Programming Language Foundations in Agda

Philip Wadler

University of Edinburgh
wadler@inf.ed.ac.uk

Abstract. The leading textbook for formal methods is *Software Foundations* (SF), written by Benjamin Pierce in collaboration with others, and based on Coq. After five years using SF in the classroom, I have come to the conclusion that Coq is not the best vehicle for this purpose, as too much of the course needs to focus on learning tactics for proof derivation, to the cost of learning programming language theory. Accordingly, I have written a new textbook, *Programming Language Foundations in Agda* (PLFA). PLFA covers much of the same ground as SF, although it is not a slavish imitation.
What did I learn from writing PLFA? First, that it is possible. One might expect that without proof tactics that the proofs become too long, but in fact proofs in PLFA are about the same length as those in SF. Proofs in Coq require an interactive environment to be understood, while proofs in Agda can be read on the page. Second, that constructive proofs of preservation and progress give immediate rise to a prototype evaluator. This fact is obvious in retrospect but it is not exploited in SF (which instead provides a separate normalise tactic) nor can I find it in the literature. Third, that using raw terms with a separate typing relation is far less perspicuous than using inherently-typed terms. SF uses the former presentation, while PLFA presents both; the former uses about 1.6 as many lines of Agda code as the latter, roughly the golden ratio.
The textbook is written as a literate Agda script, and can be found here:

http://plfa.inf.ed.ac.uk

Keywords: Agda · Coq · lambda calculus · dependent types.

1 Introduction

The most profound connection between logic and computation is a pun. The doctrine of Propositions as Types asserts that a certain kind of formal structure may be read in two ways: either as a proposition in logic or as a type in computing. Further, a related structure may be read as either the proof of the proposition or as a programme of the corresponding type. Further still, simplification of proofs corresponds to evaluation of programs.

Accordingly, the title of this paper, and the corresponding textbook, *Programming Language Foundations in Agda* (hence, PLFA) also has two readings. It may be parsed as "(Programming Language) Foundations in Agda" or "Programming (Language Foundations) in Agda"—the specifications in the proof assistant Agda both describe programming languages and are themselves programmes.

Since 2013, I have taught a course on Types and Semantics for Programming Languages to fourth-year undergraduates and masters students at the University of Edinburgh. An earlier version of that course was based on *Types and Programming Languages* by Pierce and Benjamin (2002), but my version was taught from its successor, *Software Foundations* (hence, SF) by Pierce et al. (2010), which is based on the proof assistance Coq (Huet et al. 1997). I am convinced by the claim of Pierce (2009), made in his ICFP Keynote *Lambda, The Ultimate TA*, that basing a course around a proof assistant aids learning.

However, after five years of experience, I have come to the conclusion that Coq is not the best vehicle. Too much of the course needs to focus on learning tactics for proof derivation, to the cost of learning the fundamentals of programming language theory. Every concept has to be learned twice: e.g., both the product data type, and the corresponding tactics for introduction and elimination of conjunctions. The rules Coq applies to generate induction hypotheses can sometimes seem mysterious. While the `notation` construct permits pleasingly flexible syntax, it can be confusing that the same concept must always be given two names, e.g., both `subst N x M` and `N [x := M]`. Names of tactics are sometimes short and sometimes long; naming conventions in the standard library can be wildly inconsistent. *Propositions as types* as a foundation of proof is present but hidden.

I found myself keen to recast the course in Agda (Bove et al. 2009). In Agda, there is no longer any need to learn about tactics: there is just dependently-typed programming, plain and simple. Introduction is always by a constructor, elimination is always by pattern matching. Induction is no longer a mysterious separate concept, but corresponds to the familiar notion of recursion. Mixfix syntax is flexible while using just one name for each concept, e.g., substitution is `_[_:=_]`. The standard library is not perfect, but there is a fair attempt at consistency. *Propositions as types* as a foundation of proof is on proud display.

Alas, there is no textbook for programming language theory in Agda. *Verified Functional Programming in Agda* by (Stump 2016) covers related ground, but focusses more on programming with dependent types than on the theory of programming languages.

The original goal was to simply adapt *Software Foundations*, maintaining the same text but transposing the code from Coq to Agda. But it quickly became clear to me that after five years in the classroom I had my own ideas about how to present the material. They say you should never write a book unless you cannot *not* write the book, and I soon found that this was a book I could not not write.

I am fortunate that my student, Wen Kokke, was keen to help. She guided me as a newbie to Agda and provided an infrastructure that is easy to use and produces pages that are a pleasure to view. The bulk of the book was written January–June 2018, while on sabbatical in Rio de Janeiro.

This paper is a personal reflection, summarising what I learned in the course of writing the textbook. Some of it reiterates advice that is well-known to some members of the dependently-typed programming community, but which deserves to be better known. The paper is organised as follows.

Section 2 outlines the topics covered in PLFA, and notes what is omitted.

Section 3 compares Agda and Coq as vehicles for pedagogy. Before writing the book, it was not obvious that it was even possible; conceivably, without tactics some of the proofs balloon in size. In fact, it turns out that for the results in PLFA and SF, the proofs are of roughly comparable size, and (in my opinion) the proofs in PLFA are more readable and have a pleasing visual structure.

Section 4 observes that constructive proofs of progress and preservation combine trivially to produce a constructive evaluator for terms. This idea is obvious once you have seen it, yet I cannot find it described in the literature. For instance, SF separately implements a `normalise` tactic that has nothing to do with progress and preservation.

Section 5 claims that raw terms should be avoided in favour of inherently-typed terms. PLFA develops lambda calculus with both raw and inherently-typed terms, permitting a comparison. It turns out the former is less powerful—it supports substitution only for closed terms—but significantly longer—about 1.6 times as many lines of code, roughly the golden ratio.

