Submitted to ICFP '09

A Universe of Binding and Computation

Daniel R. Licata Robert Harper

Carnegie Mellon University

{drl,rwh}@cs.cmu.edu

Abstract

In this paper, we construct a logical framework supporting datatypes that mix binding and computation, implemented as a universe in the dependently typed programming language Agda 2. We represent binding pronominally, using well-scoped de Bruijn indices, so types can be used to reason about the scoping of variables. We equip our universe with datatype-generic implementations of weakening, substitution, exchange, contraction, and subordination-based strengthening, so that programmers are provided these operations for free for each language they define. In our mixed, pronominal setting, weakening and substitution hold only under some conditions on types, but we show that these conditions can be discharged automatically in many cases. Finally, we program a variety of standard difficult test cases from the literature, such as normalization-by-evaluation for the untyped λ-calculus, demonstrating that we can express detailed invariants about variable usage in a program's type while still writing clean and clear code.

1. Introduction

There has been a great deal of research on programming languages for computing with *binding and scope* (bound variables, α-equivalence, capture-avoiding substitution). These languages are useful for a variety of tasks, such as implementing domain-specific languages and formalizing the metatheory of programming languages. Functional programming with binding and scope involves two different notions of function: *functions-as-data* and *functions-as-computation*. Functions-as-data, used to represent abstract syntax with variable binding, have an intensional, syntactic, character, in the sense that they can be inspected in ways other than function application. For example, many algorithms that process abstract syntax recur under binders, treating variables symbolically. On the other hand, functions-as-computation, the usual functions of functional programming, have an extensional character—a function from A to B is a black box that, when given an A, delivers a B. A function-as-data determines a function-as-computation by substitution (plugging a value in for a variable), but not every function-as-computation determines a function-as-data, because the syntax appropriate for a particular problem may not allow the expression of every black box.

In previous work (Licata et al., 2008), we began to study a programming language that provides support for both functions-as-data and functions-as-computation as two different types. Our framework provides one type constructor $\Rightarrow$ for functions-as-data, used to represent variable binding, and another type constructor $\supset$ for functions-as-computation, used for functional programming. This permits representations that mix the two function spaces. As a simple example of such integration, consider a syntax for arithmetic expressions constructed out of (1) variables, (2) numeric constants, (3) let binding, and (4) arbitrary binary primitive operations, represented by functions-as-computation of type $\mathtt{nat} \supset \mathtt{nat} \supset \mathtt{nat}$. In SML, we would represent this syntax with the following datatype:

```
datatype arith =
    Var     of var
  | Num     of nat
  | Letbind of arith * (var * arith)
  | Binop   of arith * (nat -> nat -> nat) * arith
```

We use ML functions(-as-computation) to represent the primops. However, because SML provides no support for functions-as-data, we must represent variable binding explicitly (with a type `var`), and code notions such as α-equivalence and substitution ourselves.

In contrast, our framework naturally supports *mixed datatypes* such as this one. We specify it by the following constructors:

$$\begin{array}{lcl} \mathsf{num} & : & \mathsf{arith} \Leftarrow \mathsf{nat} \\ \mathsf{lebind} & : & \mathsf{arith} \Leftarrow \mathsf{arith} \otimes (\mathsf{arith} \Rightarrow \mathsf{arith}) \\ \mathsf{binop} & : & \mathsf{arith} \Leftarrow \mathsf{arith} \otimes (\mathsf{nat} \supset \mathsf{nat} \supset \mathsf{nat}) \otimes \mathsf{arith} \end{array}$$

We use $\Rightarrow$ (functions-as-data) to represent the body of the letbind, and $\supset$ (functions-as-computation) to represent the primops.

Our framework takes a *pronominal* approach to the variables introduced by functions-as-data: variables are thought of as *pronouns* that refer to a designated binding site, and thus are intrinsically scoped. This is in contrast to the *nominal* approach taken by languages such as FreshML (Pitts and Gabbay, 2000; Pottier, 2007; Shinwell et al., 2003), where variables are thought of as *nouns*—they are pieces of data that exist independently of any scope. The pronominal approach inherently requires some notion of *context* to be present in the language's type system, so that variables have something to refer *to*; we write $\langle \Psi \rangle\ A$ as the classifier of a program of type A with variables Ψ. The practical advantage of these contextual types is that they permit programmers to express useful invariants about variable-manipulating code using the type system, such as the fact that a λ-calculus evaluator maps closed terms to closed terms.

In a pronominal setting, the interaction of functions-as-data and functions-as-computation has interesting consequences for the *structural properties* of variables, such as weakening (introducing a new variable that does not occur) and substitution (plugging a value in for a variable). For example, one might expect that it would be possible to weaken a value of type A to a function-as-data of type $D \Rightarrow A$. However, this is not necessarily possible when A itself is a computational function: Contextual computational functions of type $\langle \Psi \rangle\ A \supset B$ are essentially interpreted as functions from $\langle \Psi \rangle\ A$ to $\langle \Psi \rangle\ B$, and $\langle \Psi \rangle\ D \Rightarrow A$ classifies val-

ues of type A in an extended context $\Psi, x : D$. Now, suppose we are given a function f of type $\langle\Psi\rangle\ A \supset B$; we try to construct a function of type $\langle\Psi\rangle\ D \Rightarrow (A \supset B)$. This requires a function from $\langle\Psi, x : D\rangle\ A$ to $\langle\Psi, x : D\rangle\ B$. Since f is a black box, we can only hope to achieve this by pre- and post-composing with appropriate functions. The post-composition must take $\langle\Psi\rangle\ B$ to $\langle\Psi, x : D\rangle\ B$, which would be a recursive application of weakening. However, the pre-composition has a contravariant flip: we require a *strengthening* function from $\langle\Psi, x : D\rangle\ A$ to $\langle\Psi\rangle\ A$ in order to call `f`—and such a strengthening function does not in general exist, because the value of type A might be that very variable x. Similarly, substitution of terms for variables is not necessarily possible, because substitution requires weakening. Put differently, computational functions permit the expression of *side conditions* that inspect the context, which causes the structural properties to fail.

