# Equationally Correct Semantics (Extended Abstract)

Cameron Wong
Jane Street/Harvard University
USA

***CCS Concepts:*** • **Theory of computation → Logic and verification**.

***Keywords:*** dependent types, soundness, language semantics

**ACM Reference Format:**
Cameron Wong. 2022. Equationally Correct Semantics (Extended Abstract). In *Proceedings of* . ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

We developed a new technique for systemically deriving type-safe small-step operational semantics automatically satisfying progress and preservation with respect to a typing algorithm. All proofs and derivations have been mechanised in the Agda proof assistant.

Our technique is an adaptation of the one pioneered by Bahr and Hutton [1] for computing compilers. Following the example of Pickard and Hutton [3], we choose a dependently-typed setting to avoid troublesome partiality issues. We will first describe how type soundness can be phrased as an equation, then use that equation to derive a small-step operational semantics for a simple expression language.

## 2 Type Safety as an Equation

We seek to define a runtime semantics for a language, which we quantify as the function `step`. Our first order of business is to phrase our correctness condition as an equation relating `step` to our other desired quantities, namely, type soundness.

The classic statement of type soundness is the twin theorems of progress and preservation [2]. Colloquially, progress states that "well-typed programs do not get stuck", and preservation states that "a program has the same type after each evaluation step". The latter seems like a promising equation candidate, as it is a statement *equating* two things — namely, the type of an expression before and after each step.


ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We specify the target language by converting its typing judgment $\Gamma \vdash e : \tau$ into an Agda function

$$\texttt{typeof} : \texttt{Exp} \longrightarrow \texttt{Maybe Type}$$

with the property $\texttt{typeof}\ e = \tau$ iff $\emptyset \vdash e : \tau$.

Next, consider syntactic values, which cannot be stepped. Typically, this is expressed by having the `step` function return a partial value such as a `Maybe`. In an equation, however, we will have to branch on the result, which becomes unwieldy. Instead, we parameterize the `Exp` datatype by whether it can be stepped.

```
data Steppable : Set where
  Value: Steppable
  Steps: Steppable

data Exp : Steppable → Set where
  ...

step : (e: Exp Steps) → ∃S.Exp S
```

Notice that `step` now returns an existential `Exp` $S$, as we cannot know whether the result can be evaluated further.

There is a further issue of attempting to step ill-typed programs. In the mechanisation, this is addressed by further amending `step` to also take a proof that its argument is well-typed. As Agda enforces that functions are total, such a function actually serves as a proof of the progress theorem. This obscures the process, however, so we will elide it from the type of `step` and instead merely include it as an assumption.

All the pieces are in place, then, to relate progress (`step`) to the typing judgment (`typeof`) via the *preservation equation*:

$$\texttt{typeof}\ e = \texttt{typeof (step e)} \tag{1}$$

where $e : \texttt{Exp CanStep}$.

## 3 The Derivation

### 3.1 Target Language

Our target language for this demonstration is the simple, typed expression language presented in Figure 1[1].

[1]We do not use the usual dependently-typed technique of parameterizing `Exp` with its type, as it would trivialize the `typeof` function

[2]The actual implementation in Agda is somewhat more complex, and is simplified for presentation. Agda does not support Haskell-style `case` expressions, nor can it, in general, decide equality or inequality of `Set`s. Instead, we use the usual `fold` operator over the `Maybe` type and use a regular Agda variant to represent ℕ and `Bool`.

```
data Exp : Steppable → Set where
  boolVal : Bool → Exp Value
  intVal : ℕ → Exp Value
  add : Exp S → Exp Steps
  if_ : Exp Sn → Exp S1 → Exp S2 → Exp Steps

data ⊢_:_ : Exp S → Type → Set where
  typ-bool : ⊢(boolVal b):Bool
  typ-nat : ⊢(intVal i):ℕ
  typ-add : ⊢ e1 : ℕ →⊢ e2 : ℕ →⊢ (add e1 e2) : ℕ
  typ-if : ⊢ e : Bool →⊢ e1 : τ →⊢ e2 : τ →⊢ (if e e1 e2) : τ
```

**Figure 1.** Target language and typing rules

```
typeof (add e1 e2) =
  case (typeof e1, typeof e2)
    of (Just ℕ, Just ℕ) -> Just ℕ
     | _ -> Nothing
typeof (if_ e e1 e2) =
  case (typeof e, typeof e1, typeof e2)
    of (Just Bool, Just τ1, Just τ2) ->
         if τ1 = τ2
           then Just τ1
           else Nothing
     | _ -> Nothing
```

**Figure 2.** Definition of `typeof`, selected cases [2]

Our goal is to define the function `step` satisfying equation 1. As per Bahr and Hutton [1], we will proceed by structural induction on $e$, evaluating the left hand side of equation 1 and seek to transform it into an expression of the form `typeof` $c$, then take `step e` $= c$ as a definition for that case of `step`.

