exec2 True (jump l cs) s"
by pat_completeness auto

termination by (relation
  "inv_image (measure(%cs. size cs) <*lex*> measure(%s. size s)) (%(b,cs,s). (cs,s))")
  (auto simp add: size_jump1 size_jump2)

abbreviation "exec ≡ exec2 True"
abbreviation "unwind ≡ exec2 False"

1.3 The compiler
primrec compile :: "nat ⇒ expr ⇒ code * nat"
where
  "compile l (Val i) = ([Push i], l)"
| "compile l (Add x y) = (let (xs,m) = compile l x; (ys,n) = compile m y
  in (xs @ ys @ [ADD], n))"
| "compile l Throw = ([THROW],l)"
| "compile l (Catch x h) =
  (let (xs,m) = compile (l+2) x; (hs,n) = compile m h
  in (xs @ hs @ [HAN l], n))"


abbreviation
cmp :: "nat ⇒ expr ⇒ code" where
"cmp l e == fst(compile l e)"

primrec isFresh :: "nat ⇒ stack ⇒ bool" where
"isFresh l [] = True"
| "isFresh l (it#s) = (case it of VAL i ⇒ isFresh l s
| HAN l' ⇒ l' < l ∧ isFresh l s)"

definition
conv :: "code ⇒ stack ⇒ int option ⇒ stack" where
"conv cs s io = (case io of None ⇒ unwind cs s
| Some i ⇒ exec cs (VAL i # s))"

1.4 The proofs

Lemma numbers are the same as in the paper.

declare
  conv_def[simp] option.splits[split] Let_def[simp]

lemma 3:
"(∀l. c = Label l =⇒ isFresh l s) =⇒ unwind (c#cs) s = unwind cs s"
apply (induct s)
apply simp
apply (auto)
apply (case_tac a)
apply auto
apply (case_tac c)
apply auto
done

corollary [simp]:
"(!l. c ≠ Label l) =⇒ unwind (c#cs) s = unwind cs s"
by (blast intro: 3)

corollary [simp]:
"isFresh l s =⇒ unwind (Label l#cs) s = unwind cs s"
by (blast intro: 3)

lemma 5: "[ isFresh l s; l ≤ m ] =⇒ isFresh m s"
apply (induct s)
apply simp
apply (auto split:item.split)
done
corollary [simp]: "isFresh l s \implies isFresh (Suc l) s"
by (auto intro:5)

lemma 6: "\forall l. l \leq snd(compile l e)"
proof (induct e)
  case Val thus ?case by simp
next
case (Add x y)
  from 'l \leq snd (compile l x)'
  and 'snd (compile l x) \leq snd (compile (snd (compile l x)) y)'
  show ?case by (simp_all add:split_def)
next
case Throw thus ?case by simp
next
case (Catch x h)
  from 'l+2 \leq snd (compile (l+2) x)'
  and 'snd (compile (l+2) x) \leq snd (compile (snd (compile (l+2) x)) h)'
  show ?case by (simp_all add:split_def)
qed
corollary [simp]: "l < m \implies l < snd(compile m e)"
using 6[where l = m and e = e] by auto
corollary [simp]: "isFresh l s \implies isFresh (snd(compile l e)) s"
using 5 6 by blast

Contrary to what the paper says, the proof of lemma 4 does not just need lemma 3 but also the above corollary of 5 and 6. Hence the strange order of the lemmas in our proof.

lemma 4 [simp]: "\forall l cs. isFresh l s \implies unwind (cmp l e @ cs) s = unwind cs s"
by (induct e) (auto simp add:split_def)

lemma 7 [simp]: "l < m \implies jump l (cmp m e @ cs) = jump l cs"
by (induct e arbitrary: m cs) (simp_all add:split_def)

The compiler correctness theorem:
theorem comp_corr:
  "\forall l s cs. isFresh l s \implies exec (cmp l e @ cs) s = conv cs s (eval e)"
by (induct e)(auto simp add:split_def)

The specialized and more readable version (omitted in the paper):
corollary "exec (cmp l e) [] = (case eval e of None \Rightarrow [] | Some n \Rightarrow [VAL n])"
by (simp add: comp_corr[where cs = "[]", simplified])
References