I will argue that Agda has advantages over Coq for pedagogic purposes. My focus is purely on the case of a proof assistant as an aid to *learning* formal semantics using examples of *modest* size. I admit up front that there are many tasks for which Coq is better suited than Agda. A proof assistant that supports tactics, such as Coq or Isabelle, is essential for formalising serious mathematics, such as the Four-Colour Theorem (Gonthier 2008), the Odd-Order Theorem (Gonthier et al. 2013), or Kepler's Conjecture (Hales et al. 2017), or for establishing correctness of software at scale, as with the CompCert compiler (Leroy 2009; Kästner et al. 2017) or the SEL4 operating system (Klein et al. 2009; O'Connor et al. 2016).

2 Scope

PLFA is aimed at students in the last year of an undergraduate honours programme or the first year of a master or doctorate degree. It aims to teach the fundamentals of operational semantics of programming languages, with simply-typed lambda calculus as the central example. The textbook is written as a literate script in Agda. As with SF, the hope is that using a proof assistant will make the development more concrete and accessible to students, and give them rapid feedback to find and correct misaprehensions.

The book is broken into two parts. The first part, Logical Foundations, develops the needed formalisms. The second part, Programming Language Foundations, introduces basic methods of operational semantics. (SF is divided into books, the first two of which have the same names as the two parts of PLFA, and cover similar material.)

Each chapter has both a one-word name and a title, the one-word name being both its module name and its file name.

Part I, Logical Foundations

Naturals: Natural numbers Introduces the inductive definition of natural numbers in terms of zero and successor, and recursive definitions of addition, multiplication, and monus. Emphasis is put on how a tiny description can specify an infinite domain.

Induction: Proof by induction Introduces induction to prove properties such as associativity and commutativity of addition. Also introduces dependent functions to express universal quantification. Emphasis is put on the correspondence between induction and recursion.

Relations: Inductive definitions of relations Introduces inductive definitions of less than or equal on natural numbers, and odd and even natural numbers. Proves properties such as reflexivity, transitivity, and anti-symmetry, and that the sum of two odd numbers is even. Emphasis is put on proof by induction over evidence that a relation holds.

Equality: Equality and equational reasoning Gives Martin Löf's and Leibniz's definitions of equality, and proves them equivalent, and defines the notation for equational reasoning used throughout the book.

Isomorphism: Isomorphism and embedding Introduces isomorphism, which plays an important role in the subsequent development. Also introduces dependent records, lambda terms, and extensionality.

Connectives: Conjunction, disjunction, and implication Introduces product, sum, unit, empty, and function types, and their interpretations as connectives of logic under Propositions as Types. Emphasis is put on the analogy between these types and product, sum, unit, zero, and exponential on naturals; e.g., product of numbers is commutative and product of types is commutative up to isomorphism.

Negation: Negation, with intuitionistic and classical logic Introduces logical negation as a function into the empty type, and explains the difference between classical and intuitionistic logic.

Quantifiers: Universals and existentials Recaps universal quantifiers and their correspondence to dependent functions, and introduces existential quantifiers and their correspondence to dependent products.

Lists: Lists and higher-order functions Gives two different definitions of reverse and proves them equivalent. Introduces map and fold and their properties, including that fold left and right are equivalent in a monoid. Introduces predicates that hold for all or any member of a list, with membership as a specialisation of the latter.

Decidable: Booleans and decision procedures Introduces booleans and decidable types, and why the latter is to be preferred to the former.

Part II, Programming Language Foundations

Lambda: Introduction to lambda calculus Introduces lambda calculus, using a representation with named variables and a separate typing relation. The language used is PCF, with variables, lambda abstraction, application, zero, successor, case over naturals, and fixpoint. Reduction is call-by-value and restricted to closed terms.

Properties: Progress and preservation Proves key properties of simply-typed lambda calculus, including progress and preservation. Progress and preservation are combined to yield an evaluator.

DeBruijn: Inherently typed de Bruijn representation Introduces de Bruijn numbers and the inherently-typed representation. Emphasis is put on the structural similarity between a term and its corresponding type derivation; in particular, de Bruijn numbers correspond to the judgment that a variable is well-typed under a given environment.

More: More constructs of simply-typed lambda calculus Introduces product, sum, unit, and empty types as well as lists and let bindings are explained. Typing and reduction rules are given informally; a few are then give formally, and the rest are left as exercises for the reader. The inherently typed representation is used.

Bisimulation: Relating reduction systems Shows how to translate the language with "let" terms to the language without, representing a let as an application of an abstraction, and shows how to relate the source and target languages with a bisimulation.

Inference: Bidirectional type inference Introduces bidirectional type inference, and applies it to convert from a representation with named variables and a separate typing relation to a representation de Bruijn indices with inherent types.

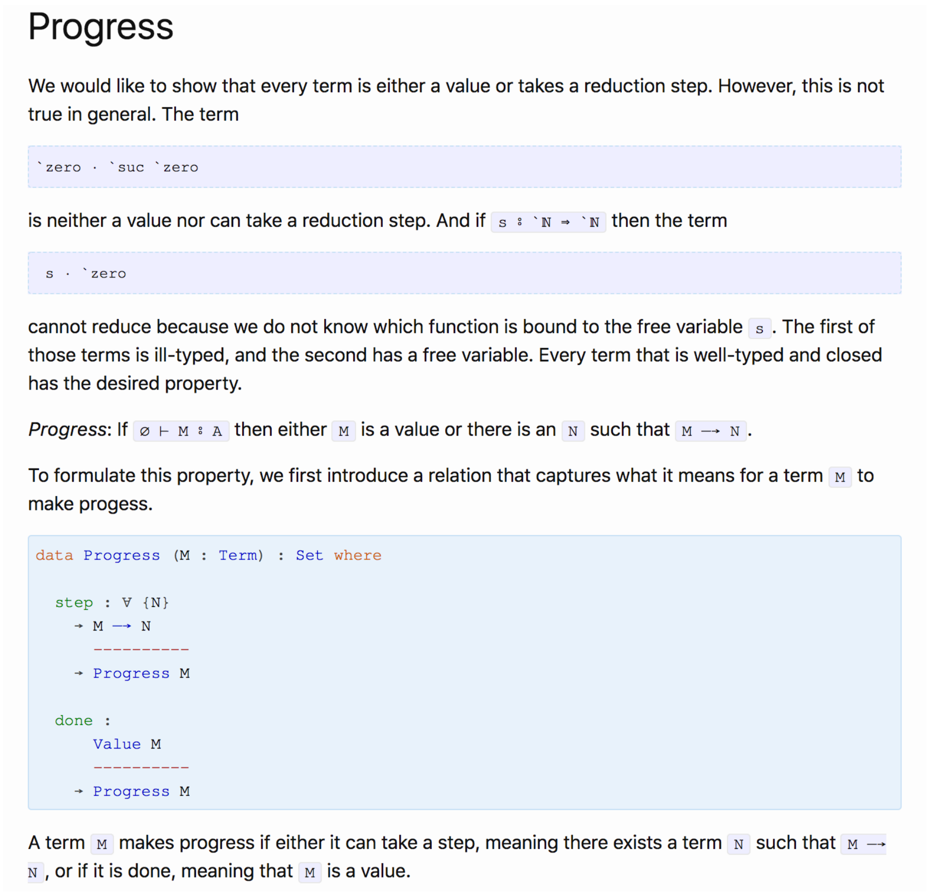
Untyped: Untyped calculus with full normalisation As a variation on earlier themes, discusses an untyped (but inherently scoped) lambda calculus. Reduction is call-by-name over open terms, with full normalisation (including reduction under lambda terms). Emphasis is put on the correspondence between the structure of a term and evidence that it is in normal form.

