Intrinsically-Typed Definitional Interpreters for Imperative Languages

Poulsen, Casper; Rouvoet, Arjen; Tolmach, Andrew; Krebbers, Robbert; Visser, Eelco

DOI
10.1145/3158104

Publication date
2018

Document Version
Final published version

Published in
Proceedings of the ACM on Programming Languages

Intrinsically-Typed Definitional Interpreters for Imperative Languages

CASPER BACH POULSEN, Delft University of Technology, The Netherlands
ARJEN ROUVOET, Delft University of Technology, The Netherlands
ANDREW TOLMACH, Portland State University, USA
ROBBERT KREBBERS, Delft University of Technology, The Netherlands
EELCO VISSER, Delft University of Technology, The Netherlands

A definitional interpreter defines the semantics of an object language in terms of the (well-known) semantics of a host language, enabling understanding and validation of the semantics through execution. Combining a definitional interpreter with a separate type system requires a separate type safety proof. An alternative approach, at least for pure object languages, is to use a dependently-typed language to encode the object language type system in the definition of the abstract syntax. Using such intrinsically-typed abstract syntax definitions allows the host language type checker to verify automatically that the interpreter satisfies type safety. Does this approach scale to larger and more realistic object languages, and in particular to languages with mutable state and objects?

In this paper, we describe and demonstrate techniques and libraries in Agda that successfully scale up intrinsically-typed definitional interpreters to handle rich object languages with non-trivial binding structures and mutable state. While the resulting interpreters are certainly more complex than the simply-typed λ-calculus interpreter we start with, we claim that they still meet the goals of being concise, comprehensible, and executable, while guaranteeing type safety for more elaborate object languages. We make the following contributions: (1) A *dependent-passing style* technique for hiding the weakening of indexed values as they propagate through monadic code. (2) An Agda library for programming with *scope graphs* and *frames*, which provides a uniform approach to dealing with name binding in intrinsically-typed interpreters. (3) Case studies of intrinsically-typed definitional interpreters for the simply-typed λ-calculus with references (STLC+Ref) and for a large subset of Middleweight Java (MJ).

CCS Concepts: • **Theory of computation → Program verification**; *Type theory*; • **Software and its engineering → Formal language definitions**;

Additional Key Words and Phrases: definitional interpreters, dependent types, scope graphs, mechanized semantics, Agda, type safety, Java

ACM Reference Format:
Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-Typed Definitional Interpreters for Imperative Languages. *Proc. ACM Program. Lang.* 2, POPL, Article 16 (January 2018), 34 pages. https://doi.org/10.1145/3158104

Authors' addresses: Casper Bach Poulsen, Delft University of Technology, The Netherlands, c.b.poulsen@tudelft.nl; Arjen Rouvoet, Delft University of Technology, The Netherlands, a.j.rouvoet@tudelft.nl; Andrew Tolmach, Portland State University, Oregon, USA, tolmach@pdx.edu; Robbert Krebbers, Delft University of Technology, The Netherlands, r.j.krebbers@tudelft.nl; Eelco Visser, Delft University of Technology, The Netherlands, e.visser@tudelft.nl.

2475-1421/2018/1-ART16
https://doi.org/10.1145/3158104

1 INTRODUCTION

When we write an interpreter for one programming language using another one, we implicitly specify the dynamic semantics of the former (the *object language*) in terms of the semantics of the latter (the *host language*). A *definitional interpreter* is simply one written with the explicit goal of providing such a semantic specification. Provided that the host language already itself has a well-understood semantics and that it has good support for representing programs and defining evaluation functions over them, writing such an interpreter is an attractive way to produce a concise definition of the object language that is accessible to any programmer who knows the host language. Moreover, the interpreter can be executed to test the semantics by observing the behavior of specific object language programs. Functional languages with algebraic data types and pattern matching, such as ML and Haskell, are particularly well-suited to writing interpreters, typically structured as recursive traversals over the AST of the object program.

What about defining the *static* semantics of the object language? One possibility is to proceed in an analogous fashion and write a recursive *type-checker* over object language ASTs. While perhaps less compact than inference rule presentations of type systems, this form of static semantic specification has advantages; for example, like an interpreter, it is also executable. However, when we consider how to *integrate* the static and dynamic semantics, things are less satisfactory. One fundamental problem is that if the checker and interpreter are written separately, the latter will need to be defined over arbitrary ASTs, not just well-typed ones, and will therefore have to repeat many of the tests already made by the checker. Moreover, to *prove* formal coherence of the static and dynamic semantics, i.e., to show that well-typed programs will never fail those interpreter tests, we need to reason about host language semantics, which traditionally has meant using tools outside the host language itself. Mastering such tools is a significant hurdle for the ordinary host-language programmer, and even for experts, producing such proofs can be tedious and error-prone.

A well-known alternative approach [Altenkirch and Reus 1999; Augustsson and Carlsson 1999; Reynolds 2004], at least for pure, functional object languages, is to use a *dependently-typed* host language, such as Agda [Norell 2007], Idris [Brady 2013a], or Coq [Coq Development Team 2017]. The idea is to define an interpreter over an *intrinsically-typed AST*, i.e., a dependently-typed data structure in which only well-typed object programs can be expressed in the first place. Instead of writing a separate type-checker, the static semantics of the object language is encoded in the host-language definition of the AST constructors, and type safety of the object language follows from the fact that the interpreter code type-checks in the host language.

To illustrate, Figure 1 defines (most of) an interpreter for an expression language with arithmetic, conditionals, and variables. Expr Γ *t* is a well-typed expression of type *t* under a *type environment* Γ, which is a mapping from variables to types. Env Γ is a well-typed *dynamic environment* that maps the typed variables in Γ to values of those types. Environments are encoded as sequences, and variables are represented in de Bruijn style, as indices into those sequences. eval takes as input a well-typed expression of type *t* and dynamic environment and returns a well-typed value (Val *t*).[1] The case expressions[2] in the plus case of the eval function *only* need to match on int-typed values, i.e., numbers.[3] Agda automatically checks that these case distinctions are exhaustive, and that the interpreter is type sound.

[1] The ∀ {Γ *t*} in the type signature of eval is *implicit quantification*, which means that the function takes as input a context Γ and a type *t*, but that we will omit these when defining and calling the function. Agda will automatically infer what Γ and *t* is (or fails to type-check when this is not possible). Agda documentation is also available online: https://agda.readthedocs.io/

[2] case_of_ is an Agda function that evaluates its first argument and passes the result to its second argument, where λ{…} is Agda syntax for a pattern matching function.

[3] Agda supports overloaded constructor names, and we use num and bool as the name of both an expression and a value constructor. The num being matched against in the case expressions of the interpreter is the value constructor.

```
data Expr (Γ : Ctx) : Ty → Set where
  bool  :  Bool → Expr Γ bool
  num   :  ℤ → Expr Γ int
  var   :  ∀ {t} → t ∈ Γ → Expr Γ t
  if    :  ∀ {t} → Expr Γ bool →
           Expr Γ t → Expr Γ t →
           Expr Γ t
  plus  :  Expr Γ int → Expr Γ int →
           Expr Γ int

data Ty : Set where
  bool : Ty
  int  : Ty

Ctx = List Ty
```

```
data Val : Ty → Set where
  bool  :  Bool → Val bool
  num   :  ℤ → Val int

Env : Ctx → Set
Env Γ = All Val Γ

eval : ∀ {Γ t} → Expr Γ t → Env Γ → Val t
eval (bool b)   E  =  bool b
eval (num x)    E  =  num x
eval (var x)    E  =  lookup E x
eval (if c t e) E  =  case (eval c E) of λ{ (bool b) →
                      if b then (eval t E) else (eval e E)}
eval (plus e₁ e₂) E = case (eval e₁ E) of λ{ (num z₁) →
                      case (eval e₂ E) of λ{ (num z₂) →
                      num (z₁ + z₂) }}
```

Fig. 1. Intrinsically-typed interpreter for an expression language with arithmetic, conditionals, and variables

A dependently-typed definitional interpreter like this one represents a sweet spot for programming language definitions: it delegates the safety proof work to the host-language type checker; it is concise yet comprehensible; and we can still run the interpreter to validate that it implements the intended semantics. Of course, there is no free lunch: to use this technique, programmers must master the complexities of dependent typing. However, for simple object languages, the approach looks very attractive.

But can we write definitional interpreters in this style for larger and more realistic object languages without sacrificing these benefits? This paper argues that the dependently-typed approach can indeed scale to describe languages with features including mutable state, multiple kinds of binding, inheritance, and sub-typing. To do this, we have invented new techniques for encoding such features into intrinsically-typed syntax, monads, and general-purpose libraries for defining binding structure. These techniques are the principal technical contribution of the paper. We illustrate and describe these techniques by developing a series of interpreters throughout the paper; the source code for these are available in the accompanying artifact [Bach Poulsen et al. 2017]. We also present case studies demonstrating the applicability of these techniques to the simply-typed λ-calculus with references (STLC+Ref) and to a large subset of Middleweight Java (MJ) [Bierman et al. 2003]. Here we briefly summarize the main challenges that we have addressed.

1.1 Mutable State

Whenever we use a pure host language to define an object language with mutable state, we must thread state through the interpreter explicitly. For safe dependently-typed interpreters, we must also thread facts about the state. If we extend the interpreter from Figure 1 with a mutable store and ML-style references, its signature must reflect that evaluation of terms *may* extend the store, but only in a well-typed manner; i.e. the types of existing store locations do not change. This is illustrated in the following extended type signature for STLC with references (STLC+Ref), where Σ ranges over *store types* (i.e., mappings from store locations to types) [Harper 1994; Pierce 2002]:

```
eval : ∀ {Γ t Σ} → Expr Γ t → Env Γ Σ → Sto Σ → (∃ λ Σ′ → (Val t Σ′) × (Sto Σ′) × (Σ ⊑ Σ′))
```

Since values and environments may contain store references, Val and Env are now indexed by the store type; and eval now returns both a value and a store described by some Σ'.[4] The final $\Sigma \sqsubseteq \Sigma'$ in the type signature above witnesses that Σ is a list prefix of Σ', i.e., that existing store locations remain invariant under store extensions. Since environments and values are indexed by a store type, and since we sometimes need to use values and environments in the context of an updated store, the _⊑_ type is used to "weaken" values and environments, such that the interpreter type checks. The following case for addition of numbers illustrates how mutable state affects intrinsically-typed interpreters (where E ranges over environments and μ ranges over stores):

```
eval (plus e₁ e₂) E μ =
  case (eval e₁ E μ) of λ{ (Σ , (num z₁) , μ′ , ext ) →
  case (eval e₂ (weaken-env ext E) μ′) of λ{ (Σ′ , (num z₂) , μ′′ , ext′ ) →
    (Σ′ , (num (z₁ + z₂)) , μ′′ , ext ◎ ext′) }}
```

Here, the weaken-env function is used to weaken the store type indices of environments, and the store extension composition function _◎_ is used to compose store extension facts in order to construct a witness that the store that is the result of a function application is an extension of the input store, before making the function call. The types of weaken-env and _◎_ are:

```
weaken-env  :  ∀ {Σ Σ′ Γ} → Σ ⊑ Σ′ → Env Γ Σ → Env Γ Σ′
_◎_         :  ∀ {Σ₁ Σ₂ Σ₃ : List Ty} → Σ₁ ⊑ Σ₂ → Σ₂ ⊑ Σ₃ → Σ₁ ⊑ Σ₃
```

This code is far less attractive than that of the STLC interpreter we saw earlier: threading of state and proof witnesses causes verbosity, and some proof witnesses need to be computed within the interpreter itself. Of course, it is well-known how to hide this sort of "plumbing" in a (non-dependent) functional language: write a monadic interpreter. Unfortunately, monads do not straightforwardly scale to safely thread intrinsically-typed stores in the dependent setting. In Section 3 of this paper we describe and analyze the problem, and propose techniques for using monads to safely thread stores in intrinsically-typed definitional interpreters.

1.2 Principled Binding

To type an interpreter it is not enough to have intrinsically typed syntax. All the other semantic components involved, such as environments and stores, must also be intrinsically typed. Existing semantic treatments of imperative languages often use several kinds of auxiliary data structures. As an example, consider Middleweight Java (MJ) [Bierman et al. 2003], a core calculus that aims to capture the key imperative features of the Java language. MJ utilizes local variable environments; objects; class tables that map class identifiers to methods; and heaps that map locations to objects. Each of these components is equipped with its own notion of well-typedness, and it is not always obvious how to encode them in a dependently-typed setting. In particular, for binding patterns that go beyond simple lexical scoping, we need to find an analogue to de Bruijn indices, which are a mainstay of intrinsically-typed syntax.

Recent work introduced *scope graphs* [Néron et al. 2015] as an approach to static name binding and resolution that gives rise to a notion of well-typed *frames* [Bach Poulsen et al. 2016]. The scopes-and-frames approach provides a systematic connection between program syntax and memory: each location in the heap is a frame that is described (or typed) by a *scope*, where a scope is an abstraction over a set of AST nodes. Scope graphs thus provide a type discipline that systematically

[4] $\exists\,(\lambda\, x \to P)$ is Agda notation for dependent pairs (e.g., $\exists x.P$), where P is a type that may depend on the value x.

connects structured memory objects (frames) to the object language AST. The *resolution calculus* [Néron et al. 2015] and *constraint language* [van Antwerpen et al. 2016] provide a way to resolve identifier references in a program automatically by generating its scope graph and resolution paths (corresponding to generalized de Bruijn indices) through the graph.