As a concrete example, the type $\langle\cdot\rangle\ (\mathsf{arith} \supset \mathsf{arith})$ classifies computational functions that take closed arithmetic expressions to closed arithmetic expressions. Such a function is likely defined by case-analysis over arithmetic expressions, giving cases for constants and binops and let-binding—but *not* for variables, because there are no variables in the empty context. Weakening such a function to type $\mathsf{arith} \Rightarrow (\mathsf{arith} \supset \mathsf{arith})$ *enlarges* its domain, asking it to handle cases that it does not cover.

What are we to do about this interaction of binding and computation? One option is to work in a less general setting, where it does not come up. For example, in nominal languages such as FreshML (Shinwell et al., 2003), the type of names is kept open-ended (it is considered to have infinitely many inhabitants). Thus, any computational function on syntax with binding must account for arbitrarily many names, and is therefore weakenable. However, many functions on syntax are only defined for certain classes of contexts (e.g., only closed arithmetic expressions can be evaluated to a numeral), and the nominal approach does not allow these invariants to be expressed in a program's type (though they can be reasoned about externally using a specification logic (Pottier, 2007)). Alternatively, in languages based on the LF logical framework, such as Twelf (Pfenning and Schürmann, 1999), Delphin (Poswolsky and Schürmann, 2008), and Beluga (Pientka, 2008), the structural properties always hold, because computational functions cannot be used in representations.

In our framework, we choose instead to bite the bullet and admit that weakening and substitution may not always be defined. Thus, we should be more careful with terminology, and say that the type $D \Rightarrow A$ classifies *values of type A with a free variable of type D*. In some cases, $D \Rightarrow A$ determines a function given by substitution, but in some cases it does not. In this sense, our approach is similar to representations of binding using *well-scoped de Bruijn indices* (Altenkirch and Reus, 1999; Bellegarde and Hook, 1994; Bird and Paterson, 1999), which are pronominal, because variables are represented by pointers into a context, but make no commitment to weakening and substitution. However, our framework improves upon such representations by observing that weakening and substitution *are* in fact definable generically, not for every type $D \Rightarrow A$, but under certain conditions on the types D and A. For example, returning to our failed attempt to weaken $A \supset B$ above, if variables of type D could never appear in terms of type A, then the required strengthening operation would exist. As a rough rule of thumb, one can weaken with types that do not appear to the left of a computational arrow in the type being weakened, and similarly for substitution. Our framework implements the structural properties generically but conditionally, providing programmers with the structural properties "for free" in many cases. This preserves one of the key benefits of working in a setting like LF where weakening and substitution are always defined.

In our previous work (Licata et al., 2008), we investigated the logical foundations of a pronominal approach to mixing binding and computation. In the present paper, we give an implementation of our framework, and we demonstrate the viability of our approach by programming some standard difficult test cases from the literature. For example, we implement normalization-by-evaluation (Berger and Schwichtenberg, 1991; Martin-Löf, 1975) (see Dybjer and Filinski (2000) for a tutorial) for the untyped λ-calculus, an example considered in FreshML by Shinwell et al. (2003). Our version of this algorithm makes essential use of a datatype mixing binding and computation, and our type system verifies that evaluation maps closed terms to closed terms.

Rather than implementing a new language from scratch, we construct our type theory as a *universe* in Agda 2 (Norell, 2007), a dependently typed functional programming language that provides good support for programming with inductive families, in the style of Epigram (McBride and McKinna, 2004). This means that we (a) give a syntax for the types of our type theory and (b) give a function mapping the types of our language to certain Agda types; the programs of our language are then the Agda programs of those types. This allows us to reuse the considerable implementation effort that has gone into Agda, and permits programs written using our framework to interact with existing Agda code. In addition, we can exploit generic programming within dependently typed programming (Altenkirch and McBride, 2003) to implement the structural properties. In our Agda implementation, we have chosen to represent variable binding using well-scoped de Bruijn indices.

In summary, we make the following technical contributions:

1. We show that our previous type theory for integrating binding and computation can be implemented as a universe in Agda. The types of the universe permit concise, "point-free" descriptions of contextual types: a type in the universe acts as a function from contexts to Agda types.
2. We implement a variety of structural properties for the universe, including weakening, substitution, exchange, contraction, and subordination-based strengthening (Virga, 1999). We generalize our previous work, where we implemented each of these structural properties independently, by abstracting out a single generic `map` function for datatypes that mix binding and computation, which can then be instantiated to each structural property. This `map` function accounts for the contravariant recursive calls needed to process computational functions.
3. We define the structural properties' preconditions computationally, so that our framework can discharge these conditions automatically in many cases. This gives the programmer free access to weakening, substitution, etc. (when they hold).
4. We program a variety examples, and demonstrate that we can express detailed invariants about variable usage in a program's type while still writing clean and clear code.

