universität freiburg

Code Generation

Mathias Fleury

Tutorial Vienna, 2024-1-12

Internals of The Code Generator

Imperative Code

Getting Rid of the Translation: Isabelle_LLVM

Getting Rid of the Translation and the Compiler: Targetting CakeML

Conclusion

Aim

- Generate code from an Isabelle function, with same behavior
- The code should perform reasonably
- The code should be reasonable
- The code should be correct (for some definition of correct)

3 Flavors of Code-Generation

- export existing functional HOL functions
  - How? Just write functions
- export HOL_Imperative functions in imperative code (using existing code generation)
  - How? Just write functions in the HOL_Imperative
  - Or use Sepref to synthesize the function from the non-determinism HOL
- export Isabelle_LLVM to generate (without using existing code generation)
  - How? Use the other Sepref to synthesize the function from the other non-determinism exception HOL

Key difference: Isabelle abstracts over the semantics of imperative code. So sometimes not matching, like arr[INT128_MAX].

3 Flavors of Code-Generation

- export existing functional HOL functions
  - How? Just write functions
- export HOL_Imperative functions in imperative code (using existing code generation)
  - How? Just write functions in the HOL_Imperative
  - Or use Sepref to synthesize the function from the non-determinism HOL
- export Isabelle_LLVM to generate (without using existing code generation)
  - How? Use the other Sepref to synthesize the function from the other non-determinism exception HOL

Key difference: Isabelle abstracts over the semantics of imperative code. So sometimes not matching, like arr[INT128_MAX].

3 Flavors of Code-Generation

- export existing functional HOL functions
  - How? Just write functions
- export HOL_Imperative functions in imperative code (using existing code generation)
  - How? Just write functions in the HOL_Imperative
  - Or use Sepref to synthesize the function from the non-determinism HOL
- export Isabelle_LLVM to generate (without using existing code generation)
  - How? Use the other Sepref to synthesize the function from the other non-determinism exception HOL

Key difference: Isabelle abstracts over the semantics of imperative code. So sometimes not matching, like arr[INT128_MAX].

3 Flavors of Code-Generation

- export existing functional HOL functions
  - How? Just write functions
- export HOL_Imperative functions in imperative code (using existing code generation)
  - How? Just write functions in the HOL_Imperative
  - Or use Sepref to synthesize the function from the non-determinism HOL
- export Isabelle_LLVM to generate (without using existing code generation)
  - How? Use the other Sepref to synthesize the function from the other non-determinism exception HOL

  Key difference: Isabelle abstracts over the semantics of imperative code. So sometimes not matching, like arr[INT128_MAX].

One Example

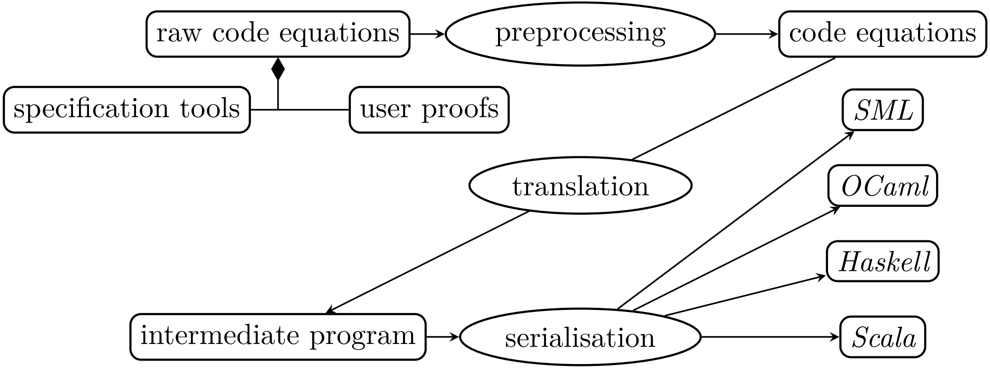
```
fun fib :: ‹nat ⇒ nat› where
  ‹fib 0 = 0› |
  ‹fib (Suc 0) = 1› |
  ‹fib (Suc (Suc n)) = fib (Suc n) + fib n›
```

```
export-code fib in SML
```

Now import HOL−Library.Code-Abstract-Nat or HOL−Library.Code-Binary-Nat and see the difference!

Internals of The Code Generator

General Infrastructure [Haftmann, PhD thesis]



raw code equations

preprocessing

code equations

specification tools

user proofs

translation

intermediate program

serialisation

SML

OCaml

Haskell

Scala

Dictionary Construction

```
fun test :: ‹'a :: order ⇒ 'a :: order ⇒ bool› where
  ‹test a b = (a < b)›
```

The translation becomes:

```
definition test-SML :: ‹'a  ⇒ 'a ⇒ bool› where
  ‹test-SML a b = (less-operator information-on-α a b)›
```

and pass information-on-α to each function.

Dictionary Construction

```
fun test :: ‹′a :: order ⇒ ′a :: order ⇒ bool› where
  ‹test a b = (a < b)›
```

The translation becomes:

```
definition test-SML :: ‹′a  ⇒ ′a ⇒ bool› where
  ‹test-SML a b = (less-operator information-on-α a b)›
```

and pass information-on-α to each function.

Intermediate Language

The intermediate language has 4 constructors: datatypes, fun, class, and inst.