In Section 4 of this paper, we show how to apply the scopes-and-frames approach in an intrinsically-typed setting, and we provide a small Agda library that can be used to define binding, memory allocation, and memory access operations for a range of object languages, including a large subset of MJ.

1.3 The Bigger Picture

Our techniques enable us to write definitional interpreters that are similar in structure to those we might write in a non-dependent functional language, but are also type-sound *by construction*. This removes the need for separate type safety proofs; more importantly, it makes type safety a *guiding contract* for engineering interpreters. By working in a dependently-typed host language, this contract is enforced interactively, i.e., while writing the interpreter. We believe that this is a promising methodology for engineering type-sound languages, and our experience with constructing the interpreters for this paper suggests that following the intrinsically-typed approach can be more pragmatic than constructing and maintaining a separate safety proof.

The techniques presented in this paper bring the benefits of intrinsically-typed interpreters to a large class of imperative languages, and we hope also to a wider audience of language designers. To make the techniques useful in practice, however, we need support for actually defining a *full language*, not just an interpreter. Our vision is to develop a Language Designer's Workbench [Visser et al. 2014] that supports: (1) declarative definition of name and type analysis using *scope graphs* [Néron et al. 2015]; (2) declarative and modular definitional interpreters; and (3) a systematic approach to verifying that such definitional interpreters and type systems are *sound by construction*. This vision is being actively developed in Spoofax [Kats and Visser 2010], using the NaBL name binding language [Konat et al. 2012; van Antwerpen et al. 2016] for declarative name binding and type analysis to realize the first goal, and DynSem [Vergu et al. 2015] for modular operational semantics to realize the second goal. This work explores how to realize the third goal.

In previous work we have shown how to systematically relate static semantics using scope graphs to semantics using *frames* for dynamic memory layout [Bach Poulsen et al. 2016]. This paper explores type safety checking for definitional interpreters using scopes-and-frames and (mostly) off-the-shelf dependent types in Agda. To integrate this solution into Spoofax we can generate intrinsically-typed syntax specifications from the constraint-based NaBL type checker [van Antwerpen et al. 2016], and develop a dependently-typed extension of DynSem. Such an extension would enable better error reporting, and would provide specialized syntax for reducing syntactic noise in interpreters (the techniques we present let us write interpreters that are mostly, but not entirely, free of "proof plumbing" – see Section 3).

1.4 Contributions

In this paper we systematically analyze the techniques needed to write dependently-typed definitional interpreters with mutable state in a way that preserves conciseness and delegates the type safety proof work to a dependently-typed host language. We present the following contributions:

- A *dependent-passing style* technique, based on *monadic strength* [Kock 1972; Moggi 1991], for hiding the weakening of indexed values as they propagate through monadic code.
- An Agda library for programming with *scope graphs* [Néron et al. 2015] and *frames* [Bach Poulsen et al. 2016], which provides a uniform approach to dealing with name binding in

intrinsically-typed interpreters. Using scopes and frames, memory operations are correct-by-construction, which alleviates the need for different encodings for name binding. Instead, it requires object language type systems to be defined in terms of scope graphs.
- Case studies of intrinsically-typed definitional interpreters for STLC+Ref and a large subset of MJ, including sub-typing. To the best of our knowledge, our MJ case study is the first intrinsically-typed semantics for a Java subset. Unlike previous formalizations [Bierman et al. 2003; Igarashi et al. 2001; Klein and Nipkow 2006; Delaware et al. 2011; Mackay et al. 2012] of Java subsets, our interpreter is sound-by-construction, i.e., there is no need for accompanying proofs of safety.
- The presented techniques and case studies are implemented in Agda and made available as code artifacts accompanying this paper [Bach Poulsen et al. 2017].

We use the Agda language throughout, but most of our constructions can be replicated easily in Idris. With considerably more work, the main ideas should also be portable to Coq and perhaps to (extended versions of) Haskell.

Organization. The remainder of this paper is organized as follows. Section 2 gives a more detailed description of an intrinsically-typed syntax interpreter for STLC. Section 3 extends the approach to STLC+Ref, gives a detailed analysis of the difficulties of using monadic style in this context, and describes how to hide all index type adjustments in the monad. Section 4 describes the scopes-and-frames library and shows how it can be used to reimplement the STLC interpreter. Section 5 describes an intrinsically-typed definitional interpreter for Middleweight Java, defined using the techniques from the previous sections. Section 6 discusses related work, and Section 7 concludes.

2 A DEFINITIONAL INTERPRETER FOR STLC

Intrinsically-typed interpreters for pure languages are part of the folklore of dependently-typed programming. In this section we review the approach by defining an interpreter for the simply-typed λ-calculus (STLC) to set the stage for exploring interpreters for more complex object languages.

2.1 Syntax

The idea of intrinsically-typed interpreters is that we internalize the object language's type judgment in the host language types used to describe object language syntax. In Agda we use *indexed inductive types* (or *inductive type families*) to define this *intrinsically-typed syntax*. For STLC we first define the syntax of types (Ty) and type contexts (Ctx):[5]

```
data Ty : Set where
  unit : Ty
  _⇒_  : (t u : Ty) → Ty
Ctx = List Ty
```

STLC expressions can refer to variables in Γ and are themselves typed. We define them as an Agda inductive type, parameterized by that type environment Γ and indexed by their type:

```
data Expr (Γ : Ctx) : Ty → Set where
  unit : Expr Γ unit
  var  : ∀ {t} → t ∈ Γ → Expr Γ t
  lam  : ∀ {t u} → Expr (t :: Γ) u → Expr Γ (t ⇒ u)
  _·_  : ∀ {t u} → Expr Γ (t ⇒ u) → Expr Γ t → Expr Γ u
```

$$\frac{\Gamma \vdash f : t \Rightarrow u \qquad \Gamma \vdash e : t}{\Gamma \vdash (f \cdot e) : u}\ \text{App}$$

[5] In Agda one can define mixfix functions using underscores for where the arguments are supposed to go. The function type _⇒_ from the syntax definition above, is thus used as $a \Rightarrow b$.

Each constructor of this type corresponds both to a standard syntactical construct and a typing rule of STLC. Note for example the similarity between the type of the constructor _·_ (which is the syntax for function-application) and its usual typing rule. The var rule does not use a name to identify a variable, but a well-typed de Bruijn index $t \in \Gamma$ which witnesses the existence of an element t in Γ as defined by the _∈_ type:

```
data _∈_ {A : Set} (x : A) : List A → Set where
  here  : ∀ {xs}              →  x ∈ (x :: xs)
  there : ∀ {y xs}  →  x ∈ xs  →  x ∈ (y :: xs)
```

While such de Bruijn indices are useful indeed for intrinsically-typed data and functions over this data (as we shortly illustrate), it is not always obvious how a de Bruijn-indexed program corresponds to a source program; and it can be painful to write test expressions when we want to run our interpreter when object language expressions use de Bruijn indices. While it is possible to translate programs without de Bruijn indices to programs with de Bruijn indices, the danger of relying extensively on dependent types in intrinsically-typed syntax definitions is that it can become challenging to relate to the concrete syntax of a program. In Section 4, we review how recent work on *scope graphs* and the *name resolution calculus* [Néron et al. 2015; van Antwerpen et al. 2016] provides a systematic approach to constructing paths corresponding to generalized de Bruijn indices, and how to define intrinsically-typed syntax and interpreters using scope graphs and resolution paths.

2.2 Values and Environments

To define our interpreter, we first need intrinsically-typed values and environments. An intrinsically-typed value is closed, and therefore it is only indexed by a type. Environments are typed by type contexts in pointwise fashion, such that we have a typed value for each type in the context. This is encoded using the type All from the Agda standard library [Danielsson and Norell 2017], which asserts that a certain predicate P (in this case Val; a predicate over object language types) holds for each entry in a list (in this case a context Γ):

```
data All {A : Set} (P : A → Set) : List A → Set where
  []  : All P []
  _::_ : ∀ {x xs} → P x → All P xs → All P (x :: xs)
```

There are two kinds of values for STLC: the unit value of type unit, and a closure value ⟨_, _⟩ which stores the lexical environment of a function, together with that function's body which is closed over that lexical environment extended with an argument. This results in the following mutually-dependent definitions of values and environments:

```
mutual
  data Val : Ty → Set where
    unit  : Val unit
    ⟨_,_⟩ : ∀ {Γ t u} → Expr (t :: Γ) u → Env Γ → Val (t ⇒ u)

  Env : Ctx → Set
  Env Γ = All Val Γ
```

The fact that environments are typed by contexts Γ permits us to look up values of a particular type t in an environment Env Γ using a witness $t \in \Gamma$:

```
lookup : ∀ {A P xs}{x : A} → All P xs → x ∈ xs → P x
```

2.3 Totality and Anticipating Effects: Fuel and Monads

Agda is a total language and, although STLC is strongly-normalizing, that is not self-evident from the defined semantics and Agda will complain that it cannot determine whether eval will terminate. We address this issue here and throughout the rest of this paper using a *fuel counter* [Siek 2013; Owens et al. 2016; Amin and Rompf 2017] in our interpreters. The idea is that the evaluation function has a fuel argument which is decreased every time our interpreter makes a recursive call. When the interpreter is called with a sufficient (but finite) amount of fuel, a value is returned.[6] On the other hand, if the counter reaches zero, the interpreter abruptly terminates and indicates it ran out of fuel. This makes all evaluation functions structurally recursive in the fuel argument.

To deal with the source of abrupt termination due to running out of fuel, and to anticipate the extension of our interpreter with mutable state, we use a monad for our evaluation function. Since the monad is going to hide an environment and since our interpreter needs to know that the hidden environment is typed in the same context as the expression, we index our monad by a type context:

```
M : (Γ : Ctx) → (A : Set) → Set
M Γ A  =  Env Γ → Maybe A
```

The option type Maybe indicates the possibility that the interpreter runs out of fuel. Our monad defines the usual monadic operations for bind and return, as well as the operations getEnv (to retrieve the current environment) and usingEnv (to run a computation in a specified environment). Finally, we define an operation timeout which abruptly terminates the computation when running out of fuel:

```
_»=_     : ∀ {Γ A B} → M Γ A → (A → M Γ B) → M Γ B
return   : ∀ {Γ A} → A → M Γ A
getEnv   : ∀ {Γ} → M Γ (Env Γ)
usingEnv : ∀ {Γ Γ′ A} → Env Γ → M Γ A → M Γ′ A
timeout  : ∀ {Γ A} → M Γ A
```

2.4 The Well-Typed Interpreter

Using the intrinsically-typed syntax and monad above, our well-typed interpreter is written as:

```
eval : ℕ → ∀ {Γ t} → Expr Γ t → M Γ (Val t)
eval zero     _          =  timeout
eval (suc k)  unit       =  return unit
eval (suc k)  (var x)    =  getEnv »= λ E → return (lookup E x)
eval (suc k)  (lam e)    =  getEnv »= λ E → return ⟨ e , E ⟩
eval (suc k)  (e₁ · e₂)  =  eval k e₁ »= λ{ ⟨ e , E ⟩ →
                            eval k e₂ »= λ v →
                            usingEnv (v :: E) (eval k e) }
```

3 EXTENDING WITH MUTABLE STATE

In this section we analyze how to extend our STLC interpreter with ML-style references (STLC+Ref).

[6] More precisely, it must hold for our interpreters that, whenever the interpreter returns a value for an amount of fuel k, then the interpreter returns that value for any $k' \geq k$. This property is not essential for type safety, but may be necessary for proving other properties about our interpreters.

3.1 Syntax

The intrinsically-typed syntax is as before (Section 2.1), with the addition of the following object language type:

```
ref  : Ty → Ty
```

and the following object-language constructs for ML-style references, typed in accordance with the usual typing rules for ML-style references [Milner et al. 1997; Pierce 2002]:

```
ref   : ∀ {t} → Expr Γ t → Expr Γ (ref t)
!_    : ∀ {t} → Expr Γ (ref t) → Expr Γ t
_:=_  : ∀ {t} → Expr Γ (ref t) → Expr Γ t → Expr Γ unit
```

3.2 Stores and Values

To extend STLC with references, we need a notion of well-typed stores. We type the store with respect to a StoreTy, which, like the type context, is defined as a list of types:

```
StoreTy = List Ty
```

We apply the same construction as in Section 2.2 and use the All type to ensure that a store is in one-to-one correspondence with this StoreTy. Since values in STLC+Ref may refer to store locations, we refine our notion of a well-typed value by indexing these by store types. For brevity we will not repeat constructors from previous definitions when we extend a type:

```
data Val : Ty → (Σ : StoreTy) → Set where
  loc  : ∀ {Σ t} → t ∈ Σ → Val (ref t) Σ

Env : (Γ : Ctx) → (Σ : StoreTy) → Set
Env Γ Σ = All (λ t → Val t Σ) Γ
```

Now we can define a well-typed store using Agda's All type, similar to how we defined environments:

```
Store : (Σ : StoreTy) → Set
Store Σ = All (λ t → Val t Σ) Σ
```

This definition establishes a pointwise relation between store types and stores: for each type t in the store type Σ, a store contains a value which is typed under that Σ and which has type t. Store updates and lookups are implemented generically on the type All and their signatures (specialized for our use case) are:

```
lookup-store : ∀ {Σ t} → t ∈ Σ → Store Σ → Val t Σ
update-store : ∀ {Σ t} → t ∈ Σ → Val t Σ → Store Σ → Store Σ
```

3.3 Towards a Well-Typed Interpreter

Extending our interpreter involves threading stores in a way that makes explicit that the store is monotonically extended; i.e., the type of each location never changes. Ideally, the monad should deal with all details of propagating effects in a well-typed manner. We investigate and analyze how to attain this goal.

