An Executable Formal Semantics of Agda

Andrej Tokarčík, supervised by Mgr. Jan Obdržálek, PhD.
Faculty of Informatics, Masaryk University, Brno

Agda

Agda is an actively developed dependently typed programming language. Its types can directly *depend* on values: it is, for instance, possible to define a function returning the *n*-th element of a list so that the typechecker itself guarantees the list to have at least *n* elements.

```
data Vec (A : Set) : ℕ → Set where
  nil  :                     Vec A  zero
  cons : {n : ℕ} → A → Vec A n → Vec A (succ n)

data Fin : ℕ → Set where
  fzero : {n : ℕ} → Fin (succ n)
  fsucc : {n : ℕ} → Fin n → Fin (succ n)

_[_] : {A : Set} {n : ℕ} →
       Vec A n →
       Fin n → A
<>         [ () ]
(x , _)    [ fzero ]   = x
(_ , xs)   [ fsucc i ] = xs [ i ]
```

Thanks to its rich and expressive type system, Agda can also serve as an interactive theorem prover. Types correspond to logical formulæ whereas values represent formal proofs of their type/formula.

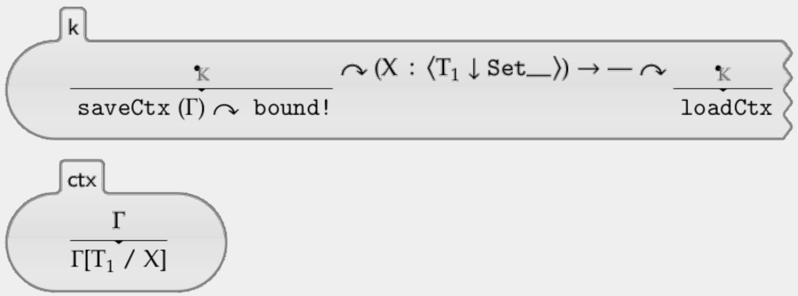
$\mathbb{K}$ Framework

$\mathbb{K}$ is a semantic framework in which formal semantics of programming languages can be specified in terms of rewriting rules and data configurations.

It provides a variety of generic, practical tools that can be used with any language defined in $\mathbb{K}$, such as parsers, interpreters, symbolic execution engines, semantic debuggers, test-case generators, state-space explorers and model checkers. Immediate availability of these tools makes $\mathbb{K}$ specifications genuinely *executable*.

Several real-world languages have been already defined in $\mathbb{K}$, including C, Java, Python and Javascript.

RULE BIND-Π



The Thesis

We successfully specified a formal semantics of Agda using the $\mathbb{K}$ semantic framework.

Challenges

Until this work, no dependently typed language has been formalised in $\mathbb{K}$, and Agda had no proper semantic description.

How should the essential typechecking and type inference algorithms be implemented?

How should declarations of parametrised datatypes, inductively defined families and dependent functions be processed and stored?

How should metavariables (implicit arguments) be inferred and inserted?

How should pattern matching with inductive families be realised?

- We created an executable $\mathbb{K}$ semantics of Agda that addresses these questions.

Our Contribution

- We provided a discussion of issues related to formalisation of Agda.
- We implemented the first formal semantics of (a substantive portion of) Agda.
- We created the first $\mathbb{K}$ semantics of a dependently typed language.

The work demonstrates the ability to provide operational semantics of dependently typed programming languages without disregarding those hard-to-formalise aspects that make their use practical.

$$\frac{\begin{array}{cc}\Gamma \vdash e_1 \downarrow S_1 \rightsquigarrow T_1 & \Gamma, x : T_1 \vdash e_2 \downarrow S_2 \rightsquigarrow T_2 \\ S_1 \to_{whnf} \mathtt{Set}_\alpha & S_2 \to_{whnf} \mathtt{Set}_\beta\end{array}}{\Gamma \vdash (x : e_1) \to e_2 \downarrow \mathtt{Set}_{\alpha \sqcup \beta} \rightsquigarrow (x : T_1) \to T_2}$$