Safe Code Generation: Metaprogramming with Type Theory

Yulong Liu & Youzhang Sun

2023-10-27

Metaprogramming and Code Generation

Metaprogramming is about writing programs that can manipulate other programs.

We will focus on code generation.

Motivation

- Code reusability
- Domain-specific languages
- Optimization

Example: C Macros

In C, gcc preprocesses macros by replacing with the content whenever a macro is used.

```c
#define square(n) ((n) * (n))

int sqsum(int a, int b) {
    return square(a + b);
}
```

gets preprocessed to

```c
int sqsum(int a, int b) {
    return ((a + b) * (a + b));
}
```

"Macros (The C Preprocessor)" https://gcc.gnu.org/onlinedocs/cpp/Macros.html

Example: Rust Generics

In Rust, `rustc` expands a generic definition into specific ones.

```
enum Option<T> { Some(T), None }

fn main() {
    let n: Option<i32> = Some(2023);
    let f: Option<f64> = Some(10.27);
}
```

gets expanded to

```
enum Option_i32 { Some(i32), None }
enum Option_f64 { Some(f64), None }

fn main() {
    let n = Option_i32::Some(2023);
    let f = Option_f64::Some(10.27);
}
```

"Generic Data Types - The Rust Programming Language"
https://doc.rust-lang.org/book/ch10-01-syntax.html#performance-of-code-using-generics

Example: Haskell Typeclasses

In Haskell, `ghc` compiles polymorphic functions by adding a dictionary that maps instances of the typeclass (types) to implementations.

```
square :: Num a => a -> a
square n = n * n
```

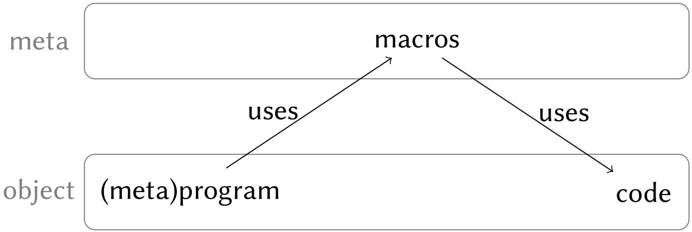
gets compiled to

```
square :: Num a -> a -> a
square d n = (*) d n n
```

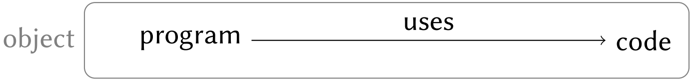
Simon Peyton Jones. "Classes, Jim, but not as we know them" https://www.cs.uoregon.edu/research/summerschool/summer13/lectures/ClassesJimOPLSS.pdf

Formalization with Type Theory

All these examples involve some code that contains metaprogramming annotations (macros, generics, etc)



The compiler evaluates them (e.g. expands macros) and substitutes with generated code.



Types are used to prevent errors in the metaprogram and provide guarantees about the generated program.

Type Theory

Let Int denote the type of integers

- Int $\longrightarrow$ Int denotes the type of functions that maps Int to Int.

Example:

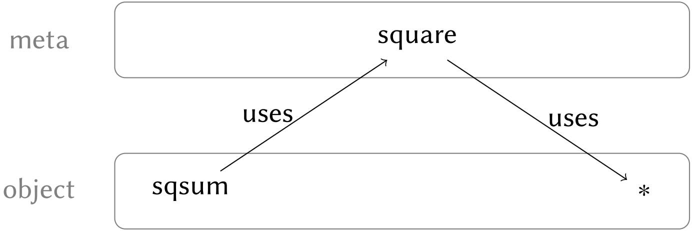
$$\mathrm{square} : \mathrm{Int} \longrightarrow \mathrm{Int}$$
$$\mathrm{square}\, n = n * n$$

$$\mathrm{sqsum} : \mathrm{Int} \longrightarrow \mathrm{Int} \longrightarrow \mathrm{Int}$$
$$\mathrm{sqsum}\, a\, b = \mathrm{square}(a + b)$$

- If n is a Int, then (square n) is a Int.
- If a and b are both Int, then (sqsum $a\,b$) is a Int

Type Theory: Metaprogramming

We can make square a macro: and let the compiler inline it in sqsum.



To distinguish between meta-level and object-level

- Meta and object level have different types
  e.g. Int in object-level is different from Int in meta-level.
- Explicit annotations are defined for interaction across levels.

