STAMP: Strongly Type-safe Meta-Programming

Thomas Winant    Dominique Devriese
Jesper Cockx

DistriNet – KU Leuven

21 September 2015
Type-safe metaprogramming:
Overview

Most metaprogramming is weakly type-safe (e.g. Template Haskell)
- generated programs may contain type errors
- type checker checks generated code

With Agda as metalanguage, we can do better:
- embed Haskell type system in Agda
- generated code type correct by construction
STAMP: Strongly Type-safe Meta-Programming

1. Why STAMP?
2. Examples
3. The STAMP architecture
4. The Agda encoding
STAMP: Strongly Type-safe Meta-Programming

1 Why STAMP?
2 Examples
3 The STAMP architecture
4 The Agda encoding
Why strongly type-safe metaprogramming?

- we cannot test all possible pieces of code generated by a metaprogram
- type errors in generated code are impossible to debug by the user
- types document what can be expected of the metaprogram
Why use Agda instead of a special-purpose metalanguage?

We can generate both the type and the typing context of the metaprogram together with the program itself.
STAMP: Strongly Type-safe Meta-Programming

1. Why STAMP?
2. Examples
3. The STAMP architecture
4. The Agda encoding
Pick k’th from n function arguments

Given k and n, generate the following definition

\[
pick :: a_1 \to \ldots \to a_n \to a_k \\
pick \ x_1 \ \ldots \ \ x_n = x_k
\]
Automatic deriving

- Derive Eq
- Derive lenses
STAMP: Strongly Type-safe Meta-Programming

1. Why STAMP?
2. Examples
3. The STAMP architecture
4. The Agda encoding
The STAMP architecture

Added syntax to Haskell to make a STAMP call

STAMP works as a Core2Core plugin
- call corresponding Agda metaprogram
- translate Agda representation to Haskell Core
- splice generated code into the right position
Current shortcoming

- calls to Haskell functions in generated code are not checked
- type error only after translation to core
- solution: need to generate Agda interface based on Haskell code
STAMP: Strongly Type-safe Meta-Programming

1. Why STAMP?
2. Examples
3. The STAMP architecture
4. The Agda encoding
The Agda encoding

Fairly standard encoding of System $F_C$

- Kinds
- Types depend on kinds
- Terms depend on types
Weakening and substitution

- definition of Term datatype requires weakening and substitution of types
- we take

\[
\text{TySubst } \Sigma_1 \Sigma_2 = \text{All } (\text{Type } \Sigma_2) \Sigma_1
\]
Design based on Haskell documentation

- good to verify correctness w.r.t. Haskell specification
- not very convenient for writing metaprogams (substitution hell)
- based on our experiences now, we hope to add a convenience layer later