Programming Language Foundations in Agda

Wen Kokke[a], Jeremy G. Siek[b], Philip Wadler[a,*]

[a]*University of Edinburgh, 10 Crichton Street, EH8 9AB, Edinburgh*
[b]*Indiana University, 700 N Woodlawn Ave, Bloomington, IN 47408, USA*

Abstract

One of the leading textbooks 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, we came 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, we 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 we 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 we find it in the literature. Third, that using extrinsically-typed terms is far less perspicuous than using intrinsically-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

*Corresponding author
Email addresses: `wen.kokke@ed.ac.uk` (Wen Kokke), `jsiek@indiana.edu` (Jeremy G. Siek), `wadler@inf.ed.ac.uk` (Philip Wadler)

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"—specifications in the proof assistant Agda both describe programming languages and are themselves programmes.

Since 2013, one of us (Philip) has 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 (2002), but this 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). We are 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, Philip came 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.

We found ourselves 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 focuses 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 that after five years in the classroom Philip had his own ideas about how to present the material. They say you should never write a book unless you cannot *not* write the book, and Philip soon found that this was a book he could not not write.

Philip considered himself fortunate that his student, Wen, was keen to help. She guided Philip as a newbie to Agda and provided an infrastructure for the book that we found easy to use and produces pages that are a pleasure to view. The bulk of the first draft of the book was written January–June 2018, while Philip was on sabbatical in Rio de Janeiro. After the first draft was published, Jeremy wrote eight additional chapters, covering aspects of operational and denotational semantics.

This paper is the journal version of Wadler (2018). It adds two new authors, summaries of Jeremy's new chapters, and sections on experience with teaching and software used to publish the book. The original text often used first person, which here is replaced by reference to Philip.

This paper is a personal reflection, summarising what was 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 might 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 our 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 we cannot find it described in the literature.

Section 5 claims that extrinsically-typed terms should be avoided in favour of intrisicly-typed terms. PLFA develops lambda calculus with both, 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.

Section 6 describes experience teaching from the textbook. The point of proof is perfection, and it turns out that an online final examination with access to a proof assistant can lead to flawless student performance.

Section 7 outlines our experience publishing the book as open source in GitHub. We were surprised at how effective this was at eliciting community participation. A large number of people have submitted pull requests to improve the book.

We argue that Agda has advantages over Coq for pedagogic purposes. Our focus is purely on the case of a proof assistant as an aid to *learning* formal semantics using examples of *modest* size. We 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 semantics of programming languages, with simply-typed and untyped lambda calculi as the central examples. 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 misapprehensions.

The book is broken into three parts. The first part, Logical Foundations, develops the needed formalisms. The second part, Programming Language Foundations, introduces basic methods of operational semantics. The third part, Denotational Semantics, introduces a simple model of the lambda calculus and its properties. (SF is divided into books, the first two of which have the same names as the first two parts of PLFA, and cover similar material.) Part I and Part II up to Untyped were written by Philip, Part II from Substitution and Part III were written by Jeremy.

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.

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

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.

Part II, Programming Language Foundations

Lambda: Introduction to lambda calculus. Introduces lambda calculus, using a representation with named variables and extrinsically typed. The language used is PCF (Plotkin, 1977), 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: Intrinsically-typed de Bruijn representation. Introduces de Bruijn indices and the intrinsically-typed representation. Emphasis is put on the structural similarity between a term and its corresponding type derivation; in particular, de Bruijn indices 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; and explains lists and let bindings. Typing and reduction rules are given informally; a few are then give formally, and the rest are left as exercises for the reader. The intrinsically-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 extrinsically typed to a representation with de Bruijn indices and intrinsically typed. Bidirectional type inference is shown to be both sound and complete.

Untyped: Untyped calculus with full normalisation. As a variation on earlier themes, discusses an untyped (but intrinsically 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.

Substitution: in the untyped lambda calculus. Delves deeper into the properties of simultaneous substitution, establishing the equations of the σ algebra of Abadi et al. (1991). These equations enable a straightforward proof of the Substitution Lemma (Barendregt, 1984), which is needed in the next chapter.

Confluence: of the untyped lambda calculus. Presents a proof of the Church-Rosser theorem based on the classic idea of parallel reduction due to Tait and Martin-Löf. The proof in Agda is streamlined by the use of ideas from Schäfer et al. (2015) and Pfenning (1992).

Big-step: evaluation for call-by-name. Introduces the notion of big-step evaluation, written $\gamma \vdash M \Downarrow V$, to develop a deterministic call-by-name reduction strategy. The main result of this chapter is a proof that big-step evaluation implies the existence of a reduction sequence that terminates with a lambda abstraction.

Part III, Denotational Semantics

Denotational: semantics of the untyped lambda calculus. The early denotational semantics of the lambda calculus based on graph models (Scott, 1976; Engeler, 1981; Plotkin, 1993) and filter models (Barendregt et al., 1983) were particularly simple and elegant: a function is modeled as a lookup table. This chapter presents such a semantics using a big-step notation that is approachable to readers familiar with operational semantics, writing $\gamma \vdash M \downarrow d$ for the evaluation of a term M to denotation d in environment γ.

Compositional: the denotational semantics is compositional. The hallmark of denotational semantics is that they are compositional: the meaning of each language form is a function of the meanings of its parts. We define two functions, named `curry` and `apply` that serve this purpose for lambda abstraction and application. The results in this chapter include congruences for `curry` and `apply` and the compositionality property for the denotational semantics. The chapter concludes with a functional definition of the denotational semantics and a proof that it is equivalent to the big-step version.

Soundness: of reduction with respect to denotational semantics. Reduction implies denotational equality. We prove each direction of the equality, first showing the reduction preserves denotations (subject reduction), and then showing that reduction reflects denotations (subject expansion). The first proof is similar to the type preservation proofs in Part II. The second goes in reverse, showing that if $M \longrightarrow N$ and $\gamma \vdash N \downarrow d$, then $\gamma \vdash M \downarrow d$.