First attempt: Indexed Monads. For languages with ML-style references, there is no way to know statically the exact effect of an arbitrary expression on the type of the store without either interpreting it, or augmenting the object-language type system and intrinsically-typed syntax. Consider for example the following expression:

```
let c = ref 42 in
let d = if (x > 0) then c else ref 11 in
(* ... *)
```

The store type needed for the inner let-expression is determined not just by static information, but also by the run-time value of x. Thus "regular" indexed monads [Atkey 2009] where we have to give the final store type Σ' upfront as a Hoare-style post-condition, as summarized by the following type signature, are not a good fit for an interpreter for such a language:

```
M : (Γ : Ctx) → (A : Set) → (Σ : StoreTy) → (Σ′ : StoreTy) → Set  -- No good!
```

Second attempt: Monads over an Indexed Set. Instead, following McBride [2011], we can use a *monad over an indexed set*:[7]

```
M : (Γ : Ctx) → (P : StoreTy → Set) → (Σ : StoreTy) → Set
M Γ P Σ = (E : Env Γ Σ) → (μ : Store Σ) → Maybe (∃ λ Σ′ → Store Σ′ × P Σ′ × Σ ⊑ Σ′)
```

This monad is indexed by a context Γ, a store type Σ, and a predicate P dependent on the store type. The monad lets us under-specify the final store type for a given computation, by providing a *predicate* over store types, which must hold for the final store. In the context of the use case that we are concerned with in this paper, the Val type constitutes such a predicate. Specifically, we can use the monad M to type our extended evaluation function for STLC+Ref as follows:

```
eval : ℕ → ∀ {Σ Γ t} → Expr Γ t → M Γ (Val t) Σ
```

Considering our implementation of M above, this type signature for eval says that evaluating a well-typed expression in the context of a well-typed environment and a well-typed store either returns a well-typed value under a well-typed store extension (where the extension is witnessed by _⊑_), or it runs out of fuel. The bind of the monad can be defined as follows:

```
_»=_ :  ∀ {Σ Γ}{P Q : StoreTy → Set} →
        (f : M Γ P Σ) → (g : ∀ {Σ′} → P Σ′ → M Γ Q Σ′) → M Γ Q Σ
```

For a computation f »= g in this monad, we cannot in general determine statically what the store type is after running f. Thus the function g is defined for *any* store-type Σ'. The implementation of f »= g requires a guarantee that environments remain well-typed after running f. The following weakening functions (or lemmas, if you will) prove that values and environment types are preserved under store extensions $\Sigma \sqsubseteq \Sigma'$:

```
weaken-val  : ∀ {t}{Σ Σ′ : StoreTy} → Σ ⊑ Σ′ → Val t Σ → Val t Σ′
weaken-env  : ∀ {Γ}{Σ Σ′ : StoreTy} → Σ ⊑ Σ′ → Env Γ Σ → Env Γ Σ′
```

We also prove a straightforward lemma stating store extension transitivity:

```
_◎_ : ∀ {Σ Σ′ Σ′′ : StoreTy} → Σ ⊑ Σ′ → Σ′ ⊑ Σ′′ → Σ ⊑ Σ′′
```

Using these lemmas, we can implement bind so that it weakens environments appropriately and applies store extension transitivity to prove that the final store after the second computation is also an extension of the initial one:

```
(f »= c) E μ = case (f E μ) of λ{
  nothing → nothing ;
  (just (_ , μ′ , x , ext)) → case (c x (weaken-env ext E) μ′) of λ{
    nothing → nothing ;
    (just (_ , μ′′ , y , ext′)) → just (_ , μ′′ , y , ext ◎ ext′) }}
```

[7] Indexed monads are specialized instances of monads over indexed sets [McBride 2011].

We can define the same monadic operations for environments as for STLC (Section 2.3), but updated to thread stores and store types. Additionally, we define monadic operations for storing a value in a new well-typed store location, and for dereferencing and updating existing store locations:[8]

```
store  : ∀ {Σ Γ t} → Val t Σ → M Γ (Val (ref t)) Σ
deref  : ∀ {Σ Γ t} → t ∈ Σ → M Γ (Val t) Σ
update : ∀ {Σ Γ t} → t ∈ Σ → Val t Σ → M Γ (λ _ → ⊤) Σ
```

With these operations in place, we seem ready to define our interpreter. But attempting to do so leads to the realization that our monad alone does not suffice to deal with mutable state! Consider the application case of the interpreter for STLC:

```
eval (suc k) (e₁ · e₂) = eval k e₁ >>= λ{ ⟨ e , E ⟩ →
                         eval k e₂ >>= λ v →
                         usingEnv (v :: E) (eval k e) }
```

This case does not type check. There is a discrepancy between the store type indices for the value v and the environment E in the last line. The problem is that the bind of our monad is ignorant of the store type extension. Recall that the definition of bind f >>= g requires g to accept any store type Σ'. In other words, g cannot make any assumptions about Σ'; and, in particular, we cannot in general assume that Σ' is an extension of Σ. McBride [2011] describes this as a *demonic bind*, since the Σ' is chosen by a "demonic" adversary.

Third attempt: explicit weakening. One way to avoid the problematic discrepancy in store-type indices is to make the assumption that Σ' is an extension of Σ explicit in the type of the bind; i.e., we can add a lower-bound Σ' by threading a store extension witness as follows:

```
_>>=_ : ∀ {Σ Γ}{P Q : StoreTy → Set} → (f : M Γ P Σ) →
        (g : ∀ {Σ'} → Σ ⊑ Σ' → P Σ' → M Γ Q Σ') → M Γ Q Σ
```

By threading store extension witnesses, we can explicitly weaken values from previous computations which are only known to be well-typed under a previous store, to use them in the context of a computation with an extended store. This is enough to implement our interpreter for STLC+Ref (we show the relevant cases):

```
eval (suc k) (e₁ · e₂)   = eval k e₁ >>= λ{ ext ⟨ e , E ⟩ →
                           eval k e₂ >>= λ{ ext' v →
                           usingEnv (v :: (weaken-env ext' E)) (eval k e) }}
eval (suc k) (ref e)     = eval k e >>= λ _ v → alloc v
eval (suc k) (! e)       = eval k e >>= λ{ _ (loc l) →
                           deref l }
eval (suc k) (e₁ := e₂)  = eval k e₁ >>= λ{ ext (loc l) →
                           eval k e₂ >>= λ{ ext' v →
                           update (weaken-loc ext' l) v >>= λ _ _ →
                           return unit }}
```

While the interpreter above type-checks, this comes at the price of propagating additional information and applying noisy weakening functions. This noise only becomes more pronounced when we consider more complex languages, where a value may be carried across multiple binds before it is used, so that store type extension facts need to be composed. In the next section we

[8] The ⊤ type is the unit type. Thus (λ _ → ⊤) is the trivially true store type predicate.

consider an alternative technique to avoid explicit weakening in order to better preserve the conciseness and clarity of our definitional interpreters for languages with mutable state.

3.4 Dependent-Passing Style

Since monads are known to be good for hiding "plumbing" in interpreters, we might hope to move the necessary weakenings on store-indexed values inside a monadic operator. But unfortunately, the values that need weakening can escape from the control of the monad. Consider again the case for object language application in the STLC+Ref interpreter of the previous section, where we extend environment E with value v. The challenge here is to weaken the store-type index of E to match that of v. We cannot perform that weakening inside the bind operator, because E is passed around outside of the monad using the host language's support for first-class functions. In other words, the problem is that the innermost λ (which binds v) has a free variable E that is indexed by a store type. If we *close* the functions passed to bind with respect to store-indexed values, we can encapsulate the necessary weakenings.

In this section we describe a monadic operator `_^_` that carries values across binds such that they remain well-typed under store extensions:

```
eval (suc k)  (e₁ · e₂)  =  eval k e₁ »= λ{ ⟨ e , E ⟩ →
                            (eval k e₂ ^ E) »= λ{ (v , E) →
                            usingEnv (v :: E) (eval k e) }}
```

The `_,_` constructor matched by the innermost function in the second line of `eval` above is a store type indexed product that intersects two predicates by pairing them with respect to the same store type index:

```
data _⊗_ (P Q : StoreTy → Set) : (Σ : StoreTy) → Set where
  _,_ : ∀ {Σ} → P Σ → Q Σ → (P ⊗ Q) Σ
```

We need to define `_^_` so that it pairs a monadic computation and a value, by "lifting" the value into the monad. The following type for `_^_` is almost—but not quite—the one we want:

```
_^_  :  ∀ {Σ Γ}{P Q : StoreTy → Set} → M Γ P Σ → Q Σ → M Γ (P ⊗ Q) Σ   -- Almost!
```

We also need to know that Q is weakenable, such that the input $Q\,\Sigma$ will be well-typed in whatever the final store of the monadic computation is. Thus we define `_^_` as follows:

```
_^_  : ∀ {Σ Γ}{P Q : StoreTy → Set} → ⦃ w : Weakenable Q ⦄ → M Γ P Σ → Q Σ → M Γ (P ⊗ Q) Σ
(f ^ x) E μ = case (f E μ) of λ {
  nothing → nothing ;
  (just (Σ , μ′ , y , ext)) → just (Σ , μ′ , (y , weaken ext x) , ext) }
```

The `⦃ · ⦄` notation is for Agda instance arguments [Devriese and Piessens 2011]. Agda will attempt to find suitable values for these arguments automatically, so that `⦃ w : Weakenable Q ⦄` is similar to a Haskell type class, in that it asserts that there exists an instance of `Weakenable` Q; where `Weakenable` is a record type that defines a weakening function:

```
record Weakenable (P : StoreTy → Set) : Set where
  field weaken : ∀ {Σ Σ′} → Σ ⊑ Σ′ → P Σ → P Σ′
```

We can straightforwardly provide instances of this record for environments, values, and locations. Doing so lets us define the interpreter for STLC+Ref in a way that it becomes free of explicit weakening, by using the `_^_` operator to close all host language functions over store-indexed variables:

```
eval : ℕ → ∀ {Σ Γ t} → Expr Γ t → M Γ (Val t) Σ
eval zero     _          =  timeout
eval (suc k)  unit       =  return unit
eval (suc k)  (var x)    =  getEnv >>= λ E → return (lookup E x)
eval (suc k)  (lam e)    =  getEnv >>= λ E → return ⟨ e , E ⟩
eval (suc k)  (e₁ · e₂)  =  eval k e₁ >>= λ{ ⟨ e , E ⟩ →
                            (eval k e₂ ^ E) >>= λ{ (v , E) →
                            usingEnv (v :: E) (eval k e) }}
eval (suc k)  (ref e)    =  eval k e >>= λ v → store v
eval (suc k)  (! e)      =  eval k e >>= λ{ (loc l) →
                            deref l }
eval (suc k)  (e₁ := e₂) =  eval k e₁ >>= λ{ (loc l) →
                            (eval k e₂ ^ l) >>= λ{ (v , l) →
                            update l v >>= λ _ →
                            return unit }}
```

We say that this interpreter is written in *dependent-passing style*, because all the store-dependent variables are explicitly passed into and out of the monadic operations. Using this approach lets us hide all plumbing in the monad. This interpreter in dependent-passing style has the same structure as a non-dependently-typed monadic interpreter, except: (1) this interpreter does not need to propagate or consider cases that break type preservation (it is safe by construction); and (2) it uses `_^_` to carry values across monadic binds. The technique is general (indeed, it corresponds to *monadic strength* [Kock 1972; Moggi 1991], as we will argue in Section 3.6) and is transferable to other host languages (like Idris or Coq) and object languages (Section 5 describes how to write a definitional interpreter for MJ using the technique), using only off-the-shelf dependently typed programming and Agda's counterpart to type class resolution to make `_^_` more generic.

