Correct-by-construction record typechecking

1. Introduction

Motivation

Typecheckers are important to software development, helping catch bugs early. They are, however, vulnerable to bugs themselves, which causes their implementation to diverge from the theoretical specification.

Correct-by-construction programming

Agda is a programming language which treats types as mathematical propositions. Its robust typechecker allows us to create programs with proofs about their functionality by incrementally refining a small set of rules. This approach is called **correctness-by-construction**.

Goal

- Create a correct-by-construction typechecker with **records** and **subtyping** and prove its soundness and completeness
- Compare the result to other approaches

```
{
  id = 42,
  name = "Arthur"
} : {
  id : Integer,
  name : String
}
```

Figure 1: Record type and value

```
{
  id : Natural,
  name : String
} <: {
  id : Integer,
  name : String
}
```

Figure 2: Depth subtyping

```
{
  id : Integer,
  name : String
} <: {
  name : String
}
```

Figure 3: Width subtyping

2. Implementation

The formal model of the typechecker's language is adapted from Pierce [1].

The typechecker makes judgements based on **typing** and **subtyping rules**. They dictate whether a term is well-typed and whether one type is a subtype of another.

The typechecker is implemented in two steps: first, the terms, types and rules are transcribed as datatypes; then, functions for deciding the subtype relation, type inference and typechecking are constructed based on the rules.

Transcribing the specification

Two typing rules concern records—one allows for their creation and the other lets us retrieve a value from a field.

```
data _⊢_::_ (Γ : Map Type) : Term → Type → Set where
  ⊢rec-empty : Γ ⊢ rec [] :: Rec []
  ⊢rec-more : ∀ {x} {v} {A} {rs : Map Term} {rt : Map Type}
    → ¬ (x ∈ₖ′ rt) → Γ ⊢ rec rs :: Rec rt
    → Γ ⊢ v :: A → Γ ⊢ rec ((x , v) ∷ rs) :: Rec ((x , A) ∷ rt)
```

The rule for function application needs to be changed to accommodate subtypes.

```
⊢· : ∀ {u v} {A B C} → Γ ⊢ u :: (A ⇒ B)
  → Γ ⊢ v :: C → C <: A
  → Γ ⊢ u · v :: B
```

Depth and width subtyping can be combined into one subtyping rule for records.

```
data _<:_ : Rel Type 0ℓ where
  <:rec : {m n : Map Type} → n ⊆′ m
    → All (λ { (k , v) → (i : k ∈ₖ′ m) → lookup′ i <: v}) n
    → Rec m <: Rec n
```

Typechecking and inference

The three typechecking functions all make use of Agda's `Dec` structure, which consists of a boolean value stating whether a proposition holds and a proof reflecting said value.

```
_<:?_ : ∀ {S T} → Dec (S <: T)
infer : ∀ (Γ : Map) u → Dec (Σ[ t ∈ Type ] Γ ⊢ u :: t)
check : ∀ (Γ : Map) u (t : Type) → Dec (Σ[ s ∈ Type ] (s <: t × Γ ⊢ u :: s))
```

Returning the relevant typing rule when the functions accept a program guarantees the typechecker is **sound**. Conversely, returning a proof that no rule can lead to rejected input ensures the typechecker's **completeness**.

4. Advantages and disadvantages

- The typechecker is **more trustworthy** than unverified typechecker
  - Typescript has over 1500 bugs, many related to typechecking
  - Rust has over 80 soundness holes
- Agda requires **additional assurances** in seemingly trivial cases
- Agda's proof assistance features are sometimes **buggy**
- Correct-by-construction programming presents **unique challenges** and is overall **more complex** than creating unverified programs
  - Creating the typechecker took 41 days, while a version without soundness and completeness proofs took only 2 hours

5. Conclusions

Correct-by-construction typecheckers are guaranteed to be consistent with their specification, making them suitable for critical applications. On the other hand, due to a combination of less-than-ideal tooling and intrinsic complexity, the technique takes significantly more time and effort, making it less suitable for general use.

Full code of the typechecker can be found in the repository linked in the QR code in Figure 5.

Figure 5: Repository link

Future Work

- Investigate how the complexity of a verified typechecker increases with its scope
- Compare the performance of a correct-by-construction typechecker to other approaches
- Improve Agda's unification and constraint-solving algorithms
- Create quality-of-life features for Agda developers, such as code completion and better documentation

6. References

[1] B. C. Pierce, *Types and programming languages*, English. Cambridge, Mass.: MIT Press, 2002. [Online]. Available: http://www.books24x7.com/marc.asp?isbn=0262162091.

Kazimierz Ciaś
k.j.cias@student.tudelft.nl

TU Delft

Professor: **Jesper Cockx**
Supervisor: **Sára Juhošová**