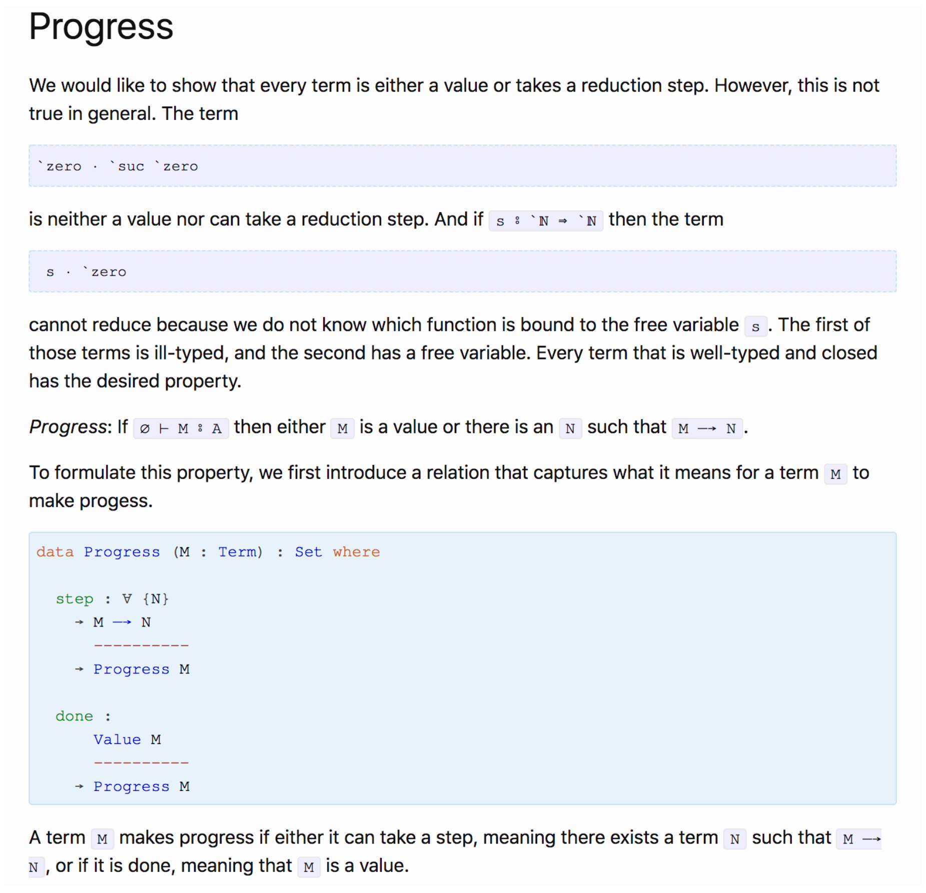
Adequacy: of denotational semantics with respect to reduction. If a term is denotationally equal to a lambda expression, then it reduces to a lambda expression. The main lemma shows that if a term's denotation is functional, i.e., $\gamma \vdash M \downarrow (d \mapsto d')$, then M terminates according to the call-by-name big-step semantics, i.e., $\gamma' \vdash M \Downarrow V$. A logical relation $\mathbb{V}$ is used to relate denotations and values (i.e. closures). The implication from the big-step semantics to reduction is proved in the Big-step chapter of Part II.

Contextual Equivalence: is implied by denotational equality. The main criteria for behavior-preserving code transformation (such as compiler optimization or programmer refactoring) is contextual equivalence. Two terms are contextually equivalent when they can both be placed into an arbitrary context (a program with a hole) and the resulting programs behave the same (e.g., they both terminate or they both diverge). This chapter ties together the previous results (Compositionality, Soundness, and Adequacy) to show that denotational equality implies contextual equivalence. Thus, it is safe to use denotational equality to justify code transformations.

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. We chose PCF because it lets us use the same examples, based on addition and multiplication, for the early chapters of both Part I and Part II. 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, and for lambda calculus covers subtyping, record types, mutable references, and normalisation—none of which are treated by PLFA. PLFA covers an intrinsically-typed de Bruijn representation, bidirectional type inference, bisimulation, and an untyped call-by-name language with full normalisation—none of which are treated by SF. The new part on Denotational Semantics also covers material not treated by SF.

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


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.

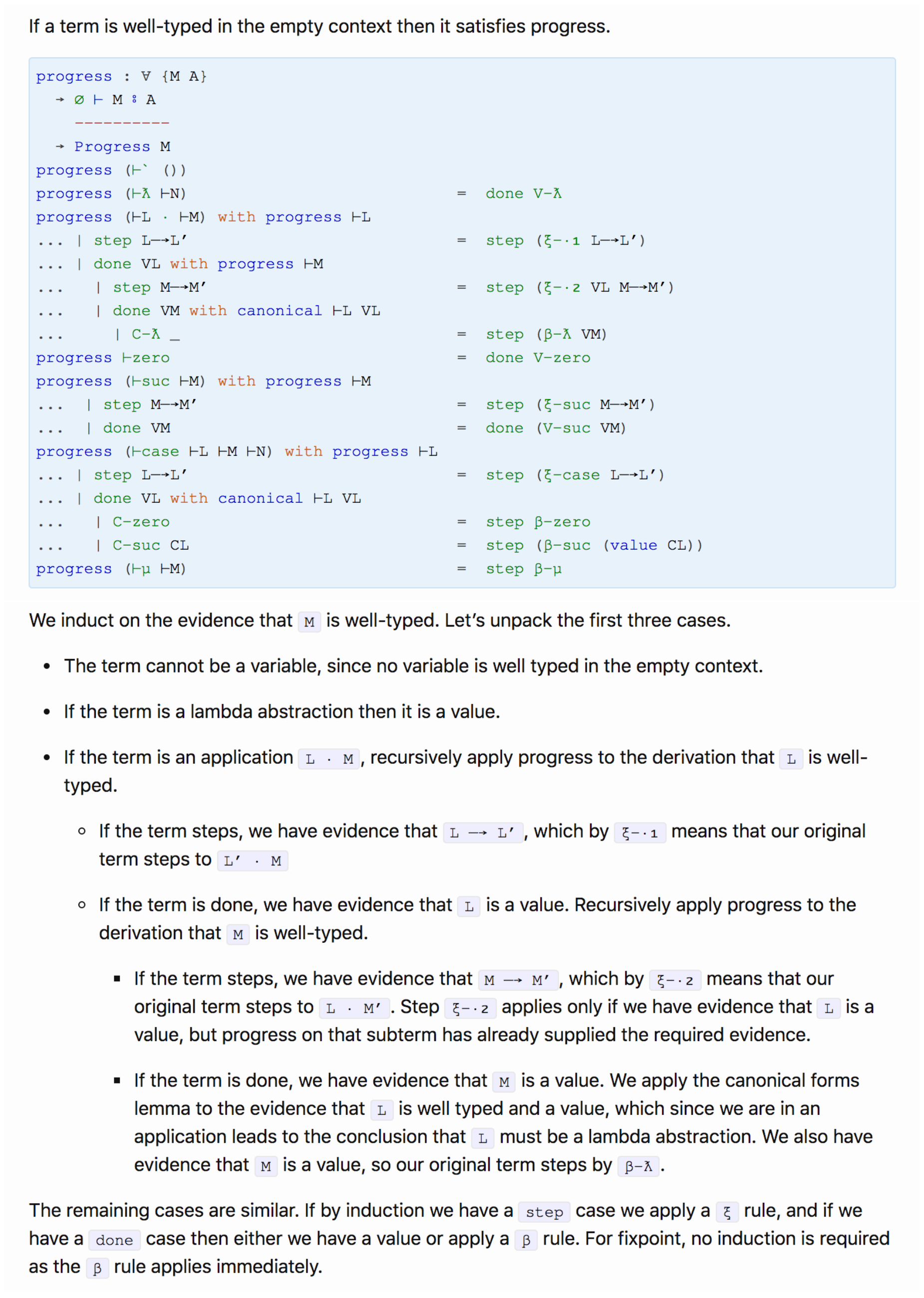
Figure 1: PLFA, Progress (1/2)