In this paper, we consider only a simply-typed universe of binding and computation, for writing ML-like programs that manipulate binding in a well-scoped manner. We have not yet considered how well our framework supports reasoning about the equational behavior of the structural properties, which will be necessary for dependent programming. Additionally, the companion code for this paper (available from `http://www.cs.cmu.edu/~drl/`) is written in "Agda minus termination checking," because many of our examples, like NBE for the untyped λ-calculus, require non-termination. We leave an investigation of totality to future work.

The remainder of this paper is organized as follows: In Section 2, we describe the type structure of our language and its semantics in Agda. In Section 3, we present a variety of examples. In Section 4, we discuss the structural properties' implementation.

2. Language Definition

2.1 Types

The grammar for the types of our language is as follows:

Defined atoms	D	::=	...
Var. Types	C	::=	(a subset of D)
Contexts	Ψ	::=	$[] \mid (\Psi, C)$
Types	A	::=	$0^+ \mid 1^+ \mid A \otimes B \mid A \oplus B \mid \mathsf{list} A \mid A \supset B$
			$\mathsf{D}^+\, D \mid D\# \mid \Psi \Rightarrow^* A \mid \Box A$
			$\forall_\mathsf{c}\, \psi.A \mid \exists_\mathsf{c}\, \psi.A \mid \forall_{\leq C} A$

The language is parametrized by a class of defined atoms D, which are the names of datatypes. A subset of these names are variable types, which are allowed to appear in contexts. This distinguishes certain types C which may be populated by variables from other types D which may not. This definition of `VarType` permits only variables of base type, rather than the full language of higher-order rules that we considered in previous work (Licata et al., 2008). Contexts are lists of variable types, written with 'cons' on the right.

The types on the first line have their usual meaning. The type $\mathsf{D}^+\, D$ is the datatype named by D. Following Delphin (Poswolsky and Schürmann, 2008), we include a type $D\#$ classifying only the variables of type D. The type $\Psi \Rightarrow^* A$ classifies inhabitants of A in the current context extended with Ψ. The type $\Box A$ classifies closed inhabitants of A. The types $\forall_\mathsf{c}$ and $\exists_\mathsf{c}$ classify universal and existential context quantification; $\forall_{\leq C} A$ provides bounded context quantification over contexts containing only the type C.

2.1.1 Agda implementation

We now represent these types in Agda. Those readers who are not fluent in dependent programming can find a review of Agda syntax, well-scoped de Bruijn indices, and universes in Appendix A. We represent defined atoms, variable types and contexts as follows:

```
DefAtom = DefinedAtoms.Atom

data VarType : Set where
  ▷ : (D : DefAtom) {_ : Check(DefinedAtoms.world D)}
    -> VarType

Vars = List VarType
```

`DefinedAtoms.Atom` is a parameter that we will instantiate later. `DefinedAtoms.world` returns true when `D` is allowed to appear in the context; `Check` turns this boolean into a proposition (`Check True` is the unit type; `Check False` is the empty type). A `VarType` is thus a pair of an atom along with the credentials allowing it to appear in contexts.

We represent the syntax of types in Agda as follows:

```
data Type : Set where
  -- types that have their usual meaning
  1⁺    : Type
  _⊗_   : Type -> Type -> Type
  0⁺    : Type
  _⊕_   : Type -> Type -> Type
  list_ : Type -> Type
  _⊃_   : Type -> Type -> Type
  -- datatypes and context manipulation
  D⁺    : DefAtom -> Type
  _#    : VarType -> Type
  _⇒*_  : Vars -> Type -> Type
  □     : Type -> Type
  ∀c    : (Vars -> Type) -> Type
  ∃c    : (Vars -> Type) -> Type
  ∀≤    : VarType -> Type -> Type
```

The only subtlety in this definition is that we represent the bodies of $\forall$`c` and $\exists$`c` by *computational functions* in Agda. This choice has some trade-offs: on the one hand, it means that the bodies of quantifiers can be specified by any Agda computation (e.g. by recursion over the domain). On the other hand, it makes it difficult to analyze the syntax of `Type`s, because there is no way to inspect the body of the quantifier. Indeed, this caused problems for our implementation of the structural properties, which we solved by adding certain instances of the quantifiers, which would otherwise be derived forms, as separate `Type` constructors. In future work, we may pursue a more syntactic treatment of the quantifiers (which would of course be easier if we had good support for variable binding...).

A *rule*, which is the type of a datatype constructor, pairs the defined atom being constructed with a single premise type (no/multiple premises can be encoded using 1^+ and $\otimes$):

```
data Rule : Set where
  _⇐_ : DefAtom -> Type -> Rule
```

We will make use of a few derived forms:

- We write (∀⇒ `A`) for (`∀c \Ψ -> Ψ ⇒* A`), and similarly for ∃⇒ (note that `\ x -> e` introduces an anonymous function). This type quantifies over a context Ψ and immediately binds it around A. Similarly, we write `[ Ψ ]* A` for $\Box\,(\Psi \Rightarrow^* A)$
- We write `(C ⇒ Ψ)` for ⇒`*` with a single premise.
- We write `(C ⁺)` for `(D⁺ C)` when `C` is a variable type.
- We write `bool` and `A option` for the encoding with 1^+ and $\oplus$.