Discussion

PLFA and SF differ in several particulars. PLFA begins with a computationally complete language, PCF, while SF begins with a minimal language, simply-typed lambda calculus with booleans. PLFA does not include type annotations in terms, and uses bidirectional type inference, while SF has terms with unique types and uses type checking. SF also covers a simple imperative language with Hoare logic, subtyping, record types, mutable references, and normalisation none of which are treated by PLFA. PLFA covers an inherently-typed de Bruijn representation, bidirectional type inference, bisimulation, and an untyped call-by-value language with full normalisation, none of which are treated by SF.

SF has a third volume, written by Andrew Appel, on Verified Functional Algorithms. I'm not sufficiently familiar with that volume to have a view on whether it would be easy or hard to cover that material in Agda.

There is more material that would be desirable to include in PLFA which was not due to limits of time. In future years, PLFA may be extended to cover additional material, including mutable references, normalisation, System F, pure type systems, and denotational semantics. I'd especially like to include pure type systems as they provide the readers with a formal model close to the dependent types used in the book. My attempts so far to formalise pure type systems have proved challenging.



Progress

We would like to show that every term is either a value or takes a reduction step. However, this is not true in general. The term

```
`zero · `suc `zero
```

is neither a value nor can take a reduction step. And if `s ⦂ `ℕ ⇒ `ℕ` then the term

```
s · `zero
```

cannot reduce because we do not know which function is bound to the free variable `s`. The first of those terms is ill-typed, and the second has a free variable. Every term that is well-typed and closed has the desired property.

Progress: If `∅ ⊢ M ⦂ A` then either `M` is a value or there is an `N` such that `M —→ N`.

To formulate this property, we first introduce a relation that captures what it means for a term `M` to make progess.

```
data Progress (M : Term) : Set where

  step : ∀ {N}
    → M —→ N
      ----------
    → Progress M

  done :
      Value M
      ----------
    → Progress M
```

A term `M` makes progress if either it can take a step, meaning there exists a term `N` such that `M —→ N`, or if it is done, meaning that `M` is a value.

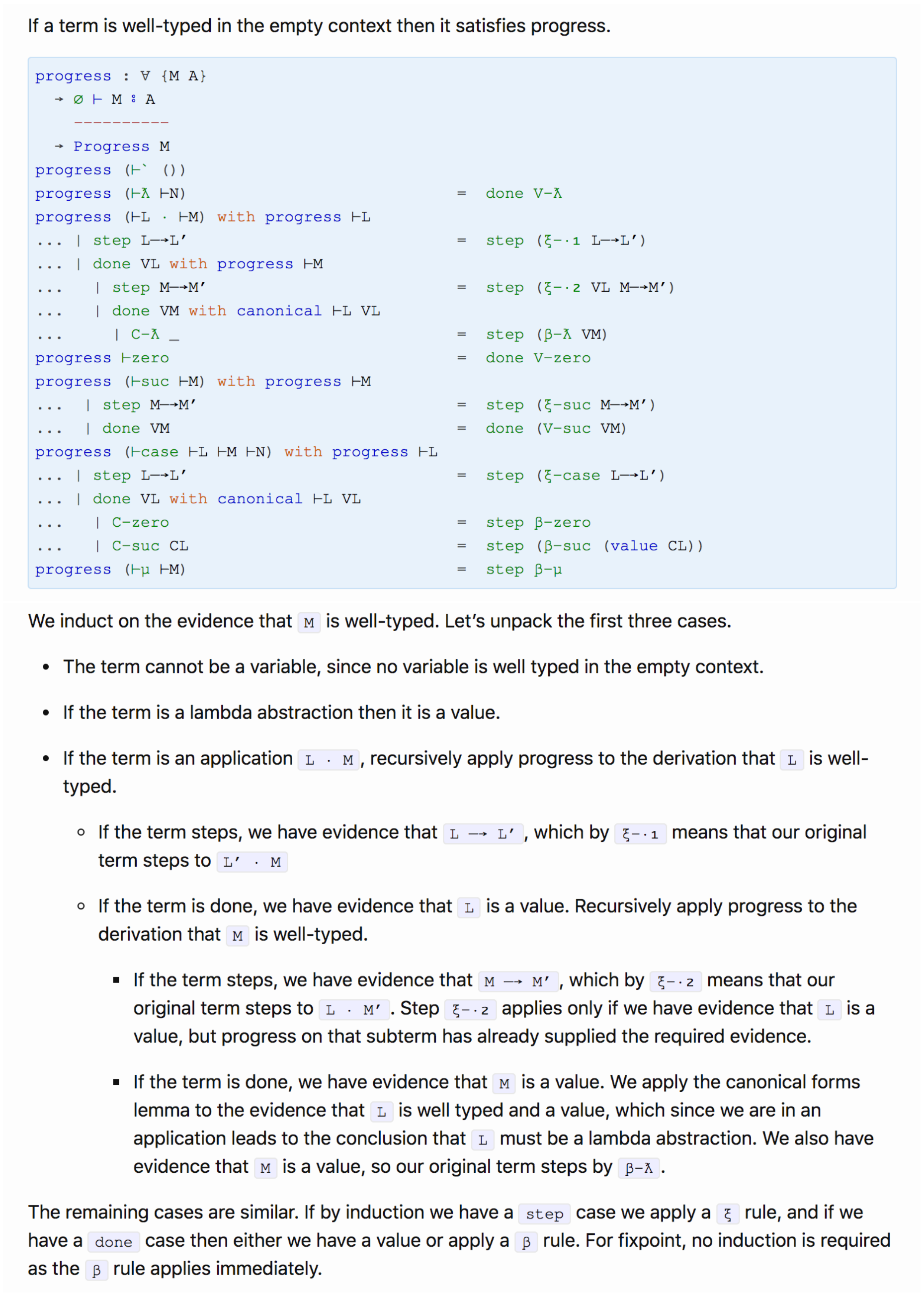
Fig. 1. PLFA, Progress (1/2)