recently added a fourth volume on random testing of Coq specifications using QuickChick. There is currently no tool equivalent to QuickChick for Agda.

There is more material that would be desirable to include in PLFA which was not due to limits of time, including mutable references, System F, logical relations for parametricity, and pure type systems. We would 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. Our attempts so far to formalise pure type systems have proved challenging, to say the least.

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?


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

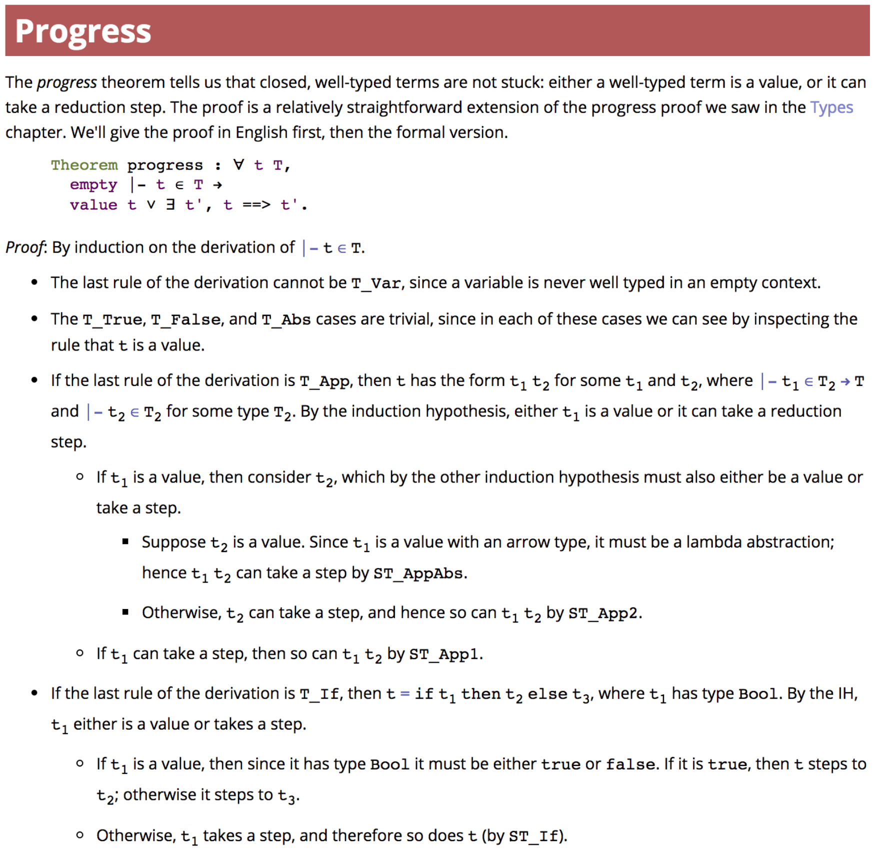
```
progress : ∀ {M A}
  → ∅ ⊢ M ⦂ A
    ----------
  → Progress M
progress (⊢` ())
progress (⊢ƛ ⊢N)                            =  done V-ƛ
progress (⊢L · ⊢M) with progress ⊢L
... | step L—→L′                            =  step (ξ-·₁ L—→L′)
... | done VL with progress ⊢M
...   | step M—→M′                          =  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.

Figure 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 $|- t_1 \in T_2 \rightarrow T$ 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`).

Figure 3: SF, Progress (1/2)

As an experiment, Philip first rewrote SF's development of simply-typed lambda calculus (SF, Chapters Stlc and StlcProp) in Agda. He was a newbie to Agda, and translating the entire development, sticking as closely as possible to the development in SF, took about two days. We were 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 for a technical reason (the context is extended by a variable binding, and hence not sufficiently "generic" to work well with Coq's induction tactic). In Agda, we 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 we were 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