Type Theory: Annotations

The object-level sqsum uses the meta-level square, so we need to pass object-level values to meta-level.

- An object-level type A can be *lifted* to meta-level:

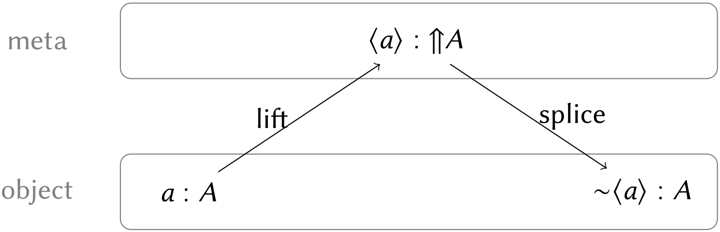
$$\Uparrow A \text{ is a meta-level type}$$

- An object-level term $a : A$ can be *quoted* to meta-level:

$$\langle a \rangle : \Uparrow A$$

- An quoted term $\langle a \rangle : \Uparrow A$ can be *spliced* back to object-level:

$$\sim\langle a \rangle : A$$



Example: Applying Annotations

$$
\begin{aligned}
(*) &: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
(+) &: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
\mathsf{square} &: \Uparrow\mathsf{Int} \to \Uparrow\mathsf{Int} \\
\mathsf{square}\ n &= \langle(\sim n) * (\sim n)\rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{sqsum} &: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
\mathsf{sqsum}\ a\ b &= \sim(\mathsf{square}\ \langle a + b\rangle)
\end{aligned}
$$

Example: Compilation

$$\begin{aligned}
&\mathsf{square} : \Uparrow\mathsf{Int} \longrightarrow \Uparrow\mathsf{Int} \\
&\mathsf{square}\; n = \langle (\sim n) * (\sim n) \rangle \\
&\mathsf{sqsum} : \mathsf{Int} \longrightarrow \mathsf{Int} \longrightarrow \mathsf{Int} \\
&\mathsf{sqsum}\; a\; b = \sim(\mathsf{square}\; \langle a + b \rangle)
\end{aligned}$$

The compiler would evaluate sqsum as follows:

$$\begin{aligned}
\mathsf{sqsum}\; a\; b &= \sim(\mathsf{square}\; \langle a + b \rangle) \\
&= \sim\langle (\sim\langle a + b \rangle) * (\sim\langle a + b \rangle) \rangle \\
&= \sim\langle (a + b) * (a + b) \rangle \\
&= (a + b) * (a + b)
\end{aligned}$$

And so the final result of evaluation is:

$$\begin{aligned}
&\mathsf{sqsum} : \mathsf{Int} \longrightarrow \mathsf{Int} \longrightarrow \mathsf{Int} \\
&\mathsf{sqsum}\; a\; b = (a + b) * (a + b)
\end{aligned}$$

Going Deeper

Formalizing annotations: inference rules

$$\frac{\Gamma \vdash_0 A}{\Gamma \vdash_1 \Uparrow A}\;\textsc{lift} \qquad \frac{\Gamma \vdash_0 t : A}{\Gamma \vdash_1 \langle t \rangle : \Uparrow A}\;\textsc{quote} \qquad \frac{\Gamma \vdash_1 t : \Uparrow A}{\Gamma \vdash_0 \sim t : A}\;\textsc{splice}$$

Type inference: preservation laws

$$\Uparrow(A \to B) \cong \Uparrow A \to \Uparrow B$$

Dependent types: Martin-Löf Type Theory

$$(A : U) \to (a : A) \to B\,a$$

Staged compilation: Two-Level Type Theory, Substitution Calculus

References

[TS97] Walid Taha and Tim Sheard. “MetaML and Multi-Stage Programming with Explicit Annotations”. In: *SIGPLAN Not.* 32.12 (Dec. 1997), pp. 203217. ISSN: 0362-1340. DOI: 10.1145/258994.259019. URL: https://dl.acm.org/doi/10.1145/258994.259019.

[Ann+19] Danil Annenkov et al. “Two-Level Type Theory and Applications”. In: *ArXiv e-prints* (May 2019). URL: http://arxiv.org/abs/1705.03307.

[Kov22] András Kovács. “Staged Compilation with Two-Level Type Theory”. In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: 10.1145/3547641. URL: https://doi.org/10.1145/3547641.

Thank you!