2.2 Semantics

A universe is specified by a inductive datatype of *codes* for types, along with a function mapping each code to a `Set`. In this case, the `Type`s above are the codes, and the semantics is specified in Figure 1 by a function `< Ψ > A`, mapping a context and a `Type` to an Agda `Set`. The first six cases interpret the basic types of the simply-typed λ-calculus as their Agda counterparts, pushing the context inside to the recursive calls.

The next two cases interpret datatypes. We define an auxiliary datatype called `Data` which represents all of the data types defined in the universe. `Data` is indexed by a context and a defined atom, with the idea that the Agda set `Data Ψ D` represents the values of datatype `D` in context Ψ. For example, the values of `Data Ψ arith` will represent the arithmetic expressions defined by the signature given in the introduction. There are two ways to construct a datatype: (1) apply a datatype constructor to an argument and (2) choose a variable from Ψ. Constants are declared in a signature, represented with a predicate on rules `In`Σ `: Rule -> Set`, where `In`Σ `R` is inhabited when the rule `R` is in the signature. The first constructor, written as infix ·, pairs a constant with the interpretation of the constant's premise. The second constructor, ▷, injects a variable from Ψ into `Data`.[1] See the appendix for the definition of the type $\in$, which represents well-scoped de Bruijn indices (Altenkirch and Reus, 1999; Bellegarde and Hook, 1994; Bird and Paterson, 1999). A `DefAtom D` is in the context if there exist credentials `c` for which the `VarType` formed by `(▷ D {c})` is in the list Ψ.

Finally, we provide a collection of types that deal with the context: `Ψ ⇒* A` extends the context (we write + for append); □ `A` clears the context. The quantifiers ∀`c` and ∃`c` are interpreted as the corresponding Agda dependent function and pair types. Finally, ∀≤

[1] Agda allows overloading of datatype constructors between different types, and we tend to use ▷ for injections from one type to another, as with `VarType` above.

```
AllEq : Vars -> VarType -> Set
AllEq Ψ D = Check (List.all (eqVarType D) Ψ)

mutual
 data Data (Ψ : Vars) (D : DefAtom) : Set where
   _·_ : {A : Type}
       -> In⊤ (D ⇐ A)  ->  < Ψ > A  ->  Data Ψ D
   ▷   : {c : _} -> (▷ D {c}) ∈ Ψ  -> Data Ψ D

 <_>_ : Vars -> Type -> Set
 -- basic types
 < Ψ > 1⁺        = Unit
 < Ψ > 0⁺        = Void
 < Ψ > (A ⊗ B)  = (< Ψ > A) × (< Ψ > B)
 < Ψ > (A ⊕ B)  = Either (< Ψ > A) (< Ψ > B)
 < Ψ > (list A) =  List (< Ψ > A)
 < Ψ > (A ⊃ B)  =  (< Ψ > A)  ->  (< Ψ > B)
 -- data types
 < Ψ > (D⁺ D)   = Data Ψ D
 < Ψ > (D #)    = D ∈ Ψ
 -- context manipulation
 < Ψ > (Ψnew ⇒* A) = <  Ψ + Ψnew  > A
 < _ > (□ A)        = < [] > A
 < Ψ > (∃c τ)       = Σ \ Ψ' -> < Ψ > (τ Ψ')
 < Ψ > (∀c τ)       = (Ψ' : Vars) -> < Ψ > (τ Ψ')
 < Ψ > (∀≤ D A) =
   (Ψ' : Vars) -> AllEq Ψ' D -> < Ψ + Ψ' > A
```

Figure 1. Semantics

`D A` quantifies over contexts `Ψ'` for which `AllEq Ψ' D` holds. The type `AllEq` says that every variable type in Ψ is equal to the given type `D` (`List.all` is true when its argument is true on all elements of the list; `eqVarType` is a boolean-valued equality function for variable types). (We could internalize `AllEq Ψ' D` as a type `alleq D` in the universe, in which case the bounded quantifier could expressed as a derived form, but we have not needed `alleq D` in a positive position in the examples we have coded so far.)

Agda does not accept the positivity of `Data`, and indeed, in our examples below, we will write signatures for general recursive types. In future work, we would like to investigate ways of positivity-checking the signature `In⊤` and explaining the construction of the inductive types in the universe to Agda, e.g. using containers (Abbott et al., 2005).

We also define versions of □ and ∀⇒ that construct Agda `Set`s; these are convenient for typing inhabitants of the of elements of the universe.

```
⊡ : Type -> Set
⊡ A = < [] > A

∀⇒_ : Type -> Set
∀⇒_ A = (Ψ : Vars) -> < Ψ > A
```

2.3 Structural Properties

In Figure 2, we present the type signatures for the structural properties; this is the interface that users of our framework see.