```
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.
```

Figure 4: SF, Progress (2/2)

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. We had to copy the cases nsuitably 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. The proof of determinism in SF (Chapter Norm) is for a different language of comparable size, and has a comparable size.

SF covers an imperative language with Hoare logic, culminating in code that takes an imperative programme suitably decorated with preconditions and postconditions and 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, as Agda does not offer support for proof automation out of the box. It is certainly possible to implement proof automation in Agda—see, e.g., the auto tactic by Kokke and Swierstra (2015), and the collection of tactics in Ulf Norell's `agda-prelude`[1]. The standard library comes equipped with solvers for equations on monoids and rings[2], and a much improved solver for equalities on rings was recently contributed by Kidney (2019). We suspect that, while Agda's automation would be up to verifying the generated conditions, some effort would be require to implement the required custom tactic, and a section would need to be added to the book to cover proof automation. For the time being, we have decided to omit Hoare logic in order to focus on lambda calculus.

To give a flavour of how the texts compare, we 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 show screengrabs of the text as displayed in a browser.

PLFA puts the formal statements first, followed by informal explanation. PLFA introduces an auxiliary relation `Progress` 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 of proposition `progress` (the different case making it a distinct name) is layed out carefully. The neat indented structure emphasises the case analysis, and all right-hand sides line-up in the same column. Our hope as authors is that students 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 figure shows what appears when the icon is expanded. As teachers, we were aware that students might skip the formal proof on a first reading, and we have to hope the students return to it and step through it with an interactive tool in order to make it intelligible. We expect the students skipped over many such proofs. This particular proof forms the basis for a question on several exams, so we expect most students will look at this one if not all the others.

(For those wanting more detail: In PLFA, variables and abstractions and applications in the object language are written `` ` x`` and `ƛ x ⇒ N` and `L · M`. The corresponding typing rules are referred to by `` ⊢` ()`` and `⊢ƛ ⊢N` and `⊢L · ⊢M`, where `⊢L`, `⊢M`, `⊢N` are the proofs that terms `L`, `M`, `N` are well typed, and '()' denotes that there cannot be evidence that a free variable is well typed in the empty context. It was decided to overload infix dot for readability, but not other symbols.

[1] https://github.com/UlfNorell/agda-prelude

[2] https://agda.github.io/agda-stdlib/Algebra.Solver.Ring.html

In Agda, as in Lisp, almost any sequence of characters is a name, with spaces essential for separation.)

(In SF, variables and abstractions and applications in the object language are written `tvar x` and `tabs x t` and `tapp` t_1 t_2. The corresponding typing rules are referred to as `T_Var` and `T_Abs` and `T_App`.)

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 in scope, and executing a new command in the script. Tactics are a whole sublanguage, which must be learned in addition to the language 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 invoke, 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 typecheck the code; to do a case analysis on a given variable; or to search for a way 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 introduce holes to interact with the code, but we expect that will be rare.

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, we have found they are reluctant to interact to understand code created by others. For this reason, we 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 of the unicode long arrow, we compose reduction —→ from an em dash — and an ordinary 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 $\emptyset \vdash N : A$.*

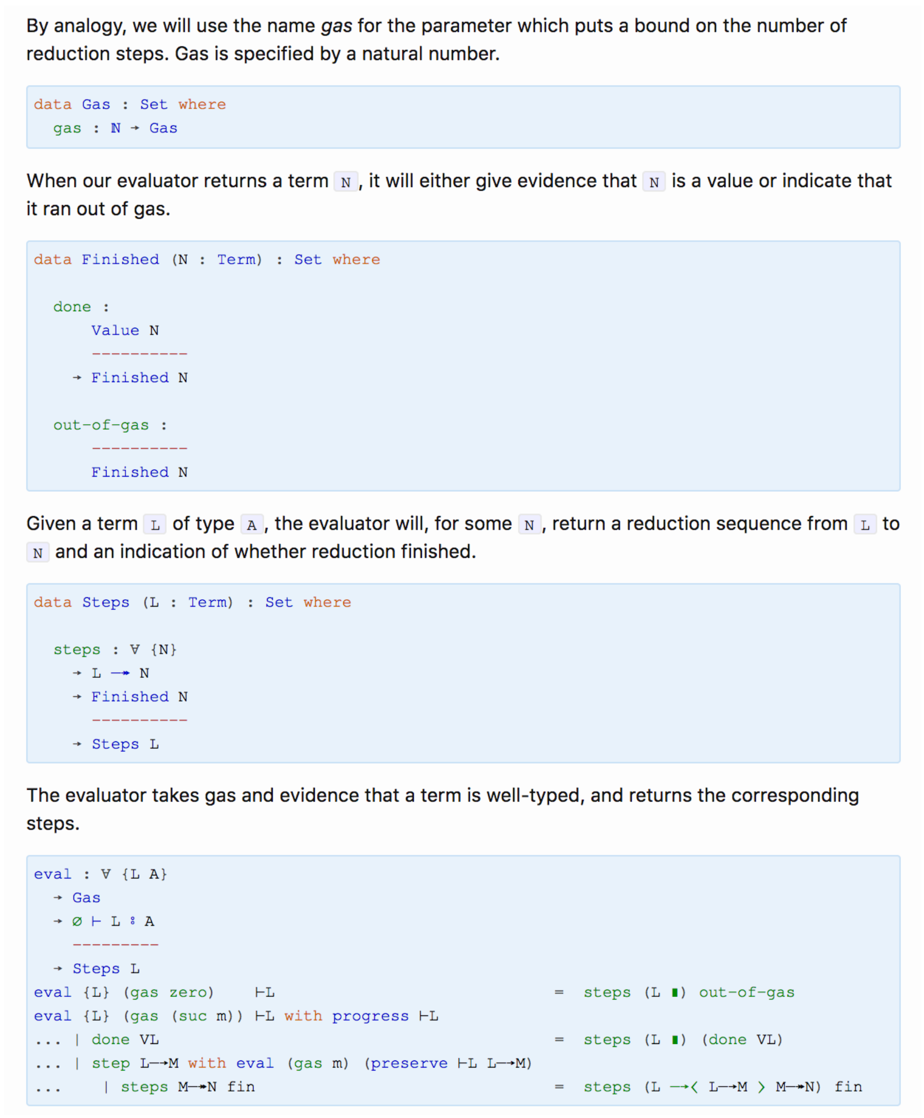
A consequence is that when a term reduces to a value it retains the same type. Further, well-typed terms don't get stuck: that is, unable to reduce further but not yet reduced to a value. The formulation neatly accommodates the case of non-terminating reductions that never reach a value.

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.

Philip had been exposed to the work of the K team, as both consulted for IOHK, a cryptocurrency firm. This put us keenly in mind of the need for animation; Philip sometime referred to this as "K-envy" or "Redex-envy".

Philip 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 intrinsically-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 such as Agda, 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 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 cryptocurrencies) the number of steps to execute is referred to as *gas*. All of the



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
```

Figure 5: PLFA, Evaluation

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 we cannot find such an observation in the literature nor mentioned in an introductory text. It does not appear in SF, which introduces a specialised `normalise` tactic instead. A plea to the Agda mailing list failed to turn up any prior mentions. The closest related observation we have seen in the published literature is that evaluators can be extracted from proofs of normalisation (Berger, 1993; Dagand and Scherer, 2015).