3 Proofs in Agda and Coq

The introduction listed several reasons for preferring Agda over Coq. But Coq tactics enable more compact proofs. Would it be possible for PLFA to cover the same material as SF, or would the proofs balloon to unmanageable size?

As an experiment, I first rewrote SF's development of simply-typed lambda calculus (SF, Chapters Stlc and StlcProp) in Agda. I was a newbie to Agda, and translating the entire development, sticking as closely as possible to the development in SF, took me about two days. I was pleased to discover that the proofs remained about the same size.

There was also a pleasing surprise regarding the structure of the proofs. While most proofs in both SF and PLFA are carried out by induction over the evidence that a term is well typed, in SF the central proof, that substitution preserves types, is carried out by induction on terms, not evidence of typing, for a technical reason (the context is extended by a variable binding, and hence not sufficiently "generic" to work well with Coq's



If a term is well-typed in the empty context then it satisfies progress.

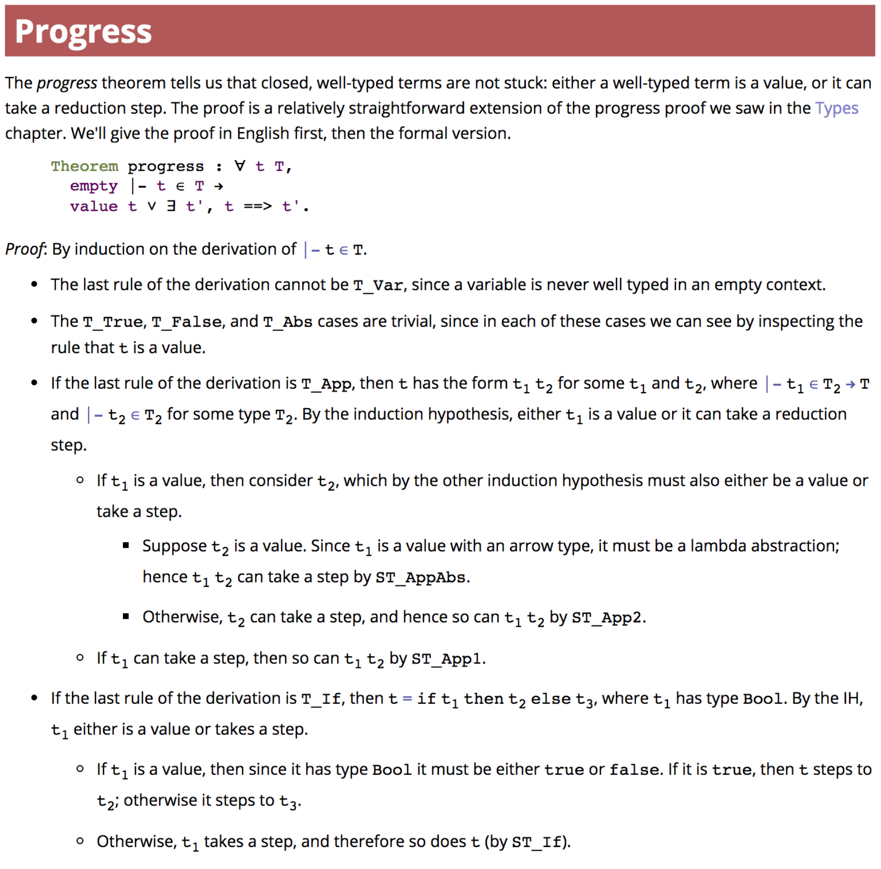
```
progress : ∀ {M A}
  → ∅ ⊢ M ⦂ A
    ----------
  → Progress M
progress (⊢` ())
progress (⊢ƛ ⊢N)                            =  done V-ƛ
progress (⊢L · ⊢M) with progress ⊢L
... | step L—→L′                            =  step (ξ-·₁ L—→L′)
... | done VL with progress ⊢M
...   | step M—→M′                          =  step (ξ-·₂ VL M—→M′)
...   | done VM with canonical ⊢L VL
...     | C-ƛ _                             =  step (β-ƛ VM)
progress ⊢zero                              =  done V-zero
progress (⊢suc ⊢M) with progress ⊢M
...  | step M—→M′                           =  step (ξ-suc M—→M′)
...  | done VM                              =  done (V-suc VM)
progress (⊢case ⊢L ⊢M ⊢N) with progress ⊢L
... | step L—→L′                            =  step (ξ-case L—→L′)
... | done VL with canonical ⊢L VL
...   | C-zero                              =  step β-zero
...   | C-suc CL                            =  step (β-suc (value CL))
progress (⊢μ ⊢M)                            =  step β-μ
```

We induct on the evidence that `M` is well-typed. Let's unpack the first three cases.