For example, the type of substitution should be read as follows: for any `A` and `D`, if the conditions for substitution hold, then there is a function of type `(∀⇒ (D ⇒ A) ⊃ (D ⁺) ⊃ A)` (for any context, given a term of type `A` with a free variable, and something of type `D ⁺` to plug in, there is a term of type `A` without the free variable). Weakening coerces a term of type `A` to a term with an extra free variable; strengthening does the reverse; exchange swaps two variables; contraction substitutes a variable for a variable. We also include an n-ary version of weakening for use with the bounded quantifier: if `A` can be weakened with `D`, then `A` can be weakened with a whole context comprised entirely of occurrences of `D`.

```
subst : (A : Type) {D : VarType}
        {_ : Check(canSubst (un▷ Cut) A)}
      -> (∀⇒ (D ⇒ A) ⊃ (D ⁺) ⊃ A)

weaken : (A : Type) {D : VarType}
         {_ : Check (canWeaken (un▷ D) A)}
       -> (∀⇒ A ⊃ (D ⇒ A))

strengthen : (A : Type) {D : VarType}
             {_ : Check (canStrengthen (un▷ D) A)}
           -> ∀⇒ (D ⇒ A) ⊃ A

exchange2 : (A : Type) {D1 D2 : VarType}
          -> (∀⇒  (D2 ⇒ D1 ⇒ A)  ⊃  (D1 ⇒ D2 ⇒ A))

contract2 : (A : Type) {D : VarType}
          -> (∀⇒  (D ⇒ D ⇒ A)  ⊃  (D ⇒ A))

weaken*/bounded : (A : Type) (Ψ : Vars) {D : VarType}
                -> (AllEq Ψ D)
                -> {canw : Check (canWeaken (un▷ D) A)}
                -> (∀⇒ A ⊃ (Ψ ⇒* A))
```

Figure 2. Type signatures of structural properties

We discuss the meaning of the conditions (`canSubst`, etc.) below; in all of our examples, they will be discharged automatically by our implementation.

3. Examples

In this section, we illustrate programming in our framework, adapting a number of examples that have been considered in the literature (Pientka, 2008; Poswolsky and Schürmann, 2008; Shinwell et al., 2003). Throughout this section, we compare the examples coded in our framework with how they are/might be represented in Twelf, Delphin, Beluga, and FreshML. We endeavor to keep these comparisons objective, focusing on what invariants of the code are expressed, and what auxiliary functions the programmer needs to define. Aside from Twelf, we are not expert users of these other systems, and we welcome corrections from those who are. Several additional examples are available in the companion Agda code, including a translation from λ-terms to combinators, and a type checker for simply-typed λ-calculus terms.

To use our framework, we give a type `DefAtom` representing the necessary datatypes names, along with a datatype

```
data In⊤ : Rule -> Set where
```

defining the datatype constructors.

We use the following naming convention: Defined atoms are given names that end in `A`; e.g., for the signature for arithmetic expressions given in the introduction, we will define `natA` and `arithA`. For types of variables, we define `<atom>C` to be `<atom>A` injected into `VarType`:

```
arithC = ▷ arithA
```

We define `<atom>` to be the `Type` constructed by `D⁺ atomA`; e.g.:

```
nat  = D⁺ natA
arith = D⁺ arithA
```

3.1 Evaluating Arithmetic Expressions

We define a signature for the arithmetic example mentioned above:

```
  zero : In⊤ (natA ⇐ 1⁺)
  succ : In⊤ (natA ⇐ nat)
```

```
num     : In⊙ (arithA ⇐ nat)
binop   : In⊙ (arithA ⇐ arith ⊗
                        (nat ⊃ nat ⊃ nat) ⊗
                        arith)
letbind : In⊙ (arithA ⇐ arith ⊗ (arithC ⇒ arith))
```

Natural numbers are specified by zero and successor. Arithmetic expressions are given as a mixed datatype, with ⇒ used to represent the body of the `letbind` and ⊃ used to represent primops: Next, we define an evaluation function that reduces an expression to a number:

```
eval : ⊡ (arith ⊃ nat)
eval (num · n)                = n
eval (binop · (e1 , f , e2)) = f (eval e1) (eval e2)
eval (letbind · (e1 , e2))   = eval (subst arith _ e2 e1)
eval (▷ ())
```

Evaluation maps closed arithmetic expressions to natural numbers. Constants evaluate to themselves; binops are evaluated by applying their code to the values of the arguments; let-binding is evaluated by applying the `letbind`'s body `e2` to the expression `e1`, performing substitution,[2] and then evaluating the result. A simple variation would be to evaluate `e1` first and then substitute its value into `e2`. The final clause covers the case for variables with a refutation pattern: there are no variables in the empty context.

Comparison. This example provides a nice illustration of the benefits of our approach: Substitution is provided "for free" by the framework, which infers that it is permissible to substitute for `arithC` variables in `arith`. The type system enforces the invariant that evaluation produces a closed natural number.

It is not possible to define the type `arith` in Twelf/Delphin/Beluga, as LF representations cannot use computational functions. One could program this example in FreshML, but it would be necessary to implement substitution directly for `arith`, as FreshML does not provide a generic substitution operation.

Agda checks that `eval`'s pattern matching is exhaustive. However, Agda is not able to verify the termination of this function, as it recurs on a substitution-instance of one of the inputs. Setting aside the computational functions in `binop`, it would be possible to get the call-by-value version of this code to pass Twelf's termination checker, which recognizes certain substitution instances as smaller. We have not yet investigated how to explain this induction principle to Agda.

