

Compiler Construction

Lecture 14: **optimisation**

Jeremy Yallop

`jeremy.yallop@cl.cam.ac.uk`

Lent 2026

Optimisation

What's an optimisation?

Optimisation

A compiler optimisation changes the code generated for a program to:

- improve its space usage
- reduce its size
- introduce parallelism
- reduce energy usage
- reduce allocation
- improve locality
- reduce stack usage
- (etc.)

Specialisation

Other

optimisations

Optimisations preserve program semantics, but improve program pragmatics.

What does it mean to preserve program semantics?

Undefined behaviour

Which optimisations are valid?

Optimisation

An optimisation is **valid** if its input and output have **equivalent semantics**.

We might use the definition of equivalence from *Semantics* (slide 256):

We say that typed L3 programs $\Gamma \vdash e_1 : T$
 $\Gamma \vdash e_2 : T$ are **contextually equivalent** if
for every context C such that $\cdot \vdash C[e_1] : \text{unit}$
 $\cdot \vdash C[e_2] : \text{unit}$ we have either

- $\langle C[e_1], \{\} \rangle \longrightarrow^\omega$
 $\langle C[e_2], \{\} \rangle \longrightarrow^\omega$
- for some s_1
 s_2 we have $\langle C[e_1], \{\} \rangle \rightarrow \langle \text{skip}, s_1 \rangle$
 $\langle C[e_2], \{\} \rangle \rightarrow \langle \text{skip}, s_2 \rangle$

Specialisation

Other optimisations

Undefined behaviour

(Note: as we shall see, optimisations can actually *reduce* the set of possible behaviours of a program)

What does it mean to preserve program semantics?

Optimisation

The definition of contextual equivalence needs adjustment for larger languages.

For example, it makes all non-terminating programs equal, regardless of effects.

This is not what we want: consider

```
let rec repeat_say msg =  
  print_endline msg;  
  repeat_say ()  
  
let () = say "Hello"
```

```
let rec repeat_say msg =  
  print_endline msg;  
  repeat_say ()  
  
let () = say "Goodbye"
```

Other

optimisations

We'll use the following (very informal) definition:

An optimisation is valid if the output program has

the same effects

the same termination behaviour

the same return value.

Undefined
behaviour

Optimisation

Are the following optimisations **valid in general**?

`let _ = g 2 in f 3` \rightsquigarrow `f 3`

invalid: `g 2` may perform effects

`let x = g 2 in f 3 + x` \rightsquigarrow `f 3 + g 2`

depends on the order of operand evaluation

`map f (map g l)` \rightsquigarrow `map (fun x → f (g x)) l`

invalid if `f` and `g` perform (non-commuting) effects

`if true then e1 else e2` \rightsquigarrow `e1`

valid

`let rec loop () = loop () in (loop(); print_endline "done")` \rightsquigarrow `let rec loop () = loop () in (loop(); ())`

valid

`fold_right f l u` \rightsquigarrow `fold_left (fun x y → f y x) u (rev l)`

valid

Specialisation

Other optimisations

Undefined behaviour

Specialisations

Inlining: examples

Optimisation

Inlining replaces a variable with its definition (typically a function):

```
let succ x = x + 1
let f = map (fun y → succ y)
           [1;2;3]
```

inline →

```
let succ x = x + 1
let f = map (fun y → y + 1)
           [1;2;3]
```

Specialisation

Note: care with free variables is needed:

```
let f y =
  let addy x = x + y in
  map (fun y → addy y) [1;2;3]
```

~~inline~~ →

```
let f y =
  let addy x = x + y in
  map (fun y → y + y) [1;2;3]
```

inline →

```
let f y =
  let addy x = x + y in
  map (fun z → z + y) [1;2;3]
```

Other
optimisations

Undefined
behaviour

Optimisation

Inlining is an **enabling transformation** that exposes optimisation opportunities.

Inlining can sometimes be a **pessimisation**. Questions to consider in each case:

- Does inlining duplicate code?
- Does inlining duplicate work?
- Does inlining expose further optimisation opportunities?

Note: inlining recursive bindings is significantly harder.