3.5 An Alternative to Dependent-Passing Style

We might define a notion of monadic bind similar to the interpreter we defined in Section 3.3, but using type classes to propagate extension facts, and define a notion of weakening that automatically resolves and composes the required weakening facts. For example, we could define a type class IsIncludedOnce whose instances witness a single extension fact, and a type class IsIncluded whose instances are the reflexive-transitive closure of IsIncludedOnce:

```
instance
  included-refl  : ∀ {Σ} → IsIncluded Σ Σ
  included-step  : ∀ {Σ Σ′ Σ′′} ⦃ i₁ : IsIncludedOnce Σ Σ′ ⦄ ⦃ i₂ : IsIncluded Σ′ Σ′′ ⦄ →
                   IsIncluded Σ Σ′′
```

We could then define a wk function in terms of the IsIncluded type class, to automatically perform a proof search to compose relevant IsIncludedOnce witnesses coming from the context:

```
wk  :  ∀ {Σ Σ′ P} → ⦃ w : Weakenable P ⦄ → P Σ → ⦃ i : IsIncluded Σ Σ′ ⦄ → P Σ′
```

In theory, this should work, but in practice Agda (v2.5.3) does not support overlapping instances during proof search (such as the choice between the included-refl or included-step instance). Idris (v1.1.1) has a similar restriction. It is possible to encode the proof search in Coq (v8.6), but Coq's limited support for dependent pattern matching makes it otherwise unattractive for writing dependently-typed programs.

It is possible to encode proof search as described above, but should we? The proof search approach results in less noisy interpreters, but it also relies on specific capabilities of the host language. Porting the approach to different dependently-typed host languages may give varying results (as we already see when we compare Agda, Idris, and Coq). Of course, our _^_ operation is also defined using proof search (for Weakenable), but it could have been encoded by packaging the weakening function in a number of ways that does not rely on proof search; see, for example, the next section. In the next section we also argue that _^_ is in fact the well-established notion of monadic strength. Thus our proposed dependent-passing style is based on standard theoretical foundations and requires only standard dependently-typed programming, which makes it conceptually attractive. From a pragmatic viewpoint, however, we would prefer to write interpreters without explicit use of _^_, and the proof search using the IsIncluded types that we sketched above is innocuous and (we think) reasonably understandable. A solution that we have not explored, but which we believe is an interesting avenue for future work, is host language support for monadic strength.

3.6 Monadic Strength

We illustrated in Section 3.3 how a monad over an indexed set is useful for threading intrinsically-typed stores in a safe way. In our context, the notion of indexed set for our monad is store type indexed sets. As a matter of fact, the indexed sets of our monad have even more structure: each set is also associated with a notion of weakening. So, instead of expressing our monad using the following types (equivalent to M from Section 3.3 and 3.4):

```
ISet = (Σ : StoreTy) → Set
M : (Γ : Ctx) → (A : ISet) → ISet
```

we could instead express our monad over an indexed set P for which there is a notion of weakening:

```
WSet = (∃ λ (P : StoreTy → Set) → Weakenable P)
M′ : (Γ : Ctx) → (A : WSet) → WSet
```

We can also define a notion of products for WSet (_⊗_ below), and a notion of arrows (_⇒_ below) between WSets:

```
_⊗_  :  WSet → WSet → WSet
_⇒_  :  WSet → WSet → Set
```

Given these definitions, our _^_ operator is typed as:

```
_^_ : {Γ : Ctx} {P Q : WSet} → (P ⊗ M′ Γ Q) ⇒ M′ Γ (P ⊗ Q)
```

This type signature coincides with monadic strength. In the artifact accompanying this paper we have proven that a simpler variant of M′ (which ignores non-termination and where the type context Γ and dynamic environment is not a part of the monad) together with _⊗_ forms a *strong monad* [Kock 1972; Moggi 1991] over the category whose objects are given by WSet and whose arrows (morphisms) are given by _⇒_. We have also proven that this category is Cartesian-closed, that is, that we can define a notion of exponentiation (functions) in the category (denoted by _⇝_ below) for WSet. It is tempting to reformulate our monadic operations and interpreters to reflect these insights. For example, we can type the monadic bind inside the category itself as:

```
_»=_ : ∀ {Γ}{P Q : WSet} → ((M′ Γ P) ⊗ (P ⇝ M′ Γ Q)) ⇒ M′ Γ Q
```

This encoding exposes the true structure of the monadic operations we have used to define the interpreters in this section. However, defining interpreters in this category poses a pragmatic challenge: one can no longer use plain Agda functions, but instead has to use exponents in the category. As such, we cannot piggy-back on Agda's support for higher-order functions and pattern

matching as liberally as we have done for the interpreters illustrated so far. For this reason, the interpreters illustrated throughout the rest of this paper utilize the less informative (but pragmatic) style of encoding monads, instead of the categorical style.

Our use of strong monads is similar to that of Moggi [1991]: monadic strength (our `_^_` operator) "mediates" between the current computation and values from the context. Moggi [1991] makes use of monadic strength in the framework of the computational λ-calculus, where `lets` use monadic strength to mediate *all* values from the context. Whereas Moggi [1991] defines the computational λ-calculus by a denotational translation into a categorical semantics, we are defining interpreters in Agda directly. Our use of monadic strength is therefore explicit, and is used only when necessary.

4 A UNIFORM APPROACH TO NAME BINDING: SCOPE GRAPHS

The previous section analyzed the problem of writing intrinsically-typed definitional interpreters for mutable state, which is one key characteristic of imperative languages. Many imperative languages share another key characteristic: they have constructs that use non-lexical binding. The de Bruijn indices we saw in STLC+Ref in the previous section describe lexical scoping quite naturally: a de Bruijn index essentially tells us how far to traverse along a static resolution path of lexically nested scopes in order to retrieve the value of a particular function-bound variable at run-time. But what is the de Bruijn index counterpart for non-lexical binding?

Recent work has proposed *scope graphs* [Néron et al. 2015; van Antwerpen et al. 2016] as a uniform approach to name binding for static semantics. Paths through scope graphs are essentially generalized de Bruijn indices that support both lexical and non-lexical binding. By structuring memory into run-time *frames* [Bach Poulsen et al. 2016], so that each frame is typed by a scope in the scope graph, static resolution paths can be used to resolve references at run-time, similarly to how de Bruijn indices were used to resolve references in STLC+Ref in the previous section. This section briefly summarizes the main ideas behind the scopes-and-frames approach, and subsequently introduces a novel intrinsically-typed encoding of scope graphs, and a library for structuring intrinsically-typed definitional interpreters using scope graphs and frames.

4.1 Scope Graphs

Scope graphs [Néron et al. 2015] provide a uniform approach to describing how references resolve to declarations. A *scope* is an abstraction over a set of abstract syntax tree (AST) nodes that behave uniformly with respect to name resolution. Figure 2 illustrates an example STLC expression; its AST; and its scope graph (or, in this case, *scope tree*[9]), which illustrates how a reference resolves via a *resolution path* (the dashed line in the scope graph on the right).

Each λ in the program in Figure 2 gives rise to a new lexical scope, so that there are four scopes in the example expression in Figure 2: one "top-level" scope (0) with no declared variables or references; and one scope for each λ term. The shaded regions in the expression and abstract syntax tree illustrate how each scope corresponds to the AST of the program. There are three kinds of edges in the scope graph on the right:

- A *declaration edge* connects a scope s (depicted as a circle node in the graph) to a declaration d (depicted as a square node in the graph) to indicate that d is declared in s. For example, the function that binds x induces an edge from scope 1 to the named declaration x.
- A *scope edge* connects a scope s_1 to a scope s_2 to indicate that the declarations of s_2 are reachable from s_1. For example, the edge from scope 2 to scope 1 makes the x declaration reachable from scope 2.

[9]Lexical scopes follow a tree-like structure. However, other kinds of binding, such as module or package imports, may contain cycles, hence we generally talk about scope *graphs*.

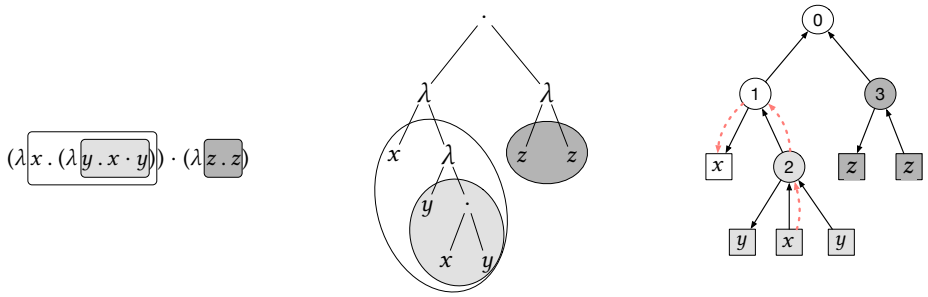

Fig. 2. An example STLC expressions (left); its AST (middle); and its scope graph (right)

- A *reference edge* connects a reference r (depicted as a square node in the graph) to a scope s to indicate that r is to be resolved starting from s. For example, resolution of the reference x should start in scope 2.

A scope graph is *resolved* by constructing a *resolution path* from each reference in the graph to a declaration of the same name. The dashed arrows decorating the scope graph in Figure 2 illustrate the path by which the reference x in the innermost λ resolves to the declared x in the outer λ. (These arrows are not part of the scope graph itself.) In general, a resolution path consists of:

- a reference source that has a reference edge to a scope in the graph (e.g., the reference x);
- a (possibly-empty) sequence of scope edge identifiers that describe which scope edge to traverse for each step of the path (e.g., the single scope edge between scope (2) and (1)); and
- a declaration identifier that describes a declaration that is connected to the destination scope of the path (e.g., the declaration x).

Resolution paths in scope graphs generalize de Bruijn indices in the following sense: a de Bruijn index corresponds to the length of a resolution path in a graph where each scope has exactly one declaration and exactly one scope edge that leads to a lexical parent scope. In contrast to the de Bruijn model, scope graphs support scopes with *multiple* declarations and *multiple* scope edges, so that lexical binding is a special case.

To illustrate how scope graphs can be used to model languages with more sophisticated notions of binding than de Bruijn indices, consider the Java program (left) in Figure 3 and its scope graph (right). The figure shows a root scope (0) and a scope for each class (1 and 2), as well as a scope for the body of the method `m` (scope 3). Inheritance is modeled by the import edge (labeled `I`) from the class scope (2) for `B` to the class scope (1) for `A`. This scope edge does not represent lexical scoping, since the `B` and `A` classes are not lexically nested. Like lexical scoping, the import edge makes all the declarations in the class scope (1) for `A` reachable from the class scope (2) for `B`. For example, the `x` reference in the method body scope for `m` (3) resolves to the `x` declaration in the `A` class. The method body scope (3) in Figure 3 also illustrates how each scope may have multiple declarations. Although not shown in this example, scope graphs can also have cycles, e.g., when two Java classes reference fields from each other.

The *resolution calculus* defined by Néron et al. [2015] and refined by van Antwerpen et al. [2016] defines how to resolve each reference in a scope graph to a corresponding declaration. In the rest of this paper, we assume that references in an AST have been replaced by their corresponding resolution paths, just like the earlier sections assumed that variables were resolved and replaced by de Bruijn indices. (References do not appear in our representation of scope graphs, described below, because they are not needed once resolution is complete.) We also assume that each declaration has an associated object-language type characterizing the values that the declaration binds.

```
class A { int x; /* ... */ }
class B extends A {
  public void m(int i, A a) {
    x = i;
    /* ... */
  }
}
```

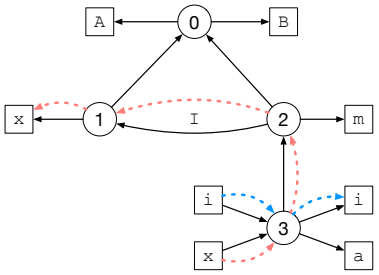

Fig. 3. An example Java program with non-lexical scoping (left) and its scope graph (right)

In the rest of this section, we present a small Agda library of scope graphs and *frames*, the run-time counterpart of scopes. We show the key definitions of this library, and illustrate how to use it to define a definitional interpreter for STLC using scopes and frames.

4.2 Intrinsically-Typed Syntax using Scope Graphs

We show how to define intrinsically-typed syntax using scope graphs.

Scope graphs. In what follows, we assume that scope graphs are finite, and that a given graph contains a set of k distinct scopes identified by the members of a finite set of k elements, where k is a parameter that must be instantiated for each particular graph. Thus we can define the following type which, for brevity, we call Scope, but which represents a scope identifier:

```
Scope = Fin k
```

We represent a scope graph as a total function from scope identifiers to *scope shapes*, where a scope shape defines the set of declarations (represented by a list of object language types, List *Ty*) and a list of scope edges (represented by a list of scope identifiers that are also in the graph, List Scope):

```
Graph = Scope → (List Ty × List Scope)
```

Here, *Ty* is a module parameter of the scope graph module. The parameter must be instantiated for a particular object language when the module is loaded. Note that the scope shape does not record the scope's references; each reference in an object program is replaced by its corresponding resolution path. Note also that this encoding allows scopes to refer to each other to define cyclic graphs, which is an important requirement; we discuss this point in more detail below.

The shape of a particular scope in the scope graph can be observed by looking up scope identifiers in the scope graph. For convenience, we define the following projection functions:

```
declsOf : Scope → List Ty      ; declsOf s = proj₁ (g s)
edgesOf : Scope → List Scope   ; edgesOf s = proj₂ (g s)
```

These projection functions and the definitions in the rest of this section are implicitly parameterized by an object-language scope graph g : Graph.