Some researchers are clearly familiar with the connection between progress and preservation and animation. In private correspondence, Bob Harper referred to it as the *pas de deux*, a dance between progress, which takes well-typing to a step, and preservation, which takes a step back to well-typing—but neither the technique nor the appealing terminology appears in Harper (2016). The appeal to the Agda mailing list bore late fruit: Oleg Kiselyov directed us 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). Nonetheless, as of this writing, we still have not located a mention in the published literature.

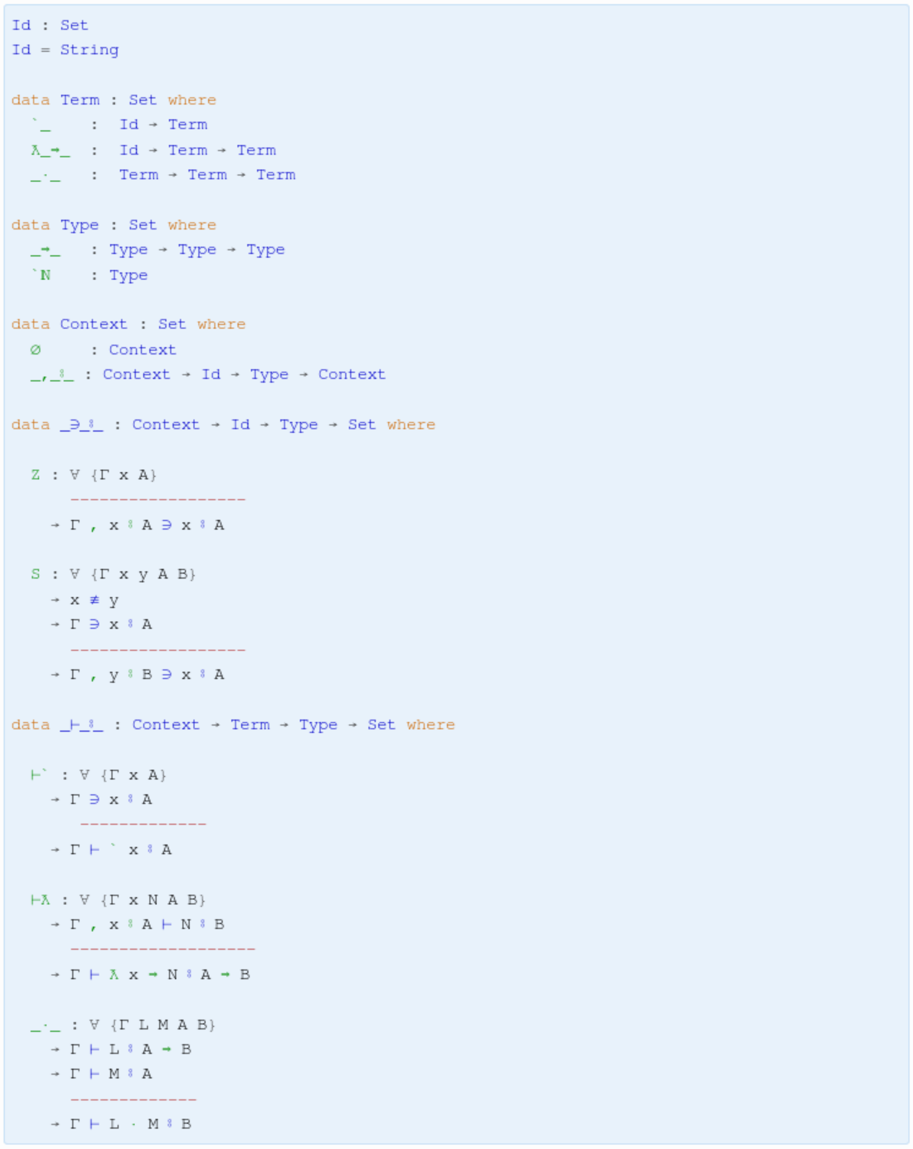
There are places in the literature where one might expect a remark on the relation between progress and preservation and animation—but no such remark appears. In the PoplMark Challenge (Aydemir et al., 2005), Challenge 2A is to prove progress and preservation for System $F_{<:}$, while Challenge 3 is to prove animation for the same system. Nowhere do the authors indicate that in an intuitionistic logic these are essentially the same problem. Owens et al. (2016), when discussing extraction of animators for small-step semantics, mention Redex and K, but no other possibilities. We hope the stress in PLFA on the fact that in an intuitionistic setting progress and preservation imply animation will mean that the connection becomes more widely known.

5. Intrinsic typing is golden

The second part of PLFA first discusses two different approaches to modeling simply-typed lambda calculus. It first presents terms with named variables and extrinsic typing relation and then shifts to terms with de Bruijn indices that are intrinsically typed. The names *extrinsic* and *intrinsic* for these two approaches are taken from Reynolds (2003). Before writing the text, Philip had thought the two approaches complementary, with no clear winner. Now he is convinced that the intrinsically-typed approach is superior.