Lots of details:

Secrets of the Glasgow Haskell Compiler inliner (1999)
Simon Peyton Jones and Simon Marlow

Specialisation

Other

optimisations

Undefined
behaviour

Monomorphisation (MLton)

Optimisation

Specialisation

Other
optimisations

Monomorphisation replaces parameterised types with unparameterised types
polymorphic functions with monomorphic functions

```
type 'a t = T of 'a
let f (x: 'a) = T x
let a = f 1
let b = f 2
let z = f (3, 4)
```

monomorphise

```
type t1 = T1 of int
type t2 = T2 of int * int
let f1 (x: int) = T1 x
let f2 (x: int * int) = T2 x
let a = f1 1
let b = f1 2
let z = f2 (3, 4)
```

Undefined
behaviour

Monomorphisation: benefits

Optimisation

Monomorphisation is also an **enabling transformation**.

The compiler can subsequently specialise representations, e.g. flattening tuples:



Specialisation

Other
optimisations

Monomorphisation is used in MLton, a whole-program-optimising ML compiler:

Whole-Program Compilation in MLton (2006)

Stephen Weeks

Undefined
behaviour

Contification turns a function into a continuation.

Contification applies when a function is always passed the same continuation.

```
let g y = y - 2
let f b =
  (if b then g 13 else g 15) + 1
```

CPS conversion



```
let g y k = k (y - 2)
let f b k =
  let k' x = k (x + 1) in
  if b then g 13 k' else g 15 k'
```

contification

```
let f b k =
  let k' x = k (x + 1) in
  let g y = k' (y - 2) in
  if b then g 13 else g 15
```

inline k' and g



```
let f b k =
  if b then k 12 else k 14
```

Optimisation

Specialisation

Other
optimisations

Undefined
behaviour

Contification is also used in MLton:

Contification Using Dominators (2001)
Matthew Fluet Stephen Weeks

where it was found to

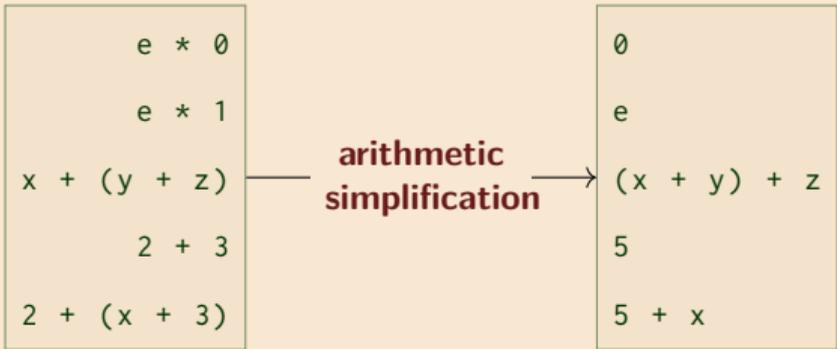
- have minimal effects on compile-time (2–4%)
- significantly reduce run-time (up to 86%)
- reduce executable size (up to 12%)

Other optimisations

Arithmetic simplification

Optimisation

Specialisation



Inlining may expose opportunities for arithmetic simplification.

Care needed: $e * 0 \rightsquigarrow 0$ only valid if e has no effects.

Care needed: very few arithmetic laws apply to floating-point numbers.

Other
optimisations



Undefined
behaviour

Tail-recursion modulo cons: motivation

Optimisation

Observation: It is difficult to implement map entirely satisfactorily:

Naive

```
let rec map f l =  
  match l with  
  | [] → []  
  | x :: xs →  
    f x :: map f xs
```

May run out of stack

CPS

```
let rec map f l k =  
  match l with  
  | [] → k []  
  | x :: xs →  
    f x (fun hd →  
      map f xs (fun tl →  
        k (hd::tl)))
```

```
let map f l =  
  map f l (fun x → x)
```

Allocates frames on the heap

Accumulator

```
let rec map f l acc =  
  match l with  
  | [] → rev acc  
  | x :: xs →  
    map f xs (f x :: acc)  
  
let map f l = map f l []
```

Traverses the list twice

Specialisation

Other
optimisations



Undefined
behaviour

Tail-recursion modulo cons: destination passing style

Optimisation

The **TRMC** optimisation transforms functions into **destination-passing style**:

Naive

```
let rec map f l =  
  match l with  
  | [] → []  
  | x :: xs →  
    f x :: map f xs
```

TRMC

Destination-passing style

```
let rec map f = function  
| [] → []  
| x :: xs → let y = f x in  
             let dst = y :: <Hole> in  
             map_dps dst 1 f xs;  
             dst  
and map_dps dst i f = function  
| [] →  
  dst.i <- []  
| x :: xs → let y = f x in  
            let dst' = y :: <Hole> in  
            dst.[i] <- dst';  
            map_dps dst' 1 f xs
```

Specialisation

Other
optimisations



Undefined
behaviour

Idea: allocate a partially-constructed cons cell with an uninitialized tail.
Pass the cons cell (the “destination”) to recursive calls.
Write the result of each call to the tail field of the destination.