Resolution paths. A resolution path (_↦_) in g is a witness that we can traverse a sequence of scope edges (_⟶_) in g to arrive at a declaration of type t:

```
data _⟶_ : Scope → Scope → Set where
  []  : ∀ {s} → s ⟶ s
  _::_ : ∀ {s s′ s′′} → s′ ∈ edgesOf s → s′ ⟶ s′′ → s ⟶ s′′

data _↦_ (s : Scope) (t : Ty) : Set where
  path : ∀{s′} → s ⟶ s′ → t ∈ declsOf s′ → s ↦ t
```

Intrinsically-Typed Syntax for STLC. We can now define the intrinsically-typed syntax for STLC, using the same object language types as in Section 2, i.e.:

```
data Ty : Set where
  unit : Ty
  _⇒_  : (t u : Ty) → Ty
```

Expressions for STLC are now indexed by scopes (and implicitly by a scope graph g) instead of a type environment:

```
data Expr (s : Scope) : Ty → Set where
  unit : Expr s unit
  var  : ∀ {t} → (s ↦ t) → Expr s t
  lam  : ∀ {s′ t u} → ⦃ shape : g s′ ≡ ( [ t ] , [ s ] ) ⦄ → Expr s′ u → Expr s (t ⇒ u)
  _·_  : ∀ {t u} → Expr s (t ⇒ u) → Expr s t → Expr s u
```

The Scope index indicates what scope each AST node is associated with in the scope graph; all new scopes (beyond the root scope) are introduced by lam expressions. The _≡_ type used in the lam constructor is Agda equality, here used to observe the structure of a scope in the graph. Thus a function body is scoped by an s' in scope graph g which has a single declaration typed by t and a single scope edge to the lexical context scope s.

We encode scope shape assertions as instance arguments to reduce noise in interpreters: pattern matching on a lam makes the scope shape fact available in the context of the match, so that it can be automatically used as an instance argument by any function call that expects an instance argument of the same type. This use of instance arguments saves having to explicitly propagate instance facts later when we define our interpreter for STLC.

Compared with the intrinsically-typed syntax of STLC from Section 2, the abstract syntax definition using scope graphs relies on scope graph lookups to observe the structure of scopes. This makes the definition of lam a little more verbose. The strength of scope graphs is that they provide a uniform model of name binding and memory that go beyond lexical scoping, and that they can be used to define a library of language-parametric memory operations, as we shortly show.

A Note on Scope Representation. The representation of scope graphs above relies on a representation of scopes and scope graphs as a lookup table: a scope is identified by a finite number (Fin); and we observe the structure of a scope by doing a lookup in the Graph. In contrast, lexical type contexts are typically represented using a nested data type (e.g., a list), as we illustrated in Section 2.2. It is tempting to define scope graphs similarly, i.e.:

```
data DataScope : Set where sc : List Ty → List DataScope → DataScope
```

This data type is fine for representing lexical scoping, but it is problematic for representing non-lexical binding, since: (1) the natural notion of scope equality is structural equality (in contrast to comparing scope identifier Fins); and (2) the type is inductive, so it does not support scope *graphs*, only scope *trees*. The first point is problematic because it is useful to be able to distinguish scopes *nominally* rather than *structurally*. For example, the Middleweight Java syntax in Section 5 of this paper identifies classes by their scope. The second problem can be addressed by generalizing DataScope to a coinductive type, but such a generalization only aggravates the first point, due to the non-trivial nature of comparing infinite data (see, e.g., [Pierce 2002, Chapter 20]).

4.3 Frames and Heaps

Following Bach Poulsen et al. [2016], we can use *frames* to structure run-time memory into heaps that follow the structure of scope graphs. A well-typed frame is typed by a scope when they are in one-to-one correspondence, as illustrated by the example in Figure 4: for every scope declaration of type `t`, there is a frame *slot* of type `Val t`; and for every scope edge to a scope s, there is a frame *link* to a frame of type s.

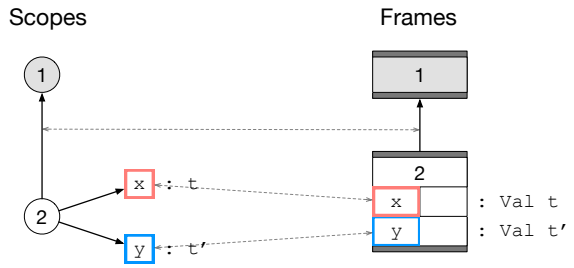

Fig. 4. An illustration of the correspondence between scopes and frames

The scopes-and-frames correspondence summarized in Figure 4 is a type discipline for structuring memory: frames are the basic memory blocks, and each frame is typed by a scope in the scope graph. At run-time, frames reside in a *heap* where each location in the heap is a frame. We say that a heap is well-typed when each location (frame) in the heap is well-typed (by a scope). Following Bach Poulsen et al. [2016], we can define memory operations that preserve this correspondence. Our contribution is to show how to define the scopes-and-frames approach in an intrinsically-typed manner that is parametric in object-language notions of types and values. In the rest of this section, we implement these memory operations as an Agda library, and illustrate how to write a definitional interpreter using this library.

Well-Typed Frames and Heaps. A *well-typed heap* is a heap where each frame is typed by a scope. Thus a *heap type* that describes the shape of a heap is given by a list of scopes:

```
HeapTy = List Scope
```

Heap types correspond to the store types we used to define well-typed stores in Section 3. The `Frame` type denotes a well-typed pointer into a heap that is typed by a heap type Σ:

```
Frame : (s : Scope) → (Σ : HeapTy) → Set
Frame s Σ = s ∈ Σ
```

A `HeapFrame` is typed by a scope when it comprises a set of well-typed `Slots` (values whose types are in one-to-one correspondence with the scope declaration types) and well-typed `Links` (frame pointers to frames whose scopes are in one-to-one correspondence with the scope edges):

```
Slots : (ds : List Ty) → (Σ : HeapTy) → Set
Slots ds Σ = All (λ t → Val t Σ) ds

Links : (es : List Scope) → (Σ : HeapTy) → Set
Links es Σ = All (λ s → Frame s Σ) es

HeapFrame : (s : Scope) → (Σ : HeapTy) → Set
HeapFrame s Σ = Slots (declsOf s) Σ × Links (edgesOf s) Σ
```

Here, the *Val* type in Slots is a parameter of the scopes-and-frames module. Similarly to how we defined stores in Section 3.1, we define a well-typed heap as being in one-to-one correspondence with a heap type Σ:

```
Heap : (Σ : HeapTy) → Set
Heap Σ = All (λ s → HeapFrame s Σ) Σ
```

The Heap type guarantees that each frame in memory is described by a scope in the scope graph.

Well-Typed Memory Operations. We can now define memory operations similar to those of Bach Poulsen et al. [2016]. In contrast to that work, where the operations were proven correct via a substantive proof effort in Coq, the operations we define are intrinsically-typed, which delegates the work of verifying them to Agda's type checker.

Since each frame is typed by a scope, we can define the following well-typed lookup functions:

```
getFrame : ∀ {s s′ Σ} → (s ⟶ s′) → Frame s Σ → Heap Σ → Frame s′ Σ
getSlot  : ∀ {s t Σ} → t ∈ declsOf s → Frame s Σ → Heap Σ → Val t Σ
```

The type of getFrame says that we can traverse a resolution path in a well-typed heap to lookup a frame whose scope matches the destination of the path. The type of getSlot says that we can use a scope declaration for looking up a frame slot value whose type matches the declaration. Using these two functions, we can define a short-hand function for looking up resolution paths:

```
getVal : ∀ {s t Σ} → (s ↦ t) → Frame s Σ → Heap Σ → Val t Σ
```

We also need the following function for initializing and extending the heap with a new frame. The function takes as input a scope s to be initialized; a scope shape description (as an instance argument which is automatically resolved where possible); and slots and links that are in one-to-one correspondence with the shape of the scope that the resulting frame initializes:

```
initFrame : (s : Scope) → ∀ {Σ ds es}⦃ shape : g s ≡ (ds , es) ⦄ →
            Slots ds Σ → Links es Σ → Heap Σ → Frame s (Σ ::ʳ s) × Heap (Σ ::ʳ s)
```

The function updates the heap with a new frame which is typed by the input scope identifier s, and returns a pointer to the freshly-allocated frame. $\Sigma ::^r s$ denotes the append of the scope identifier s to the end of the heap type Σ. Finally, we need a function that, given a frame pointer, a pointer to a declaration, and a value of the same type as the declaration, updates a frame slot with the value:

```
setSlot : ∀ {s t Σ} → t ∈ declsOf s → Val t Σ → Frame s Σ → Heap Σ → Heap Σ
```

4.4 Definitional Interpreter for STLC using Scopes and Frames

We now have the necessary ingredients for writing a definitional interpreter for STLC using scopes-and-frames: the intrinsically-typed syntax for STLC using scope graphs in Section 4.2 and the library for scopes-and-frames. All we need to do is instantiate the parameters of the library with our notion of object language type and object language value. The object language types for STLC are the usual ones. The notion of value for STLC is also the usual one, but instead of storing a *lexical environment*, a closure stores a *lexical frame* and uses an instance argument to store what the shape of the scope of the function is (a single declaration for the function argument, and a single scope edge to the lexical context scope); this scope shape fact is used to instantiate the function's frame:

```
data Val : Ty → (Σ : HeapTy) → Set where
  unit : ∀ {Σ} → Val unit Σ
  ⟨_,_⟩ : ∀ {Σ s s′ t u}⦃ shape : g s′ ≡ ([ t ] , [ s ]) ⦄ →
          Expr s′ u → Frame s Σ → Val (t ⇒ u) Σ
```

Closures thus store lexical context frames similarly to how the closure for STLC in earlier sections store their lexical environment.

To define our interpreter, we introduce a monad that threads heaps and propagates timeouts if the interpreter runs out of fuel. Instead of a type environment, the monad is indexed by a scope s which describes the "current frame":

```
M : (s : Scope) → (P : HeapTy → Set) → (Σ : HeapTy) → Set
M s P Σ = Frame s Σ → Heap Σ → Maybe (∃ λ Σ′ → (Heap Σ′ × P Σ′ × Σ ⊑ Σ′))
```

The bind and return operations for this monad are similar to the monads we have seen so far. Like in Section 3.4, we define a monadic strength operation for carrying values across monadic binds:

```
_^_ : ∀ {Σ Γ P Q} → ⦃ w : Weakenable Q ⦄ → M Γ P Σ → Q Σ → M Γ (P ⊗ Q) Σ
```

We also define operations for accessing the current frame (getFrame) and passing in a different frame (usingFrame), similarly to the getEnv and usingEnv operations from Section 3. Finally, we lift the frame operations defined in the previous section to monadic operations. The only two we need for STLC are init for initializing call-frames, and getv for resolving a resolution path relative to an offset frame described by s:

```
init : ∀ {Σ ds es s′}(s : Scope)⦃ shape : g s ≡ (ds , es) ⦄ →
       Slots ds Σ → Links es Σ → M s′ (Frame s) Σ
getv : ∀ {s t Σ} → (s ↦ t) → M s (Val t) Σ
```

The evaluation function for STLC is:

```
eval : ℕ → ∀ {s t Σ} → Expr s t → M s (Val t) Σ
eval zero    _          = timeout
eval (suc k) unit       = return unit
eval (suc k) (var x)    = getv x
eval (suc k) (lam e)    = getFrame >>= λ f → return ⟨ e , f ⟩
eval (suc k) (e₁ · e₂)  = eval k e₁ >>= λ{ ⟨ e , f ⟩ →
                          (eval k e₂ ^ f) >>= λ{ (v , f) →
                          init (sₑ e) (v :: []) (f :: []) >>= λ f′ →
                          usingFrame f′ (eval k e) }}

sₑ : ∀ {s t} → Expr s t → Scope
sₑ {s} _ = s
```

The signature of the evaluation function says that the "current frame" of the monad is always typed by the scope of the current expression. This guarantees that static resolution paths correspond to paths through frames in memory, so that the getv application in the var case is well-typed. The lam case makes use of instance arguments to automatically store the shape of a scope. The case for function application initializes a new call frame which stores the argument value (v) and is linked to the lexical context frame (f) of the closure, before evaluating the body of a function in the context of the call frame. Calling a function instantiates a frame typed by the scope of the function body. In this sense, the scope of a function is really the type of the familiar notion of a call frame.

We have presented a small library for programming with scopes and frames in intrinsically-typed interpreters. Using scopes-and-frames, definitional interpreters automatically inherit a safety principle for memory, so that object languages do not need to define their own notion of well-typed memory or auxiliary entities (such as objects, stacks, etc.). It is straightforward to extend this semantics to STLC+Ref, e.g., by using a disconnected single-declaration scope to represent reference cells. In the next section we illustrate how scopes-and-frames are well-suited for defining imperative object languages by describing a formalization of Middleweight Java using scopes-and-frames.

5 A DEFINITIONAL INTERPRETER FOR MIDDLEWEIGHT JAVA

In the scopes-and-frames approach, scopes and scope graphs are a type discipline for memory frames and heaps. We illustrate how this type discipline is well-suited for defining imperative languages where the syntax and scope structure of a program defines how memory is structured at run time by describing an intrinsically-typed definitional interpreter for a subset of Middleweight Java (MJ) [Bierman et al. 2003]. In Section 5.8 we discuss how the MJ interpreter using scopes-and-frames compares with previous Java formalizations, and compare with an alternative intrinsically-typed interpreter for MJ without scopes-and-frames, which we have also developed.

