Generating bare-metal C code from a high-level pure specification

Olivier Delande

Prove & Run

Entropy 2019, Stockholm, 16/06/2019

PROVE & RUN

Context

To develop our micro-kernel ProvenCore, we:

- wrote the program, specs, and stated theorems in Smart, a pure functional language;
- interactively proved the theorems in our IDE;
- generated a bare-metal C implementation of the program (i.e., without runtime support).

Challenge of code generation: compile high-level constructs optimized for reasoning to constrained low-level executable C code.

Compiling Smart to C

Some aspects of the gap between Smart and C:

- Ghost code: Smart programs include *ghost* data structures and computations that solely support reasoning; the generator safely removes them from the program.
- **Functional updates** (this talk): Smart's data structures are immutable; a program updates them by producing new values; we compile these functional updates to in-place updates.
- C implementation details: the programmer can customize certain types, e.g. to fit a mandated memory layout.

Functional updates

In a pure language, updates are functional, e.g. in the program:

```
// A function pausing a process: takes the current value
// of the process and returns the new value
pause(p1 : proc) : proc

// ...
p2 := pause(p1)
```

a (potentially large) object undergoes a transformation `pause` and we get its new value `p2`, *leaving the old value* `p1` *unaffected*.

PROVE & RUN

Functional updates

In a pure language, updates are functional, e.g. in the program:

```
// A function pausing a process: takes the current value
// of the process and returns the new value
pause(p1 : proc) : proc

// ...
p2 := pause(p1)
```

a (potentially large) object undergoes a transformation `pause` and we get its new value `p2`, *leaving the old value* `p1` *unaffected*.
This is great for reasoning, but how do we implement this in C? We can:

- (naively) allocate a new object for `p2`, leaving `p1` unchanged, or
- update the object in place: overwrite `p1` with `p2`
  ... but **this is only correct if the remainder of the program does not depend on the overwritten value** `p1`.

Nested in-place updates

Sometimes the object we'd like to update in place lives somewhere inside a data structure:

```
// An array of processes
type procs = proc[NR_PROCS]
```

Let's `pause` the process at index `i` in some array `procs1`:

Functionally	Imperatively (in place)
`p1 := procs1[i]`	Take pointer to element (alias)
`p2 := pause(p1)`	Update element in place
`procs2 := procs1 with [i] = p2`	Nothing to do

In the imperative program, `pause` overwrites (invalidates) both `p1` and `procs1[i]`, but this is correct because we never read them afterwards.

Our approach to safe in-place updates

We support live read-write references to overlapping objects (i.e., aliases) in a controlled way: the program must not *depend* on the overwritten contents.
We safely achieve this thanks to two static analyses:

1. A dependency analysis determines which parts of the value of a variable will be needed in the future.
2. A points-to analysis tracks aliases.

Principle: it is safe to update an object in place if we do not depend on the values of any of its potential aliases.

Example

Let's run the dependency and points-to analyses, and compile to C.

```
p1 := procs1[i]

p2 := pause(p1)

procs2 := procs1 with [i] = p2
```

Example

Let's run the dependency and points-to analyses, and compile to C.

```
p1 := procs1[i]

p2 := pause(p1)

procs2 := procs1 with [i] = p2
procs2
```

Example

Let's run the dependency and points-to analyses, and compile to C.

```
p1 := procs1[i]

p2 := pause(p1)
procs1 except [i]   p2
procs2 := procs1 with [i] = p2
procs2
```

Example

Let's run the dependency and points-to analyses, and compile to C.

```
p1 := procs1[i]
```

`procs1 except [i]` `p1`

```
p2 := pause(p1)
```

`procs1 except [i]` `p2`

```
procs2 := procs1 with [i] = p2
```

`procs2`

Example

Let's run the dependency and points-to analyses, and compile to C.

```
procs1
p1 := procs1[i]
procs1 except [i]
p1
p2 := pause(p1)
procs1 except [i]
p2
procs2 := procs1 with [i] = p2
procs2
```

Example

Let's run the dependency and points-to analyses, and compile to C.

```
procs1
```

```
p1 := procs1[i]                    p1 = &procs1[i];
```

```
procs1 except [i]   p1   p1⟼procs1[i]
```

```
p2 := pause(p1)
```

```
procs1 except [i]   p2
```

```
procs2 := procs1 with [i] = p2
```

```
procs2
```

Example

Let's run the dependency and points-to analyses, and compile to C.

`procs1`

```
p1 := procs1[i]                    p1 = &procs1[i];
```

`procs1 except [i]` `p1` `p1⟼procs1[i]`

```
p2 := pause(p1)                    pause(p1); p2 = p1;
```

`procs1 except [i]` `p2` `p2⟼procs1[i]`

```
procs2 := procs1 with [i] = p2
```

`procs2`

PROVE & RUN

Example

Let's run the dependency and points-to analyses, and compile to C.

`procs1`

```
p1 := procs1[i]                          p1 = &procs1[i];
```

`procs1 except [i]` `p1` `p1⟼procs1[i]`

```
p2 := pause(p1)                          pause(p1); p2 = p1;
```

`procs1 except [i]` `p2` `p2⟼procs1[i]`

```
procs2 := procs1 with [i] = p2           procs2 = procs1;
```

`procs2` `procs2⟼procs1`

Summary

- We have illustrated this approach on a common pattern: a read-modify-write sequence.
- This also works in more complex situations: nested updates, parallel modifications of sibling objects, ill-parenthesized modifications, etc, all of which occur in idiomatic code.
- This comes at a cost: dependency and alias tracking, rather than (just) typing.
- The cost is moderate because the problem is local to each function. Although annotations can be used to guide the generator, in practice all the programmer has to do is specify which types must be updated in place vs copied.

Conclusion

- When developing a large system like ProvenCore, most of the effort is spent on the proof.
- Choosing a language optimized for reasoning (e.g. without side effects, with support for ghost code) makes development and maintenance dramatically easier.
- With reasonably few restrictions on the executable fragment of the language, compilation techniques such as dataflow analysis work well to derive an efficient implementation.