- The term cannot be a variable, since no variable is well typed in the empty context.
- If the term is a lambda abstraction then it is a value.
- If the term is an application `L · M`, recursively apply progress to the derivation that `L` is well-typed.
    - If the term steps, we have evidence that `L —→ L′`, which by `ξ-·₁` means that our original term steps to `L′ · M`
    - If the term is done, we have evidence that `L` is a value. Recursively apply progress to the derivation that `M` is well-typed.
        - If the term steps, we have evidence that `M —→ M′`, which by `ξ-·₂` means that our original term steps to `L · M′`. Step `ξ-·₂` applies only if we have evidence that `L` is a value, but progress on that subterm has already supplied the required evidence.
        - If the term is done, we have evidence that `M` is a value. We apply the canonical forms lemma to the evidence that `L` is well typed and a value, which since we are in an application leads to the conclusion that `L` must be a lambda abstraction. We also have evidence that `M` is a value, so our original term steps by `β-ƛ`.

The remaining cases are similar. If by induction we have a `step` case we apply a `ξ` rule, and if we have a `done` case then either we have a value or apply a `β` rule. For fixpoint, no induction is required as the `β` rule applies immediately.

Fig. 2. PLFA, Progress (2/2)



Progress

The *progress* theorem tells us that closed, well-typed terms are not stuck: either a well-typed term is a value, or it can take a reduction step. The proof is a relatively straightforward extension of the progress proof we saw in the Types chapter. We'll give the proof in English first, then the formal version.

```
Theorem progress : ∀ t T,
  empty |- t ∈ T →
  value t ∨ ∃ t', t ==> t'.
```

Proof: By induction on the derivation of `|- t ∈ T`.

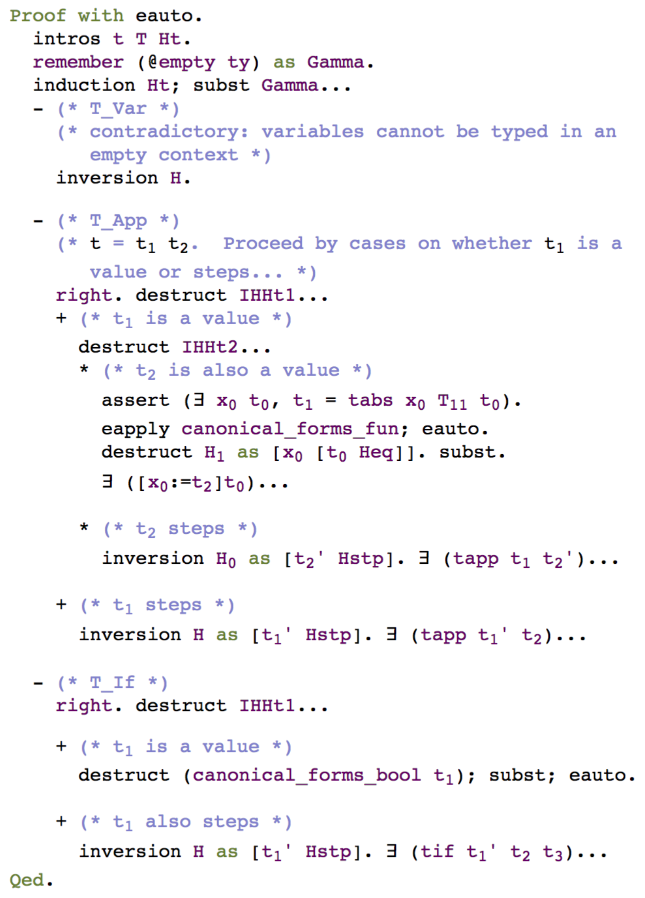
- The last rule of the derivation cannot be `T_Var`, since a variable is never well typed in an empty context.
- The `T_True`, `T_False`, and `T_Abs` cases are trivial, since in each of these cases we can see by inspecting the rule that `t` is a value.
- If the last rule of the derivation is `T_App`, then `t` has the form $t_1\ t_2$ for some t_1 and t_2, where $|- t_1 \in T_2 \to T$ and $|- t_2 \in T_2$ for some type T_2. By the induction hypothesis, either t_1 is a value or it can take a reduction step.
  - If t_1 is a value, then consider t_2, which by the other induction hypothesis must also either be a value or take a step.
    - Suppose t_2 is a value. Since t_1 is a value with an arrow type, it must be a lambda abstraction; hence $t_1\ t_2$ can take a step by `ST_AppAbs`.
    - Otherwise, t_2 can take a step, and hence so can $t_1\ t_2$ by `ST_App2`.
  - If t_1 can take a step, then so can $t_1\ t_2$ by `ST_App1`.
- If the last rule of the derivation is `T_If`, then `t` = `if` t_1 `then` t_2 `else` t_3, where t_1 has type `Bool`. By the IH, t_1 either is a value or takes a step.
  - If t_1 is a value, then since it has type `Bool` it must be either `true` or `false`. If it is `true`, then `t` steps to t_2; otherwise it steps to t_3.
  - Otherwise, t_1 takes a step, and therefore so does `t` (by `ST_If`).

Fig. 3. SF, Progress (1/2)

induction tactic). In Agda, I had no trouble formulating the same proof over evidence that the term is well typed, and didn't even notice SF's description of the issue until I was done.

The rest of the book was relatively easy to complete. The closest to an issue with proof size arose when proving that reduction is deterministic. There are 18 cases, one case per line. Ten of the cases deal with the situation where there are potentially two different reductions; each case is trivially shown to be impossible. Five of the ten cases are redundant, as they just involve switching the order of the arguments. I had to copy the cases suitably permuted. It would be preferable to reinvoke the proof on switched arguments, but this would not pass Agda's termination checker since swapping the arguments doesn't yield a recursive call on structurally smaller arguments. (I suspect tactics could cut down the proof significantly. I tried to compare with SF's proof that reduction is deterministic, but failed to find that proof.)