### 3.2 Semantics Calculation

Let $\vdash e : \tau$. For brevity, we show only two representative cases.

**Case:** $e$ = add $e_1$ $e_2$, where $e_1$ : `Exp CanStep`

We begin by applying the definition of `typeof` from Figure 2:

```
typeof (add e1 e2)
  = ⟨definition of typeof⟩
case (typeof e1, typeof e2) ...
```

We are immediately stuck, as we cannot expand `typeof` $e_1$ any further. To proceed, we have no choice but to cite the inductive hypothesis:

```
case (typeof e1, typeof e2) ...
  = ⟨inductive hypothesis on typeof e1⟩
case (typeof (step e1), typeof e2) ...
```

We finally apply the definition of `typeof` in reverse:

```
case (typeof (step e1), typeof e2) ...
  = ⟨definition of typeof⟩
typeof (add (step e1) e2)
```

This is now of the form `typeof`(add $e_1$ $e_2$) = `typeof` $c$, namely, $c$ = add (`step` $e_1$) $e_2$. We wrap up by defining `step` for this case:

```
step (add e1 e2) = add (step e1) e2
```

We note that this rule specifies "left-first" evaluation semantics. In fact, if both $e_1$ and $e_2$ can be stepped, we have the choice of invoking the inductive hypothesis on $e_1$, $e_2$ or both, corresponding to left-first, right-first or parallel evaluation respectively.

**Case:** $e$ = `if_` (`boolVal true`) $e_1$ $e_2$

As with before, we begin by expanding `typeof`:

```
typeof (if_ (boolVal true) e1 e2)
  = ⟨definition of typeof⟩
case ...
  of (Just Bool, Just τ1, Just τ2) ->
       if_ τ1 = τ2
         then Just τ1
         else Nothing
     ...
```

We are once again stuck. Unlike before, we have made no assumptions about whether $e_1$ or $e_2$ are steppable, so we cannot cite the inductive hypothesis.

By inversion on the assumption $\vdash e : \tau$, we can conclude that $\vdash e_1 : \tau$ and $\vdash e_2 : \tau$, and thus `typeof`$e_1$ = `typeof`$e_2$ = $\tau$. Then:

```
case ...
  of (Just Bool, Just τ1, Just τ2) ->
       if_ τ1 = τ2
         then Just τ1
         else Nothing
     ...
  = ⟨assumption⟩
if τ = τ then Just τ else Nothing
  = ⟨evaluation step⟩
Just τ
  = ⟨assumption⟩
typeof e1
```

Here, we needed to make a human judgment of which of $e_1$ or $e_2$ to evaluate to.

### 3.3 Implementation

The example language semantics, along with their proofs of correctness, have been fully mechanised in Agda, available at `https://github.com/CT075/calculated-semantics`. This includes the unsimplified `step` fully witnessing progress and a proof that the `typeof` function respects the static typing rules.

## 4 Reflection and Future Work

In summary, we have seen that elementary, equational reasoning can be used to discover an operational semantics for a type system. As with Bahr and Hutton [1], our proof of soundness falls out of the derivation process.

An unsatisfying part of the derivation is that, ultimately, it requires a human to make decisions. For example, choosing which branch of the `if_` variant corresponds to `true`. This has consequences on evaluation order, as previously noted, but also on correctness. Consider that, as presented, there is nothing associating the add variant with the (+) operator on naturals beyond our human intuition, which presents an obstacle to fully automating this process. We hope that the addition of a further language specification, such as a denotational semantics, may help resolve this.

A logical next step would be to apply this technique to a language with a more sophisticated typing algorithm. As a first step in this direction, we hope to derive a semantics for System $F_{\omega}$, the higher-kinded polymorphic lambda calculus.

## References

[1] Bahr, P., and Hutton, G. Calculating Correct Compilers. *Journal of Functional Programming 25* (Sept. 2015).

[2] Harper, R. *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press, 2016.

[3] Pickard, M., and Hutton, G. Calculating Dependently-Typed Compilers. *Proceedings of the ACM on Programming Languages 5*, ICFP (Aug. 2021).