3.2 Closure-based Evaluator

Next, we implement a closure-based evaluator for the untyped λ-calculus. λ-terms and closures are represented by types `exp` and `clos` as follows:

```
lam  : In⊙ (expA ⇐ (expC ⇒ exp))
app  : In⊙ (expA ⇐ exp ⊗ exp)

closure : In⊙ (closA ⇐ (∃⇒ (expC ⇒ exp) ⊗
                            (expC #  ⊃  □ clos)))
```

Expressions are defined by the usual signature, as in LF. The type of closures, `clos`, is a recursive type with one constructor `closure`. The premise of `closure` should be read as follows: a closure is constructed from a triple `(Ψ , e , σ)`, where (1) Ψ is an existentially quantified context; (2) `e` is an expression in Ψ with an extra free variable, which represents the body of a λ-abstraction; and (3) σ is a substitution of of closed closures for all the variables in Ψ. We represent a substitution as a function that maps each expression variable in the context (classified by the type `expC #`) to a closure. The type of the premise provides a succinct description of all of this: ∃⇒ introduces the variables in the existentially quantified context into scope without explicitly naming the context; ⇒ extends the context with an additional variable; (expC #) ranges over all of the variables in scope. For comparison, the meaning of this type in Ψ is:

```
Σ \(Ψ' : Vars) -> (Data (Ψ + Ψ' ,, expC) expA) ×
                  (expC ∈ (Ψ + Ψ')  ->  Data [] closA)
```

(where we write `,,` for cons on the right).

Evaluation maps an expression to a closed closure (`□ clos`), given a substitution of closed closures for each expression variable in the context:

```
env : Type
env = expC #  ⊃  □ clos

eval : ⊡ (∀⇒ exp ⊃ env ⊃ □ clos)
eval Ψ (▷ x) σ =  σ x
eval Ψ (lam · e) σ =  closure · (Ψ , e , σ)
eval Ψ (app · (e1 , e2)) σ
 with eval Ψ e1 σ
... | closure · (Ψ' , e' , σ') =
  eval (Ψ' ,, expC) e'
       (extend{(□ clos)} _ σ' (eval Ψ e2 σ))
... | ▷ x = impossible x
```

A variable is evaluated by applying the substitution. A `lam` evaluates to the obvious closure. To evaluate an application, we first evaluate the function position. To a first approximation, the reader may think of Agda's `with` syntax as a `case` statement in the body of the clause, with each branch marked by `... |`. Case-analyzing the evaluation of `e1` gives two cases: (1) the value is constructed by the constructor `closure`; (2) the value is a variable.

In the first, case we evaluate the body of the closure in an extended environment. The call to the function `extend` extends the environment σ' so that the last variable is mapped to the value of `e2`. The definition of `extend` is as follows:

```
extend : {A : Type} {D : VarType}
       -> (∀⇒ (D # ⊃ A) ⊃ A ⊃ (D ⇒ D #) ⊃ A)
extend Ψ σ new i0 = new
extend Ψ σ new (iS i) = σ i
```

At the call site of extend, we must explicitly supply the type `A` (in this case `□ clos`) to help out type reconstruction. The underscore stands for the instantiation of the $\forall_{\Rightarrow}$, which is marked as an explicit argument, but can in this case be inferred.

The second case is contradicted using the function `impossible`, which refutes the existence of a variable at a non-`VarType`—which `clos` is, because we never wish to have `clos` variables.

The context argument Ψ to `eval` does not play an interesting role in the code, but Agda's type reconstruction requires us to supply it explicitly at each recursive call. In future work, we may consider whether this argument can be inferred. Agda is unable to verify the termination of this `eval`uator for the untyped λ-calculus, as one would hope.

When writing this code, one mistake a programmer might make is to evaluate the body of the closure in σ instead of σ', which would give dynamic scope. If we make this mistake, Agda highlights the occurrence of σ and helpfully reports the type error that `Ψ' != Ψ`, indicating that the context of the expression does not match the context of the substitution.

[2] The `arith` argument to `subst` is the type `A` in the `D ⇒ A` argument to substitution; Agda's type reconstruction procedure requires this annotation. The underscore is the context argument instantiating the $\forall_{\Rightarrow}$ in the type of `subst`. The credentials for performing substitution are marked as an implicit argument, so there is no evidence of it in the call to `subst`.

Comparison. It is unclear to us how one would express this example in Delphin or Beluga, as Delphin lacks existential context quantification and $\Box$, and Beluga lacks the parameter type `exp #`, so our definition of `clos` cannot be straightforwardly ported to either of these languages. Beluga provides a built-in type of substitutions, written [Ψ']Ψ, so one might hope to represent closures using the type $\exists\psi.([\psi, x : \texttt{exp}]\texttt{exp}) \times [.]\psi$. However, the second component of this pair associates an *expression* with each expression variable in ψ, whereas we want to associate a *closure* with each expression variable in ψ in this example. One could presumably port this code to FreshML (Shinwell et al., 2003), but the type system would not enforce the invariant that closures are in fact closed. To our knowledge, a proof of this property for this example has not been attempted in Pure FreshML (Pottier, 2007), though we know of no reason why it would not be possible.

3.3 Variable Manipulation

Next, we consider a suite of simple variable manipulations.

3.3.1 Size

First, we compute the size of a λ-term. Addition is defined as usual, with a contradictory variable case because no `natA` variables are allowed.

```
plus : ⊡ (nat ⊃ nat ⊃ nat)
plus (zero · _) m = m
plus (succ · n) m = succ · (plus n m)
plus (▷ x)      m = impossible x

size : (∀⇒  exp ⊃ □ nat)
size Ψ (▷ x)             = succ · (zero · _)
size Ψ (app · (e1 , e2)) = succ · (plus (size Ψ e1)
                                        (size Ψ e2))
size Ψ (lam · e)         = succ · (size (Ψ ,, expC) e)
```

Agda successfully termination-checks these functions.