SF covers an imperative language with Hoare logic, culminating in code that takes an imperative programme suitably decorated with preconditions and postconditions and



```
Proof with eauto.
  intros t T Ht.
  remember (@empty ty) as Gamma.
  induction Ht; subst Gamma...
  - (* T_Var *)
    (* contradictory: variables cannot be typed in an
       empty context *)
    inversion H.

  - (* T_App *)
    (* t = t1 t2.  Proceed by cases on whether t1 is a
       value or steps... *)
    right. destruct IHHt1...
    + (* t1 is a value *)
      destruct IHHt2...
      * (* t2 is also a value *)
        assert (∃ x0 t0, t1 = tabs x0 T11 t0).
        eapply canonical_forms_fun; eauto.
        destruct H1 as [x0 [t0 Heq]]. subst.
        ∃ ([x0:=t2]t0)...

      * (* t2 steps *)
        inversion H0 as [t2' Hstp]. ∃ (tapp t1 t2')...

    + (* t1 steps *)
      inversion H as [t1' Hstp]. ∃ (tapp t1' t2)...

  - (* T_If *)
    right. destruct IHHt1...

    + (* t1 is a value *)
      destruct (canonical_forms_bool t1); subst; eauto.

    + (* t1 also steps *)
      inversion H as [t1' Hstp]. ∃ (tif t1' t2 t3)...
Qed.
```

Fig. 4. SF, Progress (2/2)

generates the necessary verification conditions. The conditions are then verified by a custom tactic, where any questions of arithmetic are resolved by the “omega” tactic invoking a decision procedure. The entire exercise would be easy to repeat in Agda, save for the last step: I suspect Agda’s automation would not be up to verifying the generated conditions, requiring tedious proofs by hand. However, I had already decided to omit Hoare logic in order to focus on lambda calculus.

To give a flavour of how the texts compare, I show the proof of progress for simply-typed lambda calculus from both texts. Figures 1 and 2 are taken from PLFA, Chapter Properties, while Figures 3 and 4 are taken from SF, Chapter StlcProp. Both texts are intended to be read online, and the figures were taken by grabbing bitmaps of the text as displayed in a browser.

PLFA puts the formal statements first, followed by informal explanation. PLFA introduces an auxiliary relation `Progress M` to capture progress; an exercise (not shown) asks the reader to show it isomorphic to the usual formulation with a disjunction and an existential. Layout is used to present the auxiliary relation in inference rule form. In Agda, any line beginning with two dashes is treated as a comment, making it easy to use a line of dashes to separate hypotheses from conclusion in inference rules. The proof is layed out carefully, with a neat indented structure to emphasise the case analysis, and all right-hand sides lined-up in the same column. My hope as an author is that students will read the formal proof first, and use it as a tabular guide to the informal explanation that follows.

SF puts the informal explanation first, followed by the formal proof. The text hides the formal proof script under an icon; the figures shows what appears when the icon is expanded. As a teacher I was aware that students might skip it on a first reading, and I would have to hope the students would return to it and step through it with an interactive tool in order to make it intelligible. I expect the students skipped over many such proofs. This particular proof forms the basis for a question of the mock exam and the past exams, so I expect most students will actually look at this one if not all the others.

Both Coq and Agda support interactive proof. Interaction in Coq is supported by Proof General, based on Emacs, or by CoqIDE, which provides an interactive development environment of a sort familiar to most students. Interaction in Agda is supported by an Emacs mode.

In Coq, interaction consists of stepping through a proof script, at each point examining the current goal and the variables currently in scope, and executing a new command in the script. Tactics are a whole sublanguage, which must be learned in addition to the langauge for expressing specifications. There are many tactics one can invoke in the script at each point; one menu in CoqIDE lists about one hundred tactics one might invoked, some in alphabetic submenus. A Coq script presents the specification proved and the tactics executed. Interaction is recorded in a script, which the students may step through at their leisure. SF contains some prose descriptions of stepping through scripts, but mainly contains scripts that students are encouraged to step through on their own.

In Agda, interaction consists of writing code with holes, at each point examining the current goal and the variables in scope, and typing code or executing an Emacs command. The number of commands available is much smaller than with Coq, the most important ones being to show the type of the hole and the types of the variables in scope; to check the code; to do a case analysis on a given variable; or to guess how to fill in the hole with constructors or variables in scope. An Agda proof consists of typed code. The interaction is *not* recorded. Students may recreate it by commenting out bits of code and introducing a hole in their place. PLFA contains some prose descriptions of interactively building code, but mainly contains code that students can read. They may also introduce holes to interact with the code, but I expect this will be rarer than with SF.

SF encourages students to interact with all the scripts in the text. Trying to understand a Coq proof script without running it interactively is a bit like understanding a chess game by reading through the moves without benefit of a board, keeping it all in your head. In contrast, PLFA provides code that students can read. Understanding the code often requires working out the types, but (unlike executing a Coq proof script) this is often easy to do in your head; when it is not easy, students still have the option of interaction.

While students are keen to interact to create code, I have found they are reluctant to interact to understand code created by others. For this reason, I suspect this may make Agda a more suitable vehicle for teaching. Nate Foster suggests this hypothesis is ripe to be tested empirically, perhaps using techniques similar to those of Danas et al. (2017).

Neat layout of definitions such as that in Figure 2 in Emacs requires a monospaced font supporting all the necessary characters. Securing one has proved tricky. As of this writing, we use FreeMono, but it lacks a few characters (⨾ and ∎) which are loaded from fonts with a different width. Long arrows are necessarily more than a single character wide. Instead, we compose reduction —→ from an em dash — and an arrow →. Similarly for reflexive and transitive closure —↠.

4 Progress + Preservation = Evaluation

A standard approach to type soundness used by many texts, including SF and PLFA, is to prove progress and preservation, as first suggested by Wright and Felleisen (1994).