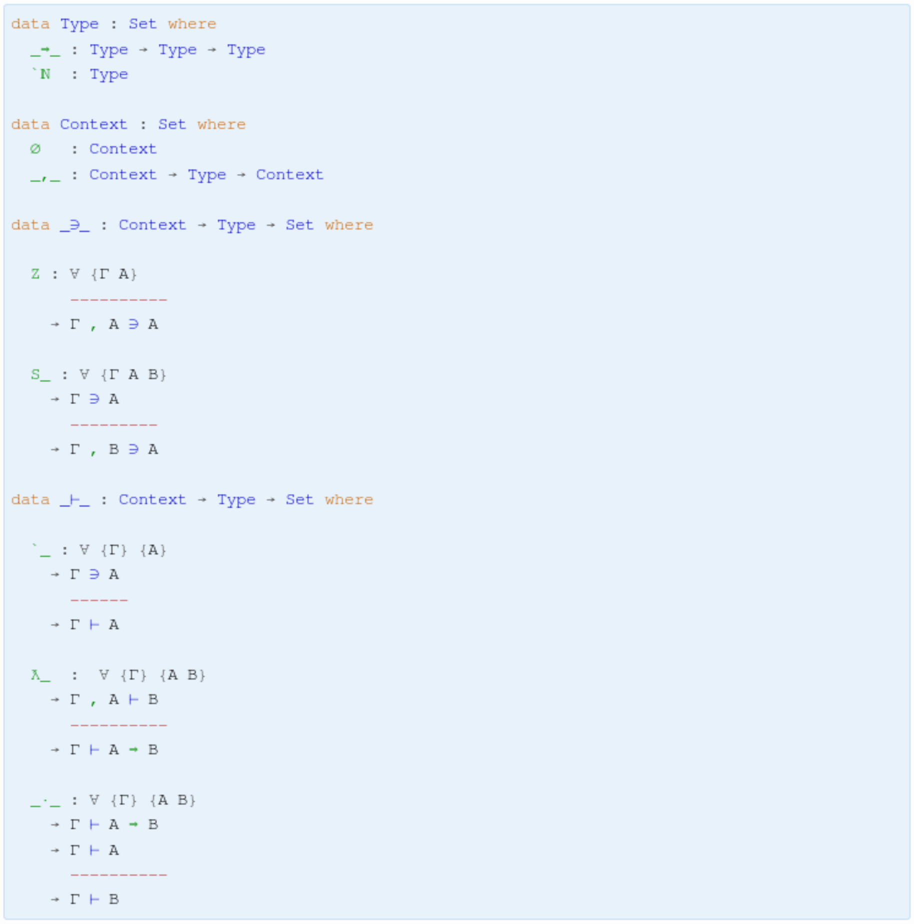
Figure 6 presents the extrinsic approach. It first defines `Id`, `Term`, `Type`, and `Context`, the abstract syntax of identifiers, raw terms, types, and contexts. It then defines two judgments, `Γ ∋ x ⦂ A` and `Γ ⊢ M ⦂ A`, which hold when under context Γ the variable `x` and the term `M` have type `A`, respectively.

Figure 7 presents the intrinsic approach. It first defines `Type` and `Context`, the abstract syntax of types and contexts, of which the first is as before and the



```
Id : Set
Id = String

data Term : Set where
  `_    :  Id → Term
  ƛ_⇒_  :  Id → Term → Term
  _·_   :  Term → Term → Term

data Type : Set where
  _⇒_   : Type → Type → Type
  `ℕ    : Type

data Context : Set where
  ∅     : Context
  _,_⦂_ : Context → Id → Type → Context

data _∋_⦂_ : Context → Id → Type → Set where

  Z : ∀ {Γ x A}
      ------------------
    → Γ , x ⦂ A ∋ x ⦂ A

  S : ∀ {Γ x y A B}
    → x ≢ y
    → Γ ∋ x ⦂ A
      ------------------
    → Γ , y ⦂ B ∋ x ⦂ A

data _⊢_⦂_ : Context → Term → Type → Set where

  ⊢` : ∀ {Γ x A}
    → Γ ∋ x ⦂ A
       -------------
    → Γ ⊢ ` x ⦂ A

  ⊢ƛ : ∀ {Γ x N A B}
    → Γ , x ⦂ A ⊢ N ⦂ B
      -------------------
    → Γ ⊢ ƛ x ⇒ N ⦂ A ⇒ B

  _·_ : ∀ {Γ L M A B}
    → Γ ⊢ L ⦂ A ⇒ B
    → Γ ⊢ M ⦂ A
      -------------
    → Γ ⊢ L · M ⦂ B
```

Figure 6: Extrinsic approach in PLFA



```
data Type : Set where
  _⇒_ : Type → Type → Type
  `ℕ  : Type

data Context : Set where
  ∅   : Context
  _,_ : Context → Type → Context

data _∋_ : Context → Type → Set where

  Z : ∀ {Γ A}
      ----------
    → Γ , A ∋ A

  S_ : ∀ {Γ A B}
    → Γ ∋ A
      ---------
    → Γ , B ∋ A

data _⊢_ : Context → Type → Set where

  `_ : ∀ {Γ} {A}
    → Γ ∋ A
      ------
    → Γ ⊢ A

  ƛ_  :  ∀ {Γ} {A B}
    → Γ , A ⊢ B
      ----------
    → Γ ⊢ A ⇒ B

  _·_ : ∀ {Γ} {A B}
    → Γ ⊢ A ⇒ B
    → Γ ⊢ A
      ----------
    → Γ ⊢ B
```

Figure 7: Intrinsic approach in PLFA

second is as before with identifiers dropped. In place of the two judgments, the types of variables and terms are indexed by a context and a type, so that Γ ∋ A and Γ ⊢ A denote variables and terms, respectively, that under context Γ have type A. The indexed types closely resemble the previous judgments: we now represent a variable or a term by the proof that it is well typed. In particular, the proof that a variable is well typed in the extrinsic approach corresponds to a de Bruijn index in the intrinsic approach.

The extrinsic approach requires more lines of code than the intrinsic approach. The separate definition of raw terms is not needed in the intrinsic approach; and one judgment in the extrinsic approach needs to check that x ≢ y, while the corresponding judgment in the intrinsic approach does not. The difference becomes more pronounced when including the code for substitution, reductions, and proofs of progress and preservation. In particular, where the extrinsic approach requires one first define substitution and reduction and then prove they preserve types, the intrinsic approach establishes substitution preserves types at the same time it defines substitution and reduction.

Stripping out examples and any proofs that appear in one but not the other (but could have appeared in both), the full development in PLFA for the extrinsic approach takes 451 lines (216 lines of definitions and 235 lines for the proofs) and the development for the intrinsic approach takes 275 lines (with definitions and proofs interleaved). We have 451 / 235 = 1.64, close to the golden ratio.

The intrinsic approach also has more expressive power. The extrinsic approach is restricted to substitution of one variable by a closed term, while the intrinsic approach supports simultaneous substitution of all variables by open terms, using a pleasing formulation due to McBride (2005), inspired by Goguen and McKinna (1997) and Altenkirch and Reus (1999) and described in Allais et al. (2017). In fact, we did manage to write a variant of the extrinsic approach with simultaneous open substitution along the lines of McBride, but the result was 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 the extrinsic approach is more familiar, and because placing the intrinsic approach 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 and intrinsic types.

There are actually four possible designs, as the choice of named variables vs de Bruijn indices, and the choice of extrinsic vs intrinsic typing may be made independently. But the two designs we chose work well, while the other two are problematic. Manipulation of de Bruijn indices can be notoriously error-prone without intrinsic types to give assurance of correctness. For instrinsic typing with named variables, simultaneous substitution by open terms remains difficult.