The type of size expresses that it returns a closed natural number. For comparison, we implement a second version that does not make this invariant explicit:

```
size' : ⊡ (∀≤ expC (exp ⊃ nat))
size' Ψ bound (▷ x) = succ · (zero · _)
size' Ψ bound (app · (e1 , e2)) =
 succ · (plus' Ψ bound (size' Ψ bound e1)
                       (size' Ψ bound e2))  where
  plus' : ⊡ (∀≤ expC (nat ⊃ nat ⊃ nat))
  plus' Ψ b = weaken*/bounded (nat ⊃ nat ⊃ nat) Ψ b []
                              plus
size' Ψ bound (lam · e)         =
  strengthen nat _ (size' (Ψ ,, expC) bound e)
```

Without the $\Box$, size must return a number in context Ψ: in the application case, we must weaken `plus` into Ψ, and in the `lam` case we must strengthen the extra `expC` variable out of the recursive call. Strengthening expression variables out of natural numbers is permitted by our implementation of the structural properties because natural numbers cannot mention expressions; we use a subordination-like analysis to determine this (Virga, 1999). To do ensure that these weakenings and strengthenings are permitted, we type `size` with a bounded quantifier over `exp` variables to ensure that these

Comparison. The first version is similar to what one writes in FreshML, except in that setting there is no need to pass around a context Ψ. In the second version, the strengthening of the recursive result in the `lam` case is analogous to the need, in FreshML 2000 (Pitts and Gabbay, 2000), to observe that `nat` is pure (always has empty support); FreshML (Shinwell et al., 2003) does not require this.

In Beluga, one can express either the first or second versions. In Twelf and Delphin, one can only express the second variation, as these languages do not provide $\Box$. However, the Twelf/Delphin/Beluga syntax for weakening and strengthening is terser than what we have been able to construct in Agda: weakening is handled by world subsumption and is not marked in the proof term; strengthening is marked by pattern-matching the result of the recursive call and marking those variables that *do* occur, which in this case does not include the expression variable. For example, the `lam` case of size in Twelf looks like this:

```
- : size (lam ([x] E x)) (succ N)
      <- ({x : exp} size (E x) N).
```

Twelf's coverage checker verifies that expression variables can be strengthened out of natural numbers when checking this case. We would like to explore a similarly terse syntax for weakening/strengthening in future work.

3.3.2 Counting occurrences of a variable

A simple variation is to count the number of occurrences of a distinguished free variable. The input to this function has type `(expC ⇒ exp)`, and we count the occurrences of the bound variable:

```
cnt : ∀⇒ (expC ⇒ exp)  ⊃  □ nat
cnt Ψ (▷ i0) = succ · (zero · _)
cnt Ψ (▷ (iS _)) =  zero · _
cnt Ψ (app · (e1 , e2)) = plus (cnt Ψ e1) (cnt Ψ e2)
cnt Ψ (lam · e) = cnt (Ψ ,, expC) (exchange2 exp Ψ e)
```

In the first two cases, we pattern-match on the variable: when it is the last variable, the last variable occurs once; when it is not, it occurs zero times. The `lam` case recurs on the exchange of `e`, so that the last variable remains the one we are looking for. Agda fails to termination-check this example because it recurs on the result of exchange. Because this use of exchange is a common recursion pattern for `(exp -> exp)` in Twelf, we plan to consider a derived induction principle that covers this case in future work.

Comparison. Pattern-matching on variables is represented using higher-order metavariables in Twelf/Delphin/Beluga and using equality tests on names in FreshML. The exchange needed in the `lam` case is written as a substitution in the Twelf/Delphin/Beluga version of this clause. In Twelf one would write:

```
- : cnt ([x] lam ([y] E x y)) N
     <- ({y:exp} cnt ([x] E x y) N).
```

In the input to this clause, the metavariable `E`, which stands for the body of the function, refers to the last variable in the context (the `lam`-bound variable) as `y` and the second-last variable (the variable being counted) as `x`. In the recursive call, `y` is exchanged past the binding of `x`, so the instantiation `E x y` swaps "last" and "second-last".

3.3.3 Computing free variables

Next, we consider a function computing the free variables of an expression, of type `(∀⇒ exp ⊃ list (expC #))`—in any context, this function accepts an expression in that context and produces a list of variables in that context. This typing ensures that we do not accidentally return a bound variable.

```
remove : {D : VarType}
       -> (∀⇒  (D ⇒ list (D #))  ⊃  list (D #))
remove Ψ []             = []
remove Ψ (i0 :: ns)     =  (remove Ψ ns)
remove Ψ ((iS i) :: ns) = i :: (remove Ψ ns)
```

```
fvs : (∀⇒  exp  ⊃  list (expC #))
fvs Ψ (▷ x) = [ x ]
fvs Ψ (lam · e) = remove Ψ (fvs (Ψ ,, expC) e)
fvs Ψ (app · (e1 , e2)) = (fvs Ψ e1) ++ (fvs Ψ e2)
```

In the `lam` case, we use the helper function `remove` to remove the `lam`-bound variable from the recursive result. The function `remove` takes a list of variables, itself with a distinguished free variable, and produces a list of variables without the distinguished variable. If the programmer were to make a mistake in the second clause by accidentally including `i0` in the result, he would get a type error. Agda successfully termination-checks this example.

Comparison. For comparison with FreshML (Shinwell et al., 2003), the type given to `remove` here is analogous to their Figure 6:

```
remove : (<name> (name list)) -> name list
```

where <a>τ is a nominal abstractor. The authors comment that they prefer the version of `remove` in their Figure 5:

```
remove : name -> (name list) -> name list
```

where the name to removed is specified by the first argument, rather than using a binder.