5.1 Overview of MJ

MJ aims to be an *imperative core calculus* for Java [Bierman et al. 2003]. It can itself be regarded as an extension of Featherweight Java (FJ) [Igarashi et al. 2001], which includes: class definitions, object creation, readable fields and callable methods, method overrides, sub-typing and casting. MJ extends FJ with imperative features such as field assignment, mutable lexical environments, null pointers, and block scopes. From the features that MJ covers, we omit a few for brevity, namely: constructors, up-casts, and object equality. There are some more minor differences that simplify our presentation and discussion without loss of generality of the approaches presented, e.g., we do not differentiate between expressions that can be promoted to statements and those that cannot. These discrepancies are discussed in detail in the artifact accompanying this paper [Bach Poulsen et al. 2017]. Throughout this section we use "MJ" to refer to the subset of the original language definition that we consider.

An MJ program consists of a set of class definitions and a main method. Classes are defined by an optional named parent reference, a set of fields, and a set of methods; all class-members are explicitly typed. Method declarations are not ordered and methods may invoke each other. Class definitions may be mutually recursive, but the *class hierarchy* should be well-founded; i.e. two classes may not both inherit (transitively) from each other. Sub-typing on classes is defined as the reflexive-transitive closure of the child-parent relation between class declarations. All classes inherit from the distinguished class Object which is at the base of the inheritance relation.

MJ is a useful case study because of its mix of lexical and static binding, its imperative character, and because of the encoding challenges presented by features such as method overrides and dynamic dispatch. The remainder of this section presents a large part of the intrinsically-typed syntax for MJ using scope graphs as well as the semantics for expressions and statements, with a focus on the challenges that MJ presents compared to STLC+Ref.

5.2 Instantiating the Agda Scope Graph Library

We will use the scope graph library to model *all name binding* in MJ, both lexical and non-lexical. The names that appear in MJ are either variables, classes, fields, or methods. We construct an Agda type Ty^t of *tagged types* to represent all the types that declarations in an MJ program can have:

```
data Ty^t : Set where
  v^t : Ty → Ty^t
  m^t : List Ty → Ty → Ty^t
  c^t : Scope → Scope → Ty^t
```

```
data Ty : Set where
  void : Ty
  int  : Ty
  ref  : Scope → Ty
```

The three constructors of tagged types Ty^t correspond to the three kinds of names in MJ. Variables and fields are typed using v^t and can have any of the three *value types* of MJ, represented by Ty: either it is one of the primitive types (void or int), or the type ref *s* of a reference to an object. Methods are typed by the list of their argument types together with their return type using the

constructor m^t. Classes are typed by the identifiers of their surrounding scope and the scope that they introduce using c^t.

We define a type for tagged values Val^t as the well-typed inhabitants of tagged types Ty^t:

```
data Val^t : Ty^t → HeapTy → Set where
  v^t : ∀ {t Σ} → Val t Σ → Val^t (v^t t) Σ
  m^t : ∀ {s ts rt Σ} → (self : Frame s Σ) → Meth s ts rt → Val^t (m^t ts rt) Σ
  c^t : ∀ {s^r s Σ} → Class s^r s → (root : Frame s^r Σ) → Val^t (c^t s^r s) Σ
```

The type Val, appearing here in the constructor v^t, represents MJ values and is indeed indexed by a value type Ty. Class values consist of a pointer to the root frame and a well-typed class definition Class in it. Method values are given by a well-typed method body (Meth) and a pointer to the frame representing the object to which it is bound (i.e., the "self" reference). These structures are treated in more detail in Section 5.4 about object representation and Section 5.5 about sub-typing.

Ty^t defines the type of declarations in the scope graph, and Val^t defines the type of slots in frames. We instantiate the Agda library for scopes-and-frames by passing these two types as parameters.

5.3 Program Scopes are Class Tables

An MJ program consists of a set of classes and a main method. For a given MJ program, we let the root scope (or program scope) contain a declaration for each defined class. Thus the root scope is the type of a run-time frame that contains the class definitions, corresponding to the usual notion of a class table [Igarashi et al. 2001; Bierman et al. 2003]. The following type defines MJ programs whose root scope is given by a scope s^r, and whose main methods return a value of type t:

```
data Program (s^r : Scope)(t : Ty) : Set where
  program : ∀ {cs s'}{⦃ shape : g s^r ≡ (cs , []) ⦄ →
            (classdefs : All (λ{ (c^t s^r s) → Class s^r s ;
                                 _           → ⊥ }) cs) →
            (main : Stmts s^r t s') → Program s^r t
```

The shape of the root scope s^r is constrained to have a list cs of declarations. The second parameter uses a pattern matching λ to constrain those declarations to have class type c^t (and not, say, a value type v^t) and associates them with their definitions.

5.4 Object Representation

Each class has a scope that defines the type of a frame which has a slot for each class member. The following Class type defines the set of all MJ classes with a parent class:

```
data Class (s^r s : Scope) : Set where
  class : ∀ {ms s^p}{oms : List (List Ty × Ty)} ⦃ shape : g s ≡ (ms , s^r :: s^p :: []) ⦄ →
          (parent : s ↦ (c^t s^r s^p)) →
          (members : All (λ{ (m^t ts rt) → Meth s ts rt ;
                             (v^t _)     → ⊤ ;
                             _           → ⊥ }) ms) →
          (overrides : All (λ{ (ts , rt) → (s ↦ (m^t ts rt)) × Meth s ts rt }) oms) →
          Class s^r s
```

A Class is parameterized by two scopes: the root scope s^r of the program that is the lexical context for the class, and the scope s that the class defines.

The $s \mapsto (c^t\ s^r\ s^p)$ parameter for class is a path from the defined class to the declaration for the parent class scoped by s^p. The *members* parameter defines the non-overriding class members. Class members are typed by *ms*, which may be either method- or value-types (for fields), but not class-types; once again, this restriction is enforced by a pattern matching λ. For each method declaration in the class scope, there is a well-typed method definition in the class definition. Meth *s ts rt* is the type of method bodies which have a "self" reference to the class scope *s*, take a list of type parameters *ts*, and have return type *rt*. The *overrides* parameter provides a path to the original implementation, as well as a well-typed new implementation, for all the method signatures that the class claims to override (*oms*).

```
class A { int x; /* ... */ }
class B extends A {
  public void m(int i, A a) {
    x = i;
    /* ... */
  }
}
```

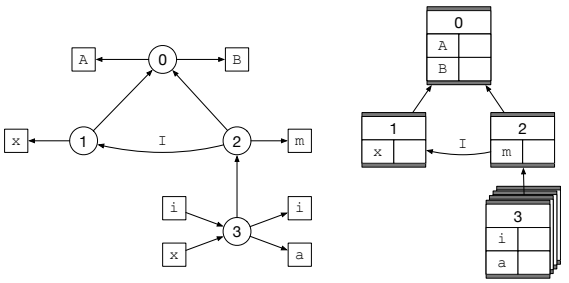

Fig. 5. A Java program with inheritance (left); and its scope graph and hierarchical frame structure (right)

The class definition above defines a scope shape where all class members reside in the same scope. This scope defines the type of MJ objects in our interpreter. Objects with inheritance are represented by hierarchies of frames: one for every link in the inheritance chain of a class. As an example, involving two simple classes, Figure 5 repeats the code and scope graph from Figure 3 and shows the corresponding hierarchy of frames on the right. The top frame (0) in the figure represents the program frame (or the *class table*); the two frames 1 and 2 represent a *single* object instance for the B class. Frame 1 contains all class members for A while frame 2 contains all class members for B. There may be multiple frames labeled with scope 3, corresponding to possible activations of method m.

The chosen object representation is not very efficient: it could be improved by using a copy-down transformation to "flatten" scopes so that objects are represented as a single frame, or by factoring method members into a separate scope (and frame) so that class method frames could be initialized once and reused between object instances.

5.5 Dealing With Sub-Typing

Sub-typing in MJ is based on class inheritance and only defined between reference types. Because the direct inheritance relation between classes is witnessed in the scope graph by an edge between class scopes, and inheritance is defined as the reflexive-transitive closure of this relation, the notion of inheritance coincides with our definition of a path:

```
_<:_ : Scope → Scope → Set
_<:_ = _⟶_
```

Using this notion of sub-typing, we can define a type for MJ values:

```
data Val : Ty → HeapTy → Set where
  ref  : ∀ {s s′ Σ} → s <: s′ → Frame s Σ → Val (ref s′) Σ
  null : ∀ {s Σ} → Val (ref s) Σ
```

The ref constructor includes a sub-type fact that witnesses how the reference has been cast. Consequently, casting reduces to concatenation of paths, as shown by upcastRef:[10]

```
upcastRef : ∀ {s s′ Σ} → s <: s′ → Val (ref s) Σ → Val (ref s′) Σ
upcastRef p (ref p′ f) = ref (concat_p p′ p) f
upcastRef p null = null
```

The sub-typing fact carried by references bridges the gap between the static and the dynamic type of a reference at run-time. In Section 5.7 we will see how to cross this bridge as we interpret member access on references that have been up-cast.

5.6 Object Initialization and Dynamic Dispatch

The purpose of distinguishing overriding methods from other methods in class definitions is to support dynamic dispatch. Objects are initialized by constructing a frame for each class in the inheritance chain for the initialized class. Frames for super classes are constructed first. Child classes override methods by *overwriting* the method slots in the frames of super classes. This way, whenever we dispatch on methods in a super-class, any overridden methods will be used in place of the methods that the super class is statically associated with.

For example, an object instance of class `B` from Figure 6 is represented as two frames, where the frame for the class members of `A` contains the implementation of `B`'s `m1()` method in the slot corresponding to the class member `m1()`. Evaluating **`new`** `B().test()` initializes the frames for `B`, and overrides `m1`, so that the overriding method is the one that is going to be called. For more details on object initialization and overriding, see the artifact accompanying this paper [Bach Poulsen et al. 2017].

```
class A {
  public void m1() {
    System.out.println("I'm sorry, Dave");
  }
  public void m2() {
    return this.m1();
  }
}
```

```
class B extends A {
  public void m1() {
    System.out.println("42");
  }
  public void test() {
    return m2();
  }
}
```

Fig. 6. Two Java classes with dynamic dispatch; running **`new`** `B().test()` prints the string "42"

5.7 Interpretation

Now we can describe well-scoped and well-typed MJ expressions and their interpretation, focusing on the object-oriented features and the handling of sub-typing. Statements (and sequences thereof) are implemented in a similar manner; the full details are available in the accompanying artifact [Bach Poulsen et al. 2017].

We assume that MJ programs have been analyzed and checked so that every expression position where sub-typing might occur is annotated with a sub-typing proof that witnesses how the actual type of the expression can be cast to match the expected type of the expression. The Expr type (only select constructs are shown here) is the type of ordinary expressions whose sub-expressions may have been up-cast using the upcast constructor:

[10] An alternative approach is to drop the sub-typing witness from reference values and implement upcastRef eagerly by traversing the heap and returning the parent frame. This would be sufficiently expressive for MJ, but would lose information about the dynamic type of a reference, thus precluding adding features such as down-casts.

```
data Expr (s : Scope) : Ty → Set where
  var     : ∀ {t} → (s ↦ v^t t) → Expr s t
  new     : ∀ {s' s''} → s ↦ c^t s' s'' → Expr s (ref s')
  get     : ∀ {s' t} → Expr s (ref s') → (s' ↦ v^t t) → Expr s t
  call    : ∀ {s' ts t} → Expr s (ref s') → (s' ↦ (m^t ts t)) →
            All (Expr s) ts → Expr s t
  upcast  : ∀ {s' s''} → s' <: s'' → Expr s (ref s') → Expr s (ref s'')
```

We define the following evaluation function:

```
eval : ℕ → ∀ {s t Σ} → Expr s t → M s (Val t) Σ
```

The monad M is similar to the one defined in the previous section: it threads heaps and deals with possible-timeouts. Additionally, because MJ has null values, the monad also propagates null-pointer exceptions. All cases of the interpreter rely on the memory operations defined in the Agda library for scopes-and-frames. To illustrate, here are some of the cases for expression evaluation:

```
eval (suc k) (new x)          = getv x »= λ{ (c^t class f) →
                                usingFrame f (init-obj k class) »= λ{ f' →
                                return (ref [] f') }}
eval (suc k) (get e p)        = eval k e »= λ{ null → raise ; (ref p' f) →
                                usingFrame f (getv (prepend p' p)) »= λ{ (v^t v) →
                                return v }}
eval (suc k) (call e p args)  = eval k e »= λ { null → raise ; (ref p' f) →
                                usingFrame f (getv (prepend p' p)) »= λ{ (m^t f' (meth s b)) →
                                (eval-args k args ^ f') »= λ{ (slots , f') →
                                init s slots (f' :: []) »= λ f'' →
                                usingFrame f'' (eval-body k b) }}}
eval (suc k) (upcast p e)     = eval k e »= λ v → return (upcastRef p v)
```

Here, getv is the monadic version of getVal that we defined in Section 4.3 and init-obj initializes a new default object instance, where all method slots have been properly initialized, and all fields have been initialized with a *default value* (0 for int fields; null for ref typed fields). Note also how member access occurs on references by prepending the sub-typing path p' to bridge the gap between the dynamic and static type of a reference.