The benefits of the intrinsic approach 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). Philip is grateful to David Darais for bringing it to his attention.

6. Teaching experience

Philip now has five years of experience teaching from SF and one year teaching from PLFA. To date, he has taught three courses from PLFA.

- University of Edinburgh, September–December 2018 (with teaching assistance from Wen and Chad Nester); twenty 2-hour slots, comprising one hour of lecture followed by one hour of lab. Ten students completed the course, fourth-year undergraduates and masters. The course covered Parts I and II of PLFA, up through chapter Untyped.

- Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), March–July 2019, hosted by Roberto Ieuramalischy; ten 3-hour slots, comprising two hours of lecture followed by one hour of lab. Ten students completed the course, mostly doctoral students. The course covered Parts I and II of PLFA, up through chapter Untyped, save students read chapter Lists on their own, and chapter Bisimilarity was skipped.

- University of Padova, June 2018, hosted by Maria Emilia Maietti; two 3-hour slots, comprising two hours of lecture followed by one hour of lab. Thirty undergraduate students sat the course, which covered chapters Naturals, Induction, and Relations.

In addition, David Darais at University of Vermont and John Leo at Google Seattle have taught from PLFA.

Exercises in PLFA are classified in three ways.

- Exercises labelled "(recommended)" are the ones students are required to do in the classes taught at Edinburgh and PUC-Rio.

- Exercises labelled "(stretch)" are there to provide an extra challenge. Few students do all of these, but most attempt at least a few.

- Exercises without a label are included for those who want extra practice. To Philip's surprise, students at PUC-Rio completed quite a few of these.

Students are expected to complete coursework for all of the required questions in the text, optionally doing any stretch or unlabelled exercises. Coursework also includes a "mock mock" exam, as described below.

The mark for the course is based on coursework and a final-exam, weighted 1/4 for coursework and 3/4 for the final exam. The weighting for coursework is designed to be high enough to encourage students to do it (they all do), but not so high as to encourage cheating. Students are encouraged to help each other with coursework.

The final-exam is two hours online. Students have access to the Agda proof assistant to check their work. At Edinburgh, students use computers with a special exam operating system that disables access to the internet. Students are given access to the text of PLFA, the Agda standard libraries, and the Agda language reference manual, but no other materials.

Students must answer question 1 on the exam, and one of questions 2 or 3. Question 1 gives predicates over a data structure, such lists or trees, to be formalised in Agda, and a theorem relating the predicates, to be formalised and proved in Agda. Question 2 gives the students the intrinsic formulation of lambda calculus from chapter DeBruijn, which they must extend with a described language feature. Question 3 give the students the the bidirectional type inferencer from chapter Inference, which they must extend with a described language feature.

Because the course is taught using a proof assistant, it is important that students have access to a proof assistant during the exam. Students are told in advance that they are expected to get perfect on the exam, and that they will have to study hard to achieve it. Given that the goal of formal methods is to avoid error, we believe a pedagogical purpose is served by telling the students that they are expected to achieve perfection and making it possible for them to do so. Students are given two opportunities to practice in the run up to the exam, a "mock" exam given in class under exam conditions (two hours online), and before that a "mock mock" exam as coursework (in their own time, encouraged to ask questions, tasked to do all three questions rather than two of three).

For the courses run at Edinburgh and PUC-Rio, the scores vary widely on the mock: minimum <20%, maximum 100%, mean 77.8, standard deviation 27.6. But all students achieve perfection on the exam. (The one exception was a PUC-Rio student who did not attend classes or sit the mock.) Similar results were achieved at Edinburgh over the previous five years, using SF as the course textbook and Coq as the proof assistant. We consider these results a tribute to the students' ability to study and learn.

7. Software

The book is written using a combination of literate Agda and Markdown. At the time of writing, the book is published using GitHub Pages and the Jekyll static site generator. The book is open source—the source is currently also hosted on GitHub, under a Creative Commons CC-BY license. The open-source aspect is important—as the book is written in literate Agda, it is essential that anyone can download and execute the source.

We maintain a number of tools, which play various roles in rendering the book in all its "glorious clickable HTML". We render the literate Agda to highlighted HTML using Agda's HTML backend. In addition to highlighting, this inserts clickable links, linking each constructor and function to the site of its definition. However, the links Agda inserts are local and don't match the structure of the book. We maintain a script, `highlight.sh`, which fixes these links, rerouting links to the standard library to the online version, and correcting links to local modules.

(Before the release of Agda 2.6, Agda did not support highlighting embedded literate code in HTML. We maintained `agda2html`, a tool which rewrites the output of Agda's HTML highlighter to highlight embedded code. The tool had much more functionality, including the fixing of links as outlined above, the stripping of implicit arguments to achieve a Haskell-like look, and the support for new Markdown constructs for linking to Agda names. However, Agda 2.6 has incorporated almost all of this functionality, and `agda2html` is now deprecated.)

The book is built, tested, and published after each commit, using Travis CI, a web service for continuous integration. This means that the book is constantly changing. To accommodate those who want a more stable version, e.g., for teaching, we maintain a stable version of the book at http://plfa.inf.ed.ac.uk/stable

The stable version of the book is updated much less frequently, and updates are announced.

We maintain a tool called, simply, `acknowledgements`, which uses the GitHub API to automatically extract a list of contributors to the book, and add them to the Acknowledgements page, each time the book is published. We consider anyone who has sent a successful pull request a contributor, and sort contributors in the acknowledgments by the number of accepted requests. Arguably, a different metric, such as total number of affected lines, might be more appropriate, though any solution will have its flaws.

8. Conclusion

One sign of a successful publication is that it will attract a few letters from readers who have noticed typos or other problems. An unexpected benefit of publishing on GitHub is that to date forty-one readers have sent a total of *two hundred seventy-five* pull requests. Most of these fix typos, but a fair number make more substantial improvements.

There is much left to do! We hope others may be inspired to join us to expand and improve the book.

Acknowledgments. 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 how much more compact it is to use intrinsically-typed terms, thanks to David Darais. For inspiring our 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. This research was supported by EPSRC Programme Grant EP/K034413/1.

References

References

Abadi, M., Cardelli, L., Curien, P.L., Levy, J.J., 1991. Explicit substitutions. Journal of Functional Programming 1, 375–416.