Optimisations and undefined behaviour

Optimising programs with undefined behaviour

Optimisation

Our optimisation correctness criterion is based on the behaviour of programs.

Specialisation

What optimisations are justified when a program's behaviour is undefined?

Two principles:

Other
optimisations

1. There are no constraints on the behaviour of programs with undefined behaviour.
2. A compiler can therefore assume that programs do not have undefined behaviour.

Consequently, optimisation can change the observed behaviour of ill-defined programs.

**Undefined
behaviour**



Integer overflow

Optimisation

sum.c

```
#include <stdio.h>
#include <limits.h>

int sum_range(int start, int len) {
    int total = 0;
    for (int i = start; i <= start + len; i += 1) total += i;
    return total;
}

int main() {
    printf("%d %d\n", sum_range(10, 10), sum_range(INT_MAX-1, 2));
}
```

Specialisation

Other
optimisations

Without optimisation

```
$ clang -o sum sum.c
$ ./sum
165 0
```

With optimisation

```
$ clang -O3 -o sum sum.c
$ ./sum
165 2147483646
```

Undefined
behaviour

(Adapted from an example by Taras Tsugrii)

Integer overflow: what is going on?

Optimisation

sum.c (excerpt)

```
int sum_range(int start, int len) {  
    int total = 0;  
    for (int i = start; i <= start + len; i += 1) total += i;  
    return total;  
}
```

Specialisation

Other
optimisations

Some reasoning about arithmetic justifies a significant optimisation:

$$\begin{aligned} & \text{sum_range}(\text{start}, \text{len}) \\ \equiv & \text{start} + (\text{start} + 1) + \dots + (\text{start} + \text{len}) \\ \equiv & \text{start} \times (\text{len} + 1) + 1 + \dots + \text{len} \\ \equiv & \text{start} \times (\text{len} + 1) + (\text{len} \times (\text{len} + 1)) / 2 \end{aligned}$$

This reasoning assumes that integer overflow cannot occur.

Undefined
behaviour



Optimisation

null.c

```
#include <stdio.h>

static void (*action)(void) = NULL;

int main(void) { action(); }

static void erase_all_files(void) { puts("deleting all files..."); }
void never_called(void) { action = erase_all_files; }
```

Specialisation

Other
optimisations

Without optimisation

```
$ clang -o null null.c
$ ./null
Segmentation fault
```

With optimisation

```
$ clang -O3 -o null null.c
$ ./null
deleting all files
```

Undefined
behaviour

(Adapted from an example by Krister Walfridsson)

Null pointers: what is going on?

Optimisation

null.c

```
#include <stdio.h>

static void (*action)(void) = NULL;

int main(void) { action(); }

static void erase_all_files(void) { puts("deleting all files..."); }
void never_called(void) { action = erase_all_files; }
```

Specialisation

Other
optimisations

The following reasoning about the program justifies the “optimisation”:

- There is only one assignment to `action`, setting it to `erase_all_files`
- `action` must therefore equal either its initial value (`NULL`) or `erase_all_files`
- if `action` is `NULL`, the program has undefined (unconstrained) behaviour
- so calling `erase_all_files` is valid for all possible values of `action`

Undefined
behaviour



Optimisation

Specialisation

Other optimisations

alias.c

```
#include <stdio.h>

long read_write(long *p, int *q) {
    *p = 3;
    *q = 4;
    return *p;
}

int main(void) {
    long x;
    printf("%ld\n", read_write(&x, (int*)&x));
}
```

Without optimisation

```
$ clang -o alias alias.c
$ ./alias
4
```

With optimisation

```
$ clang -O3 -o alias alias.c
$ ./alias
3
```

Undefined
behaviour



Aliasing: what is going on?

Optimisation

Specialisation

Other
optimisations

alias.c (excerpt)

```
long read_write(long *p, int *q) {  
    *p = 3;  
    *q = 4;  
    return *p;  
}
```

C forbids writing to the same object through both `long *` and `int *`.

The compiler assumes that writing to `*q` cannot affect the value at `*p`.

Undefined
behaviour



Next time: linking