Front-end optimisations

Compiler Construction
Dan R. Ghica

Optimisation before code generation

- we have the parse tree
- we have an equivalence in the PL
  - $P_1 \cong P_2$
  - P_2 is “easier to compute” than P_1
    - $P_1 \Longrightarrow P_2$ (usually but not necessarily)
  - replace P_1 with P_2
- note: the equivalence must hold in **all contexts**!

Constant folding

- we detect an expression with constant operands
- we carry out the operation at compile time
- example:
  - `342 * 433 ≅ 148046`
  - `“hello” ^ “ “ ^ “world ≅ “hello world”`
  - `if 4 > 5 then 7*5 else 9-3`
    `≅ if false then 7*5 else 9-3`
    `≅ 9-3`
    `≅ 6`

Constant propagation

- it involves variables which have known values
- it builds on top of constant folding
- example:
  - `let x = 10 in let y = 20 in x * y`
    `≅ 10 * 20`
    `≅ 200`
  - `let x = 4 in let y = 5 in if x > y then 7*5 else 9-3`
    `≅ if false then 7*5 else 9-3`
    `≅ 9-3`
    `≅ 6`

Function inlining

- it involves functions which have known arguments
- it builds on top of constant propagation and folding
- example:
  - `let f x y = if x > y then 7*x else 9-y in`
    `f 4 5`
    `≅ if 4 > 5 then 7*4 else 9-5`
    `≅ if false then 28 else 4`
    `≅ 4`
- **Note:** Functions can also be inlined for non-constant arguments
  - “de-sugaring” syntax

Function inlining caveats

- functions can be inlined for non-constant arguments (symbolic)
- but you need to be very careful!
  - `let f x = x + x in f (g y)` ≅ `(g y) + (g y)` ?
- also, you are only guaranteed savings on **affine** use
  - see the above
  - saving a function call but re-evaluation!

Partial evaluation

- precompute **everything** that can be precomputed at compile time
- a classic paper (“Futamura projections”):
  - Yoshihiko Futamura. *Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler* [PDF] (advanced material)

State and propagation

- Constant propagation can also involve assignment
- example:
    - `x:=1; y:=!x+3 ≅ x:=1; y:=4`
- Can be trickier!
    - if functions / procedures are involved
    - if concurrency / parallelism is involved

Fancier optimisation

- common sub-expression elimination:
  - `let x = a + b * c + d in`
    `let y = e + b * c + d in …`
    ≅
    `let bc = b * c in`
    `let x = a + bc + d in`
    `let y = e + bc + d in …`
- But be careful if effects are involved!
  - `… b() * c () …`

Even fancier optimisation

- memoisation
- automatic parallelisation
  `f()+g()` $\Longrightarrow$ `{x:=f() || y:= g()}; !x+!y`
- loop unrolling (+parallelisation)
  - `for x = 1 to 5 do f(x)`
    $\Longrightarrow$ `f(1) ; f(2) ; f(3) ; f(4) ; f(5)`
    $\Longrightarrow$ `f(1) || f(2) || f(3) || f(4) || f(5)`

Weekly task

- implement front-end optimisations (using the evaluator)
  - basic : constant folding + constant inlining
  - medium : constant propagation + loop unrolling
  - advanced : anything else
- testing:
  - basic :10 more test cases to illustrate your best optimisations
  - make optimisations optional (-o)
  - compare output
  - advanced : count evaluation steps (define a monad)