Allais, G., Chapman, J., McBride, C., McKinna, J., 2017. Type-and-scope safe programs and their proofs, in: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, ACM. pp. 195–207.

Altenkirch, T., Reus, B., 1999. Monadic presentations of lambda terms using generalized inductive types, in: International Workshop on Computer Science Logic, Springer. pp. 453–468.

Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S., 2005. Mechanized metatheory for the masses: the poplmark challenge, in: International Conference on Theorem Proving in Higher Order Logics, Springer. pp. 50–65.

Barendregt, H., 1984. The Lambda Calculus. volume 103 of *Studies in Logic*. Elsevier.

Barendregt, H., Coppo, M., Dezani-Ciancaglini, M., 1983. A filter lambda model and the completeness of type assignment. Journal of Symbolic Logic 48, 931–940.

Berger, U., 1993. Program extraction from normalization proofs, in: International Conference on Typed Lambda Calculi and Applications, Springer. pp. 91–106.

Bove, A., Capretta, V., 2001. Nested general recursion and partiality in type theory, in: International Conference on Theorem Proving in Higher Order Logics, Springer. pp. 121–125.

Bove, A., Dybjer, P., Norell, U., 2009. A brief overview of agda–a functional language with dependent types, in: International Conference on Theorem Proving in Higher Order Logics, Springer. pp. 73–78.

Capretta, V., 2005. General recursion via coinductive types. Logical Methods in Computer Science 1.

Chapman, J.M., 2009. Type checking and normalisation. Ph.D. thesis. University of Nottingham.

Dagand, P.É., Scherer, G., 2015. Normalization by realizability also evaluates, in: Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015).

Danas, N., Nelson, T., Harrison, L., Krishnamurthi, S., Dougherty, D.J., 2017. User studies of principled model finder output, in: International Conference on Software Engineering and Formal Methods, Springer. pp. 168–184.

Engeler, E., 1981. Algebras and combinators. algebra universalis 13, 389–392.

Felleisen, M., Findler, R.B., Flatt, M., 2009. Semantics engineering with PLT Redex. By Press.

Goguen, H., McKinna, J., 1997. Candidates for substitution. Technical Report. Laboratory for Foundations of Computer Science, University of Edinburgh.

Gonthier, G., 2008. The four colour theorem: Engineering of a formal proof, in: Computer mathematics. Springer, pp. 333–333.

Gonthier, G., Asperti, A., Avigad, J., et al., 2013. A machine-checked proof of the odd order theorem, in: International Conference on Interactive Theorem Proving, Springer. pp. 163–179.

Hales, T., Adams, M., Bauer, G., Dang, T.D., Harrison, J., Le Truong, H., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T.T., et al., 2017. A formal proof of the Kepler conjecture, in: Forum of Mathematics, Pi, Cambridge University Press.

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

Huet, G., Kahn, G., Paulin-Mohring, C., 1997. The Coq proof assistant a tutorial. Rapport Technique 178.

Kästner, D., Leroy, X., Blazy, S., Schommer, B., Schmidt, M., Ferdinand, C., 2017. Closing the gap–the formally verified optimizing compiler compcert, in: SSS'17: Safety-critical Systems Symposium 2017, CreateSpace. pp. 163–180.

Kidney, D.O., 2019. Automatically and Efficiently Illustrating Polynomial Equalities in Agda. URL: https://doisinkidney.com/pdfs/bsc-thesis.pdf.

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

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al., 2009. sel4: Formal verification of an os kernel, in: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM. pp. 207–220.

Kokke, W., Swierstra, W., 2015. Auto in agda: Programming proof search using reflection, in: Hinze, R., Voigtländer, J. (Eds.), Mathematics of Program Construction, Springer International Publishing. pp. 276–301. URL: https://wenkokke.github.io/pubs/mpc2015.pdf, doi:10.1007/978-3-319-19797-5_14.

Leroy, X., 2009. Formal verification of a realistic compiler. Communications of the ACM 52, 107–115.

McBride, C., 2005. Type-preserving renaming and substitution. URL: https://personal.cis.strath.ac.uk/conor.mcbride/ren-sub.pdf. unpublished manuscript.

McBride, C., 2015. Turing-completeness totally free, in: International Conference on Mathematics of Program Construction, Springer. pp. 257–275.

O'Connor, L., Chen, Z., Rizkallah, C., Amani, S., Lim, J., Murray, T., Nagashima, Y., Sewell, T., Klein, G., 2016. Refinement through restraint: Bringing down the cost of verification, in: ICFP, pp. 89–102.

Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K., 2016. Functional big-step semantics, in: European Symposium on Programming, Springer. pp. 589–615.

Pfenning, F., 1992. A Proof of the Church-Rosser Theorem and Its Representation in a Logical Framework. Technical Report CMU-CS-92-186. Carnegie Mellon University. Pittsburgh, PA, USA.

Pierce, B.C., 2002. Types and programming languages. MIT press.

Pierce, B.C., 2009. Lambda, The Ultimate TA, in: ICFP, pp. 121–22.

Pierce, B.C., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Yorgey, B., 2010. Software foundations. URL: http://www.cis.upenn.edu/bcpierce/sf/current/index.html.

Plotkin, G.D., 1977. Lcf considered as a programming language. Theoretical Computer Science 5, 223–255.

Plotkin, G.D., 1993. Set-theoretical and other elementary models of the λ-calculus. Theoretical Computer Science 121, 351 – 409.

Reynolds, J.C., 2003. What do types mean?–from intrinsic to extrinsic semantics, in: Programming methodology. Springer, pp. 309–327.

Roşu, G., Şerbănuţă, T.F., 2010. An overview of the K semantic framework. Journal of Logic and Algebraic Programming 79, 397–434.

Schäfer, S., Tebbi, T., Smolka, G., 2015. Autosubst: Reasoning with de bruijn terms and parallel substitutions, in: Interactive Theorem Proving - 6th International Conference, Springer. pp. 359–374.

Scott, D., 1976. Data types as lattices. SIAM Journal on Computing 5, 522–587.

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

Wadler, P., 2018. Programming language foundations in agda, in: Formal Methods: Foundations and Applications (SBMF 2018), Springer.

Wright, A.K., Felleisen, M., 1994. A syntactic approach to type soundness. Information and computation 115, 38–94.