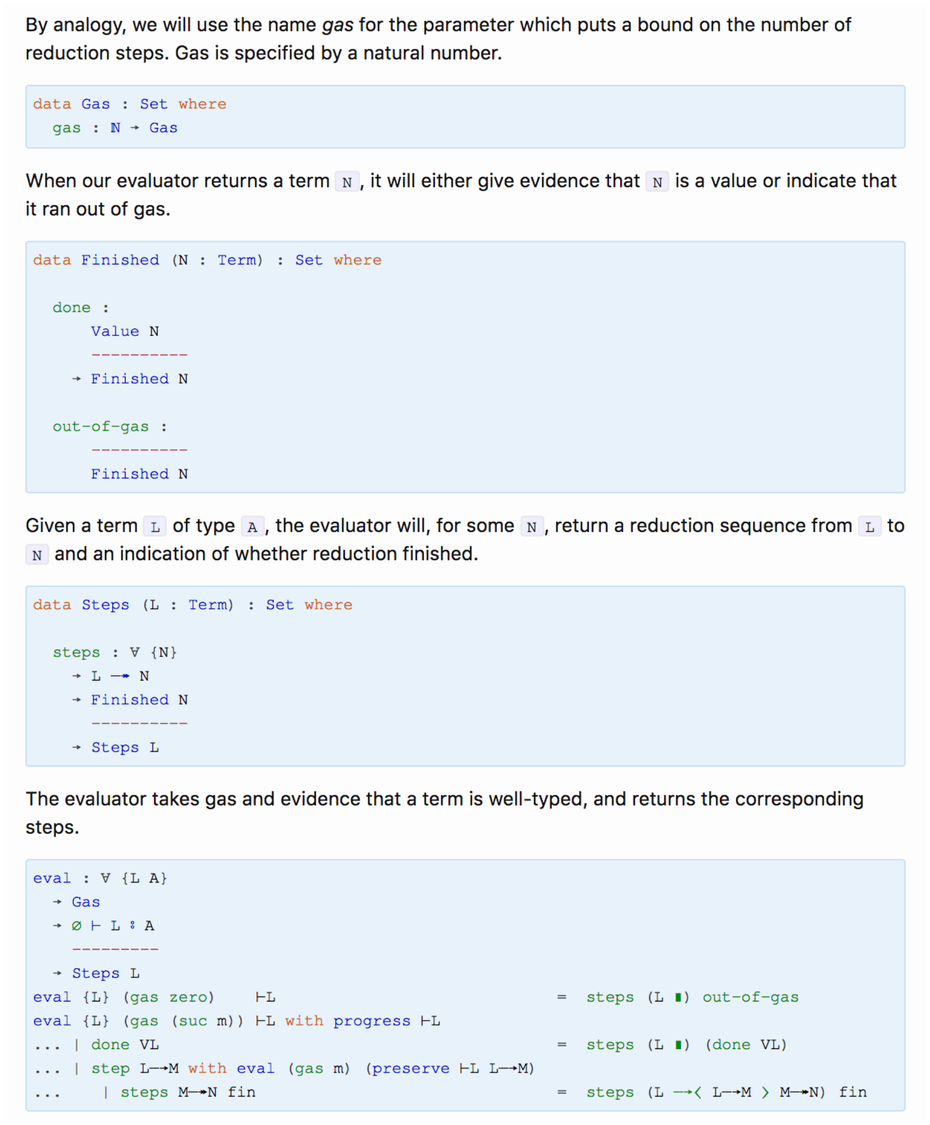
Theorem 1 (Progress). *Given term M and type A such that $\emptyset \vdash M : A$ then either M is a value or $M \longrightarrow N$ for some term N.*

Theorem 2 (Preservation). *Given terms M and N and type A such that $\emptyset \vdash M : A$ and $M \longrightarrow N$, then $\emptyset \vdash N : A$.*

A consequence is that when a term reduces to a value it retains the same type. Further, well-typed terms don't get stuck: that is, unable to reduce further but not yet reduced to a value. The formulation neatly accommodates the case of non-terminating reductions that never reach a value.

One useful by-product of the formal specification of a programming language may be a prototype implementation of that language. For instance, given a language specified by a reduction relation, such as lambda calculus, the prototype might accept a term and apply reductions to reduce it to a value. Typically, one might go to some extra work to create such a prototype. For instance, SF introduces a `normalize` tactic for this purpose. Some formal methods frameworks, such as Redex (Felleisen et al. 2009) and K (Roşu and Şerbănuţă 2010), advertise as one of their advantages that they can generate a prototype from descriptions of the reduction rules.

I was therefore surprised to realise that any constructive proof of progress and preservation *automatically* gives rise to such a prototype. The input is a term together with evidence the term is well-typed. (In the inherently-typed case, these are the same thing.) Progress determines whether we are done, or should take another step; preservation provides evidence that the new term is well-typed, so we may iterate. In a language with guaranteed termination, we cannot iterate forever, but there are a number of well-known techniques to address that issue; see, e.g., Bove and Capretta (2001), Capretta (2005), or McBride (2015). We use the simplest, similar to McBride's *petrol-driven* (or *step-indexed*) semantics: provide a maximum number of steps to execute; if that number



By analogy, we will use the name *gas* for the parameter which puts a bound on the number of reduction steps. Gas is specified by a natural number.

```
data Gas : Set where
  gas : ℕ → Gas
```

When our evaluator returns a term `N`, it will either give evidence that `N` is a value or indicate that it ran out of gas.

```
data Finished (N : Term) : Set where

  done :
      Value N
      ----------
    → Finished N

  out-of-gas :
      ----------
      Finished N
```

Given a term `L` of type `A`, the evaluator will, for some `N`, return a reduction sequence from `L` to `N` and an indication of whether reduction finished.

```
data Steps (L : Term) : Set where

  steps : ∀ {N}
    → L —↠ N
    → Finished N
      ----------
    → Steps L
```

The evaluator takes gas and evidence that a term is well-typed, and returns the corresponding steps.

```
eval : ∀ {L A}
  → Gas
  → ∅ ⊢ L ⦂ A
    ---------
  → Steps L
eval {L} (gas zero)    ⊢L                                   =  steps (L ∎) out-of-gas
eval {L} (gas (suc m)) ⊢L with progress ⊢L
... | done VL                                               =  steps (L ∎) (done VL)
... | step L—→M with eval (gas m) (preserve ⊢L L—→M)
...    | steps M—↠N fin                                     =  steps (L —→⟨ L—→M ⟩ M—↠N) fin
```

Fig. 5. PLFA, Evaluation

proves insufficient, the evaluator returns the term it reached, and one can resume execution by providing a new number.