Correctness

Theorem

The translation is correct if for any input, it fails or yields a compatible output.

Pretty-Printing

Finally, translation to target language is so simple that it is trusted.

If:

- the code setup is correct
- you do not violate the assumptions of the code setup
- the semantic the developer has in mind matches the compiler semantic

Pretty-Printing

Finally, translation to target language is so simple that it is trusted.

If:

- the code setup is correct
- you do not violate the assumptions of the code setup
- the semantic the developer has in mind matches the compiler semantic

Calling the Code Generator

eval is actually calling the code generator...
And you assume that it is correct because you can use the result.
Lochbihler has found a bug in the word implementation [ITP'18].

Imperative Code

Modelization

Idea:

1. Program depends on a polymorphic heap where you can put content
2. heap operations are automatically mapped by the code generator.

Modelization

Idea:

1. Program depends on a polymorphic heap where you can put content
2. heap operations are automatically mapped by the code generator.

Code Generation

Idea:

1. insert closure y to force ordering of operations.
2. actually the code generator attempts to remove some of them

Code Generation

Idea:

1. insert closure y to force ordering of operations.
2. actually the code generator attempts to remove some of them

Code Generation

```
definition f :: ‹nat ⇒ -› where
‹f x = do {
  a ← Array.new x (5::nat);
  a ← Array.upd 1 2 a;
  x ← Array.nth a 1;
  (if x = 0 then return a else Array.upd 1 2 a)
}›
```

```
export-code f in SML-imp
  module-name F — Do not forget to name the target module, or you get a very weird error message.
```

I am not aware of anyone using this directly.

Most people generating efficient code use Sepref that can transform a functional code on list into an array.

But: this is so simple that there is entire *day* planned on that.

I am not aware of anyone using this directly.

Most people generating efficient code use Sepref that can transform a functional code on list into an array.

But: this is so simple that there is entire *day* planned on that.

I am not aware of anyone using this directly.
Most people generating efficient code use Sepref that can transform a functional code on list into an array.
But: this is so simple that there is entire *day* planned on that.

Sepref

Quick idea:

1. Annotate you program with assertion.
2. Let Sepref translate data structures and program flow into imperative code

```
term
 ‹g xs n = do {
    ASSERT (n ≤ length xs);
    return (xs ! n)
  }›
```

Compiler

Many SML compilers out there... but for imperative code you should use MLton.
I tried at some point IsaSAT in PolyML (like Isabelle), MLton, Scala, and OCaml.
MLton one order of magnitude faster.

Compiler

Many SML compilers out there... but for imperative code you should use MLton.
I tried at some point IsaSAT in PolyML (like Isabelle), MLton, Scala, and OCaml.
MLton one order of magnitude faster.

Getting Rid of the Translation: Isabelle_LLVM

What is the Idea? [Lammich, ITP 19]

Idea:

1. have a modelization of the LLVM IR semantics
2. let Sepref generate LLVM IR
3. trivial pretty-printer!

Drawbacks

But: LLVM IR is purely imperative, so instead map ((+) (1::'a)) create a constant map-add-1 with a loop.

Also: no GMP integers.

This change alone made my SAT solver twice as fast.

Drawbacks

But: LLVM IR is purely imperative, so instead map ((+) $(1{::}'a)$) create a constant map-add-1 with a loop.
Also: no GMP integers.
This change alone made my SAT solver twice as fast.

Drawbacks

But: LLVM IR is purely imperative, so instead map ((+) (1::′a)) create a constant map-add-1 with a loop.
Also: no GMP integers.
This change alone made my SAT solver twice as fast.

Parallel Code

There is no some parallelization based on splitting arrays into 2.

I never saw how to use it in my SAT solver.

Parallel Code

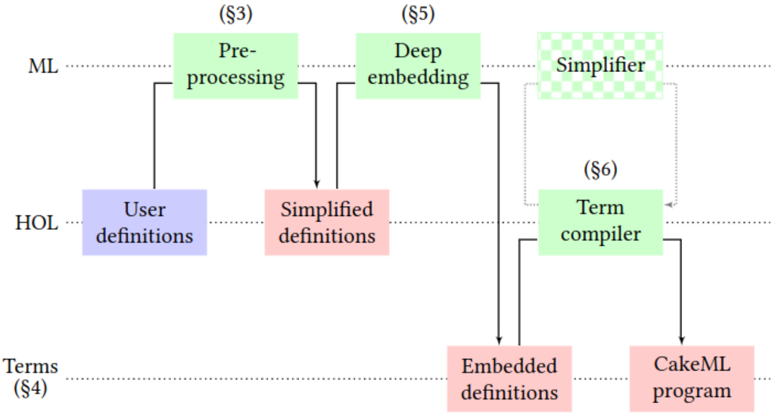
There is no some parallelization based on splitting arrays into 2.
I never saw how to use it in my SAT solver.

Getting Rid of the Translation and the Compiler: Ta

Idea [Hupel PhD thesis]

Verify the standard code generator approach.
And target the verified compiler CakeML, via Lem to translate the semantics.

Idea



green	implemented as part of this thesis (checkered: already available in Isabelle)
blue	specified by user
red	generated object

Issues

- Very slow
- No support for Imperative code
- no native types, like machine words

Conclusion