Most cases are straightforward; the most complicated case is the one for method calls. Reading from top to bottom, call first evaluates the receiver expression to an object value (or raises a null-pointer exception if the resulting value is null); then uses the resulting frame to look up the method of the object using the path recorded in the call expression. The resulting method value records a frame pointer f' to the "self" of the object in which the method resides. Next, the arguments to the method are evaluated to produce a set of well-typed slot values. The "self" frame f' is used to initialize the static link of the call-frame, and the slots of the frame are populated with the well-typed argument slot values. Finally, the body of the method is evaluated in the context of the call frame.

5.8 What Does it Buy Us?

We have described a definitional interpreter for MJ using the Agda library for scopes-and-frames. We discuss the process of writing the interpreter and compare the resulting specification with other specifications for subsets of Java, written by ourselves and others.

First and foremost we repeat our observation from the introduction that working with well-typed syntax turns safety into a guiding contract that is interactively enforced during the process of engineering interpreters. Additionally, we stress the benefit of producing a specification that is *executable*. Just because an interpreter is free of type errors does not guarantee that it implements the right semantics. The ability to execute example programs is useful for such validation.

In parallel with the MJ interpreter described in this section, we also built "Scopeless MJ", a definitional interpreter in Agda for MJ that uses tailor-made intrinsically-typed data structures for class tables and environments, rather than the scopes-and-frames library. A significant amount of the development effort for Scopeless MJ went into inventing intrinsically-typed auxiliary data structures for name binding aspects of MJ. Each data structure needs the right balance between being practical for representing invariants and practical to program with. Discovering such data structures requires creativity and insight. In our experience, the language-parametric scopes-and-frames library strikes this balance in a language-parametric way that is useful for writing interpreters. Using scopes-and-frames we get for free both a type discipline for the memory of the object language and a set of sound-by-construction memory operations. A potential drawback of using a general-purpose name binding model is that it may not always be obvious how to encode language features. However, previous work [Néron et al. 2015] has shown that scope graphs already scale to a wide range of different language features. As ongoing work, we are exploring how scope graphs can be used to model even more features, such as polymorphism and dependent types.

Figure 7 compares our two MJ formalizations with each other and with other Java formalizations. For our two semantics, we count lines of Agda code for the interpreter itself, but also for the monad and auxiliary functions. If we consider only the size of the evaluation functions for expressions and statements, our interpreter is 82 LOC. This number is roughly comparable to the 73 LOC for the relatively densely typeset reduction rules in the original MJ paper [Bierman et al. 2003, Fig. 2 and 3]. This is only a rough comparison, since there are several stylistic differences between our sound-by-construction monadic definitional interpreter and the abstract machine semantics of Bierman et al. [2003], and since we model a subset of MJ (excluding up-casts and explicit constructors). Jinja [Klein and Nipkow 2005] is a subset of Java that is larger than MJ, formalized in Isabelle. Klein and Nipkow [2005] give both a small-step and a big-step semantics for Jinja, prove them equivalent, and prove type safety for the small-step semantics. Featherweight Java (FJ) is a subset of Java that is smaller than MJ. FJ omits imperative features such as mutable fields and environments. The original definition of FJ [Igarashi et al. 2001] is on pen-and-paper, and includes high level proofs of progress and preservation. Figure 7 summarizes the size of two Coq mechanizations of FJ: Delaware

	MJ	Scopeless MJ	Original MJ	Jinja	FJ_{PLoTs}	FJ_{AI}
Syntax	179[a]	316	98	368	325	233
Extrinsic Type Rules	0	0	61 rules	1207	234	111
Semantics	215	249	73[b]	432[c]	198	159
Type safety proof	0	0	15 pages	711[d]	2449	1978
Total	404	565	-	2718	3206	2481

Fig. 7. Approximate sizes (in LOC if unit is omitted) for our MJ and related developments

[a]This includes 76 LOC for the scopes-and-frames library
[b]This includes 12 LOC for an "obvious" frame substitution function omitted in the paper
[c]Only counting the lines of the big-step semantics
[d]Type safety is proven for an equivalent small-step semantics

et al. [2011] (FJ_{PLoTs}) focus on extensibility of specifications and Mackay et al. [2012] (FJ_{AI}) include field assignment and immutability modifiers.

There are also formalizations for larger subsets of Java that take a different approach to validation. For example, K-Java [Bogdanas and Rosu 2015] is an executable specification for the static and dynamic semantics of Java 1.4, developed using test-driven development in the context of the K Framework [Rosu and Serbanuta 2010]. K-Java is not proven type sound, but is validated by 840 tests that check agreement with the Java language specification and official compiler. The purpose of our work is to show how to engineer definitional interpreters that are sound-by-construction, in a way that also supports testing. As summarized in Section 1.3, it remains to integrate the research in this paper with a parser and type checker that produces the intrinsically-typed syntax that our interpreters take as input.

6 RELATED WORK

Definitional Interpreters. Definitional interpreters date back to the earliest days of computer science, and there was already a large body of literature by the time Reynolds [1972] published his seminal study. Definitional interpreters have been somewhat overshadowed, particularly when proving type safety is a primary goal, by more syntactic approaches [Wright and Felleisen 1994; Milner et al. 1997; Pierce 2002]. However, recent years have seen a revival in interest in functional encodings of semantics. For example Danielsson [2012] shows how to prove type safety for definitional interpreters for languages with non-termination using the partiality monad [Capretta 2005], and a number of recent papers have advocated modeling execution in a total language by using fuel [Siek 2013; Owens et al. 2016; Amin and Rompf 2017].

The connection between intrinsically-typed syntax and definitional interpreters has also been made early and often. For example, early work on generalized algebraic datatypes [Xi et al. 2003; Bird and Meertens 1998; Augustsson and Carlsson 1999; Altenkirch and Reus 1999] often uses definitional interpreters as a motivational example. Popular approaches to modeling name binding with GADTs include higher-order abstract syntax [Pfenning and Elliott 1988] and de Bruijn indices [de Bruijn 1972]. Recently, Allais et al. [2017] presented a framework based on de Bruijn indices that factor out common scope and type structure in a way that facilitates the development of type sound internal DSLs. Their framework appears focused on languages without mutable state.

Work on intrinsically-typed definitional interpreters seems to have focused mainly on pushing the boundary of what object-language type systems can be expressed. For example, attempts to give intrinsically-typed syntax for type-theory itself, i.e., to use dependently-typed languages as both object and host, have motivated the exploration of many extensions to dependently typed languages (e.g. [Chapman 2009; Licata and Harper 2009; McBride 2010; Altenkirch and Kaposi 2016a,b]). The object languages studied are usually pure, and typically limited to using de Bruijn indices for binding [Brady and Hammond 2006; Benton et al. 2012]. Our work in this paper explores different dimensions of language complexity: non-trivial binding structures and mutation in imperative languages. Brady [2013b] shows how to use the combined features of dependent types and algebraic effects to implement a type-preserving interpreter for a small imperative language. The small language that Brady [2013b] considers supports mutable lexical environments where references cannot escape their local lexical context. It is not clear how Brady's interpreter would scale to deal with ML-style references without adopting similar techniques as described in this paper.

Intrinsic and Extrinsic Verification. There are two approaches to verifying that functions satisfy a given specification: extrinsic and intrinsic. Bertot and Castéran [2004, Chapter 9] describe these approaches as *weak* and *strong* specifications. In *extrinsic verification*, a function is defined with a *weak specification*, and *companion lemmas* are added to prove that the function satisfies the

specification. In *intrinsic verification*, the function is given a *strong specification* whose type directly guarantees that the function satisfies the specification.

Intrinsically-typed interpreters are one example of intrinsic verification where the property being verified is type safety. There are many other applications of intrinsic verification for programs besides definitional interpreters. For example, Affeldt and Sakaguchi [2014] use intrinsic encodings to verify C programs implementing TLS network package processing. Korkut et al. [2016] use intrinsic verification for a regular expression matcher. Harper and Stone [2000] give a type-theoretic account of Standard ML by using intrinsically-typed syntax, and defining a typed small-step reduction relation that is (manually) proven to satisfy progress and preservation.

In this paper we have focused on intrinsic verification by using a dependently-typed host language as the automatic proof procedure and underlying logic. There are several lines of work on automatic verification that take different approaches. Grewe et al. [2015] demonstrate how to translate language specifications into first-order logic and use off-the-shelf theorem provers for first-order logic to prove type safety automatically. It is also possible to automate type safety proof search in tactic-based theorem provers, like Coq [Chlipala 2013] or Isabelle/HOL [Nipkow and Klein 2014]. Cimini et al. [2016] classify an expressive subset of evaluation context-based reduction semantics that satisfy type safety, and propose a procedure for automatically constructing an extrinsic mechanized proof of safety in the Abella proof assistant [Baelde et al. 2014]. Lorenzen and Erdweg [2016] define a procedure for checking that language extensions and desugarings are type-preserving, and prove that their procedure is sound. The techniques and interpreters that we have presented are based on the well-established theory of dependent types. This makes our techniques transferable to a range of dependently-typed host languages.

Recent work by Krebbers et al. [2017] shows how to embed a domain-specific logic (such as concurrent separation logic) into Coq. Although that work applies primarily to embeddings of logic (as opposed to programming languages at large) it would be interesting to see if a similar approach could be used to define interpreters in terms of the categorical framework discussed in Section 3.6. As also discussed in Section 3.6, we did not pursue that direction here, since our purpose was to see how far we could get with writing intrinsically-typed interpreters in Agda without extensive use of reflection or meta-programming (like type classes or user-defined tactics or macros).

Swierstra et al. [2016] recently presented a novel approach to encoding and defining store types with automatic variable weakening using existential types and circular lazy programming. It is not clear how this approach can be integrated in more traditional dependently-typed host languages.

Dependently-Typed Monads. Swierstra [2009a] first showed how to type ML-style references intrinsically using Atkey-style monads [Atkey 2009] parameterized by both the initial and final store type. Swierstra [2009a] also showed how to weaken stores automatically when the initial and final store type are known statically. The monads we use are based on the monads over indexed sets due to McBride [2011], where the final type index is not assumed to be statically known. McBride [2011, Section 6] also illustrates how such monads can be defined as *free monads* that correspond to the interaction structures of Hancock and Setzer [2000].

Hoare and Dijkstra monads, as used in the context of systems for program verification such as Hoare type theory (HTT) [Nanevski et al. 2008], Ynot [Chlipala et al. 2009], and F* [Swamy et al. 2013; Ahman et al. 2017], provide an extrinsic approach to program verification: they are parameterized by pre- and post-conditions given as arbitrary propositions over the computation state, and any use of a monadic bind between two computations gives rise to proof obligations that the pre- and post-conditions for the two computations are consistent. Proof obligations about pre- and post-conditions are either proven by appealing to automation for general-purpose proof search (e.g., F* has integrated proof search facilities) or by manual proof work (e.g., using Coq's Program

facility [Sozeau 2008] following Swierstra [2009b]). Recent work on F* [Ahman et al. 2018] adds integrated support for verification properties over monotonic state, which could in theory be used to implement definitional interpreters with store type weakening. In contrast to previous work on Dijkstra and Hoare monads, the monads we use are *intrinsically-typed* "plain" monads over sets with more structure than the monads used in non-dependently-typed programming. Each kind of dependently-typed monad has its advantages and disadvantages: Hoare and Dijkstra monads can be used to state and verify general-purpose properties, but they also require general-purpose proof automation or manual proof work. Monads over indexed sets support automatic verification by using little more than a dependently-typed type checker, but they mainly support verification of intrinsic properties of monadic computations.

Semantic Specification Frameworks. There is a range of systems and frameworks for specifying programming language semantics and deriving interpreters from specifications. Examples include PLT Redex [Klein et al. 2012], the K Framework [Rosu and Serbanuta 2010], Maude [Clavel et al. 2007], DynSem [Vergu et al. 2015], MSOS [Mosses 1999; van Binsbergen et al. 2016], and Lem [Mulligan et al. 2014]. These systems generally support specification via either small-step or big-step inference rules, in contrast to the monadic definitional interpreters we use. Some of these systems support automatic dynamic verification via random test generation (PLT Redex) or model checking (K Framework and PLT Redex). The K Framework also has support for automated static verification of object language programs against functional specifications [Stefanescu et al. 2016]. Typed Racket (of which PLT Redex is an internal DSL) has recently [Racket Development Team 2017] added support for *refinement types*, a variant of dependent types. As far as we are aware, none of the systems address automatic static verification of type safety for object languages.

7 CONCLUSION AND FUTURE WORK

In this paper, we have described and demonstrated techniques and libraries in Agda that successfully scale up intrinsically-typed definitional interpreters to handle imperative object languages with non-trivial binding structures. While the resulting interpreters, e.g. for Middleweight Java, are certainly more complex than the simply-typed λ-calculus interpreter we started with, we claim that they still meet the goals of being concise, (relatively) comprehensible, and executable, while guaranteeing type safety for much more elaborate object languages.