Such an evaluator from PLFA is shown in Figure 5, where (inspired by McBride and cryptocurrencies) the number of steps to execute is referred to as *gas*. All of the example reduction sequences in PLFA were computed by the evaluator and then edited to improve readability; in addition, the text includes examples of running the evaluator with its unedited output.

It is immediately obvious that progress and preservation make it trivial to construct a prototype evaluator, and yet I cannot find such an observation in the literature nor mentioned in an introductory text. It does not appear in SF, nor in Harper (2016). A plea to the Agda mailing list failed to turn up any prior mentions. The closest related observation I have seen in the published literature is that evaluators can be extracted from proofs of normalisation (Berger 1993; Dagand and Scherer 2015).

(Late addition: My plea to the Agda list eventually bore fruit. Oleg Kiselyov directed me to unpublished remarks on his web page where he uses the name `eval` for a proof of progress and notes “the very proof of type soundness can be used to evaluate sample expressions” Kiselyov (2009).)

5 Inherent typing is golden

The second part of PLFA first discusses two different approaches to modelling simply-typed lambda calculus. It first presents “raw” terms with named variables and a separate typing relation and then shifts to inherently-typed terms with de Bruijn indices. Before writing the text, I had thought the two approaches complementary, with no clear winner. Now I am convinced that the inherently-typed approach is superior.

The clearest indication comes from counting lines of code. Stripping out examples and any proofs that appear in one but not the other, the development for raw terms takes 451 lines (216 lines of definitions and 235 lines for the proofs) and the development for inherently typed terms takes 275 lines (with definitions and proofs interleaved, as substitution cannot be defined without also showing it preserves types). We have 451 / 235 = 1.64, close to the golden ratio.

Another indication is expressive power. The approach with named variables and separate typing is restricted to substitution of one variable by a single closed term, while de Bruijn indices with inherent typing support simultaneous substitution of all variables by open terms, using a pleasing formulation due to McBride (2005), inspired by Goguen and McKinna (1997) and described in Allais et al. (2017). In fact, I managed to extend McBride’s approach to support raw terms with named variables, but the resulting code was too long and too complex for use in an introductory text, requiring 695 lines of code—more than the total for the other two approaches combined.

The text develops both approaches because named variables with separate typing is more familiar, and placing de Bruijn indices and inherent typing first would lead to a steep learning curve. By presenting the more long-winded but less powerful approach first, students can see for themselves the advantages of de Bruijn indices with inherent typing.

There are actually four possible designs, as the choice of named variables vs de Bruijn indices, and the choice of raw vs inherently-typed terms may be made independently. There are synergies beween the two. Manipulation of de Bruijn indices can be notoriously error-prone without inherent-typing to give assurance of correctness. In inherent typing with named variables, simultaneous substitution by open terms remains difficult.

The benefits of replacing raw typing by inherent are well known to some. The technique was introduced by Altenkirch and Reus (1999), and widely used elsewhere, notably by Chapman (2009) and Allais et al. (2017). I'm grateful to David Darais for bringing it to my attention.

6 Conclusion

I look forward to experience teaching from the new text, and encourage others to use it too. Please comment!

Acknowledgements A special thank you to my coauthor, Wen Kokke. For inventing ideas on which PLFA is based, and for hand-holding, many thanks to Conor McBride, James McKinna, Ulf Norell, and Andreas Abel. For showing me how much more compact it is to avoid raw terms, thanks to David Darais. For inspiring my work by writing SF, thanks to Benjamin Pierce and his coauthors. For comments on a draft of this paper, an extra thank you to James McKinna, Ulf Norell, Andreas Abel, and Benjamin Pierce.

Bibliography

Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 195–207. ACM, 2017.

Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *International Workshop on Computer Science Logic*, pages 453–468. Springer, 1999.

Ulrich Berger. Program extraction from normalization proofs. In *International Conference on Typed Lambda Calculi and Applications*, pages 91–106. Springer, 1993.

Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In *International Conference on Theorem Proving in Higher Order Logics*, pages 121–125. Springer, 2001.

Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.

James Maitland Chapman. *Type checking and normalisation*. PhD thesis, University of Nottingham, 2009.

Pierre-Évariste Dagand and Gabriel Scherer. Normalization by realizability also evaluates. In *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, 2015.

Natasha Danas, Tim Nelson, Lane Harrison, Shriram Krishnamurthi, and Daniel J Dougherty. User studies of principled model finder output. In *International Conference on Software Engineering and Formal Methods*, pages 168–184. Springer, 2017.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. By Press, 2009.

Healfdene Goguen and James McKinna. Candidates for substitution. Technical report, Laboratory for Foundations of Computer Science, University of Edinburgh, 1997.

Georges Gonthier. The four colour theorem: Engineering of a formal proof. In *Computer mathematics*, pages 333–333. Springer, 2008.

Georges Gonthier, Andrea Asperti, Jeremy Avigad, et al. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179. Springer, 2013.

Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. A formal proof of the Kepler conjecture. In *Forum of Mathematics, Pi*, volume 5. Cambridge University Press, 2017.

Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.

Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant a tutorial. *Rapport Technique*, 178, 1997.

Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. Closing the gap–the formally verified optimizing compiler compcert. In *SSS'17: Safety-critical Systems Symposium 2017*, pages 163–180. CreateSpace, 2017.

Oleg Kiselyov. Formalizing languages, mechanizing type-soundess and other meta-theoretic proofs, 2009. URL http://okmij.org/ftp/formalizations/index.html.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

Conor McBride. Type-preserving renaming and substitution, 2005. unpublished manuscript.

Conor McBride. Turing-completeness totally free. In *International Conference on Mathematics of Program Construction*, pages 257–275. Springer, 2015.

Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. In *ICFP*, pages 89–102, 2016.

Benjamin C Pierce. Lambda, The Ultimate TA. In *ICFP*, pages 121–22, 2009.

Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software foundations*. 2010. URL \url{http://www.cis.upenn.edu/bcpierce/sf/current/index.html}.

Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

Aaron Stump. *Verified functional programming in Agda*. Morgan & Claypool, 2016.

Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.