However, much interesting future work remains. We have already described in Section 1.3 how we envision integrating this research in Spoofax [Kats and Visser 2010] to bring type-sound interpreters and formal methods to a wider audience of language designers. While we have illustrated the usefulness of intrinsically-typed techniques for developing type-sound interpreters, it is an open question how well the approach scales to more sophisticated object-language type systems (such as polymorphism, dependent types, ownership types, or strong updates) and features (such as concurrency). It also not clear how useful the intrinsically-typed approach is for defining and verifying other kinds of meta-programs (such as compilers, program transformations, or garbage collectors) and properties of object programs (such as complexity bounds or functional correctness). In addition to investigating how these features might be modeled using Agda, or other similar dependently-typed languages, as host language, it is also interesting to explore how these host languages might themselves be improved to better support such modeling.

ACKNOWLEDGMENTS

We thank the reviewers for their comments, and for pointing out the relation with monadic strength. We also thank Sam Lindley, Sven Keidel, and Wouter Swierstra for helpful discussions. This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206).

REFERENCES

Reynald Affeldt and Kazuhiko Sakaguchi. 2014. An Intrinsic Encoding of a Subset of C and its Application to TLS Network Packet Processing. *Journal of Formalized Reasoning* 7, 1 (2014), 63–104. https://doi.org/10.6092/issn.1972-5787/4317

Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 2018. Recalling a Witness. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2018, Los Angeles, CA, USA, January 8-13, 2018*. ACM.

Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra Monads for Free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 515–529. https://doi.org/10.1145/3009837.3009878

Guillaume Allais, James Chapman, Conor McBride, and James McKinna. 2017. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, 195–207. https://doi.org/10.1145/3018610.3018613

Thorsten Altenkirch and Ambrus Kaposi. 2016a. Normalisation by Evaluation for Dependent Types. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal (LIPIcs)*, Delia Kesner and Brigitte Pientka (Eds.), Vol. 52. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.FSCD.2016.6

Thorsten Altenkirch and Ambrus Kaposi. 2016b. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodik and Rupak Majumdar (Eds.). ACM, 18–29. https://doi.org/10.1145/2837614.2837638

Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In *Computer Science Logic, 13th International Workshop, CSL 99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings (Lecture Notes in Computer Science)*, Jörg Flum and Mario Rodríguez-Artalejo (Eds.), Vol. 1683. Springer, 453–468.

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 666–679. https://doi.org/citation.cfm?id=3009866

Robert Atkey. 2009. Parameterised notions of computation. *Journal of Functional Programming* 19, 3-4 (2009), 335–376. https://doi.org/10.1017/S095679680900728X

Lennart Augustsson and Magnus Carlsson. 1999. An exercise in dependent types: A well-typed interpreter. In *In Workshop on Dependent Types in Programming, Gothenburg*.

Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. 2016. Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2016.20

Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2017. Artifact Intrinsically-Typed Definitional Interpreters for Imperative Languages. https://github.com/metaborg/mj.agda

David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and YuTing Wang. 2014. Abella: A System for Reasoning about Relational Specifications. *J. Formalized Reasoning* 7, 2 (2014), 1–89. https://doi.org/10.6092/issn.1972-5787/4650

Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. 2012. Strongly Typed Term Representations in Coq. *Journal of Automated Reasoning* 49, 2 (2012), 141–159. https://doi.org/10.1007/s10817-011-9219-0

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

G. M. Bierman, M. J. Parkinson, and A. M. Pitts. 2003. *MJ: An imperative core calculus for Java and Java with effects*. Technical Report UCAM-CL-TR-563. University of Cambridge.

Richard S. Bird and Lambert G. L. T. Meertens. 1998. Nested Datatypes. In *Mathematics of Program Construction, MPC 98, Marstrand, Sweden, June 15-17, 1998, Proceedings (Lecture Notes in Computer Science)*, Johan Jeuring (Ed.), Vol. 1422. Springer, 52–67. https://doi.org/link/service/series/0558/bibs/1422/14220052.htm

Denis Bogdanas and Grigore Rosu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 445–456. https://doi.org/10.1145/2676726.2676982

Edwin Brady. 2013a. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Edwin Brady. 2013b. Programming and reasoning with algebraic effects and dependent types. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 133–144. https://doi.org/10.1145/2500365.2500581

Edwin Brady and Kevin Hammond. 2006. A verified staged interpreter is a verified compiler. In *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings*, Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen (Eds.). ACM, 111–120. https://doi.org/10.1145/1173706.1173724

Venanzio Capretta. 2005. General recursion via coinductive types. *Logical Methods in Computer Science* 1, 2 (2005). https://doi.org/10.2168/LMCS-1(2:1)2005

James Chapman. 2009. Type Theory Should Eat Itself. *Electronic Notes in Theoretical Computer Science* 228 (2009), 21–36. https://doi.org/10.1016/j.entcs.2008.12.114

Adam Chlipala. 2013. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press. https://doi.org/books/certified-programming-dependent-types

Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2009. Effective Interactive Proofs for Higher-order Imperative Programs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 79–90. https://doi.org/10.1145/1596550.1596565

Matteo Cimini, Dale Miller, and Jeremy G. Siek. 2016. Well-Typed Languages are Sound. *CoRR* abs/1611.05105 (2016). http://arxiv.org/abs/1611.05105

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott (Eds.). 2007. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science, Vol. 4350. Springer.

The Coq Development Team. 2017. *The Coq Proof Assistant Reference Manual*. http://coq.inria.fr/doc/

Nils Anders Danielsson. 2012. Operational semantics using the partiality monad. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, Peter Thiemann and Robby Bruce Findler (Eds.). ACM, 127–138. https://doi.org/10.1145/2364527.2364546

Nils Anders Danielsson and U Norell. 2017. The Agda standard library. https://github.com/agda/agda-stdlib

N. G. de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae* 34, 5 (1972), 381–392.

Benjamin Delaware, William R. Cook, and Don S. Batory. 2011. Product lines of theorems. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 595–608. https://doi.org/10.1145/2048066.2048113

Dominique Devriese and Frank Piessens. 2011. On the bright side of type classes: instance arguments in Agda. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 143–155. https://doi.org/10.1145/2034773.2034796

Sylvia Grewe, Sebastian Erdweg, Pascal Wittmann, and Mira Mezini. 2015. Type systems for the masses: deriving soundness proofs and efficient checkers. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Gail C. Murphy and Guy L. Steele Jr. (Eds.). ACM, 137–150. https://doi.org/10.1145/2814228.2814239

Peter Hancock and Anton Setzer. 2000. Interactive Programs in Dependent Type Theory. In *Computer Science Logic, 14th Annual Conference of the EACSL, Fischbachau, Germany, August 21-26, 2000, Proceedings (Lecture Notes in Computer Science)*, Peter Clote and Helmut Schwichtenberg (Eds.), Vol. 1862. Springer, 317–331. https://doi.org/link/service/series/0558/bibs/1862/18620317.htm

Robert Harper. 1994. A Simplified Account of Polymorphic References. *Inf. Process. Lett.* 51, 4 (1994), 201–206.

Robert Harper and Christopher A. Stone. 2000. A type-theoretic interpretation of standard ML. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, Gordon D. Plotkin, Colin Stirling, and Mads Tofte (Eds.). The MIT Press, 341–388.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505

Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, Reno/Tahoe, Nevada, 444–463. https://doi.org/10.1145/1869459.1869497

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 285–296. https://doi.org/10.1145/2103656.2103691

Gerwin Klein and Tobias Nipkow. 2005. Jinja is not Java. *Archive of Formal Proofs* 2005 (2005). https://doi.org/entries/Jinja.shtml Formal proof development.

Gerwin Klein and Tobias Nipkow. 2006. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems* 28, 4 (2006), 619–695. https://doi.org/10.1145/1146811

Anders Kock. 1972. Strong functors and monoidal monads. *Archiv der Mathematik* 23, 1 (Dec 1972), 113–120. https://doi.org/10.1007/BF01304852

Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. 2012. Declarative Name Binding and Scope Rules. In *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers (Lecture Notes in Computer Science)*, Krzysztof Czarnecki and Görel Hedin (Eds.), Vol. 7745. Springer, 311–331. https://doi.org/10.1007/978-3-642-36089-3_18

Joomy Korkut, Maksim Trifunovski, and Daniel R. Licata. 2016. Intrinsic Verification of a Regular Expression Matcher. (2016). http://dlicata.web.wesleyan.edu/pubs/ktl16regexp/ktl16regexp.pdf Unpublished draft.

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. https://doi.org/citation.cfm?id=3009855

Daniel R. Licata and Robert Harper. 2009. Positively dependent types. In *Proceedings of the 3rd ACM Workshop Programming Languages meets Program Verification, PLPV 2009, Savannah, GA, USA, January 20, 2009*, Thorsten Altenkirch and Todd D. Millstein (Eds.). ACM, 3–14. https://doi.org/10.1145/1481848.1481851

Florian Lorenzen and Sebastian Erdweg. 2016. Sound type-dependent syntactic language extension. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodik and Rupak Majumdar (Eds.). ACM, 204–216. https://doi.org/10.1145/2837614.2837644

Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas R. Cameron. 2012. Encoding Featherweight Java with assignment and immutability using the Coq proof assistant. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*, Wei-Ngan Chin and Aquinas Hobor (Eds.). ACM, 11–19. https://doi.org/10.1145/2318202.2318206

Conor McBride. 2010. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP 2010, Baltimore, MD, USA, September 27-29, 2010*, Bruno C. d. S. Oliveira and Marcin Zalewski (Eds.). ACM, 1–12. https://doi.org/10.1145/1863495.1863497

Conor McBride. 2011. Kleisli Arrows of Outrageous Fortune. (2011). Accepted for publication.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML, Revised.* MIT Press, Cambridge, MA, USA.

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (July 1991), 55–92.

Peter D. Mosses. 1999. *A Modular SOS for ML Concurrency Primitives.* BRICS Research Series RS-99-57. Department of Computer Science, Aarhus University.

Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 175–188. https://doi.org/10.1145/2628136.2628143

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2008. Hoare Type Theory, Polymorphism and Separation1. *J. Funct. Program.* 18, 5-6 (Sept. 2008), 865–911. https://doi.org/10.1017/S0956796808006953

Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics - With Isabelle/HOL.* Springer. https://doi.org/10.1007/978-3-319-10542-0

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Vol. 9632. Springer Berlin Heidelberg, Berlin, Heidelberg, 589–615. https://doi.org/10.1007/978-3-662-49498-1_23

Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *PLDI*. 199–208.

Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press, Cambridge, Massachusetts.

The Racket Development Team. 2017. Racket v6.11. http://blog.racket-lang.org/2017/10/racket-v6-11.html

John Reynolds. 2004. *The Meaning of Types – From Intrinsic to Extrinsic Semantics.* BRICS Research Series RS-00-32. Department of Computer Science, Aarhus University.

John C. Reynolds. 1972. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2 (ACM '72)*. ACM, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

Jeremy G. Siek. 2013. Type Safety in Three Easy Lemmas. http://siek.blogspot.co.uk/2013/05/type-safety-in-three-easy-lemmas.html.

Matthieu Sozeau. 2008. *Un environnement pour la programmation avec types dépendants. (An environment for programming with dependent types)*. Ph.D. Dissertation. University of Paris-Sud, Orsay, France. https://doi.org/tel-00640052

Andrei Stefanescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Rosu. 2016. Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 74–91. https://doi.org/10.1145/2983990.2984027

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the dijkstra monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 387–398. https://doi.org/10.1145/2491956.2491978

S. Doaitse Swierstra, Marcos Viera, and Atze Dijkstra. 2016. A Lazy Language Needs a Lazy Type System: Introducing Polymorphic Contexts. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages, IFL 2016, Leuven, Belgium, August 31 - September 2, 2016*, Tom Schrijvers (Ed.). ACM. https://doi.org/10.1145/3064899.3064906

Wouter Swierstra. 2009a. *A functional specification of effects*. Ph.D. Dissertation. University of Nottingham, UK. https://doi.org/OrderDetails.do?uin=uk.bl.ethos.514786 British Library, EThOS.

Wouter Swierstra. 2009b. A Hoare Logic for the State Monad. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 440–451. https://doi.org/10.1007/978-3-642-03359-9_30

Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2016. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 49–60. https://doi.org/10.1145/2847538.2847543

L. Thomas van Binsbergen, Neil Sculthorpe, and Peter D. Mosses. 2016. Tool support for component-based semantics. In *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*, Lidia Fuentes, Don S. Batory, and Krzysztof Czarnecki (Eds.). ACM, 8–11. https://doi.org/10.1145/2892664.2893464

Vlad A. Vergu, Pierre Néron, and Eelco Visser. 2015. DynSem: A DSL for Dynamic Semantics Specification. In *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland (LIPIcs)*, Maribel Fernández (Ed.), Vol. 36. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 365–378. https://doi.org/10.4230/LIPIcs.RTA.2015.365

Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël D. P. Konat. 2014. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz (Eds.). ACM, 95–111. https://doi.org/10.1145/2661136.2661149

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (November 1994), 38–94.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *POPL*. 224–235. https://doi.org/10.1145/640128.604150