Using dependent types, we can type this second version of remove as follows:

```
remove : (Ψ : Vars) (i : exp ∈ Ψ)
       -> List (exp ∈ Ψ) -> List (exp ∈ (Ψ - i))
```

where `Ψ - i` removes the indicated element element from the list. This type is of course expressible in Agda, but we have not yet integrated dependent types into our universe.

3.3.4 η-Contraction

In Twelf/Delphin/Beluga, one can recognize η-redices by writing a meta-variable that is not applied to all enclosing locally bound variables. E.g. in Twelf one would write

```
- : contract (lam [x] app F x) F.
```

The metavariable `F:exp` is bound outside the scope of `x`, and thus stands only for terms that do not mention `x`. (To allow it to mention `x`, we would bind `F:exp -> exp` and write `(F x)` in place of `F`.)

Unfortunately, Agda does not provide this sort of pattern matching for our encoding—pattern variables are always in the scope of all enclosing local binders—so we must explicitly call a strengthening function that checks whether the variable occurs:

```
strengthen? : ∀⇒ (expC ⇒ exp) ⊃ exp option
strengthen? Ψ (▷ i0)     = Inr _
strengthen? Ψ (▷ (iS i)) = Inl (▷ i)
strengthen? Ψ (app · (e1 , e2))
 with strengthen? Ψ e1 | strengthen? Ψ e2
... | Inl e1'          | Inl e2' = Inl (app · (e1' , e2'))
... | _                | _       = Inr _
strengthen? Ψ (lam · e)
 with strengthen? (Ψ ,, expC) (exchange2 exp Ψ e)
... | Inl e' = Inl (lam · e')
... | _      = Inr _

contract-η : ∀⇒ exp ⊃ exp option
contract-η Ψ (lam · (app · (f , ▷ i0))) = strengthen? Ψ f
contract-η Ψ _ = Inr <>
```

We conjecture that `strengthen?` could be implemented datatype-generically for all purely positive types (no $\supset$ or $\forall$c or $\forall\leq$). However, it is not possible to decide whether a variable occurs in the values of these computational types (cf. FreshML, where it is not possible to decide whether a name is in the support of a function).

This strengthening function is not an instance of the generic `map` that we define below, as it changes the type of the term (`exp` to `exp option`); in future work, we plan to consider a more general traversal that admits this operation.

```
napp : In⊠ (neuA ⇐ neu ⊗ sem)
neut : In⊠ (semA ⇐ neu)
slam : In⊠ (semA ⇐ (∀≤ neuC (sem ⊃ sem)))
```

```
reifyn : ∀⇒ ∀c \ Ψs -> (var2var neuC Ψs expC)
       ⊃  [ Ψs ]* neu  ⊃  exp
reifyn Ψe Ψs σ (▷ x) = ▷ (σ x)
reifyn Ψe Ψs σ (napp · (n , s)) =
  app · (reifyn Ψe Ψs σ n  , reify Ψe Ψs σ s)

reify :  ∀⇒ ∀c \ Ψs -> (var2var neuC Ψs expC)
      ⊃  [ Ψs ]* sem  ⊃ exp
reify Ψe Ψs σ (slam · φ) =
  lam ·  reify (Ψe ,, expC) (Ψs ,, neuC)
               (extendv2v Ψs Ψe σ)
               (φ [ neuC ] _ (neut · (▷ i0)))
reify Ψe Ψs σ (neut · n) = reifyn Ψe Ψs σ n
reify Ψe Ψs σ (▷ x) = impossible x
```

```
appsem : ∀⇒ sem ⊃ sem ⊃ sem
appsem _ (slam · φ) s2 = φ [] _ s2
appsem _ (neut · n) s2 = neut · (napp · (n , s2))
appsem _ (▷ x) _       =  impossible x

evalenv : Vars -> Type
evalenv Ψs = (expC #) ⊃ ([ Ψs ]* sem)

eval : ∀⇒ ∀c \Ψs -> evalenv Ψs
     ⊃  exp  ⊃  ([ Ψs ]* sem)
eval Ψe Ψs σ (▷ x) = σ x
eval Ψe Ψs σ (app · (e1 , e2)) =
     appsem Ψs (eval Ψe Ψs σ e1) (eval Ψe Ψs σ e2)
eval Ψe Ψs σ (lam · e) = slam · φ where
 φ : < Ψs > (∀≤ neuC (sem ⊃ sem))
 φ Ψ' bnd s' = eval (Ψe ,, expC) (Ψs + Ψ') σ' e where
    σ' : < Ψe > (expC ⇒ (evalenv (Ψs + Ψ')))
    σ' i0     = s'
    σ' (iS i) = weaken*/bounded sem Ψ' bnd Ψs (σ i)
```

Figure 3. Normalization by evaluation

3.4 Normalization by Evaluation

In Figure 3, we present a serious example mixing binding and computation, β-normalization-by-evaluation for the untyped λ-calculus. NBE works by giving the syntax a semantics in terms of computational functions (evaluation) and then reading back a normal form (reification). The NBE algorithm is similar to a Kripke logical relations argument, where one defines a type- and context-indexed family of relations $[\![A]\!]$ in Ψ. The key clause of this definition is:

$$([\![A \text{ arrow } B]\!] \text{ in } \Psi) = \forall \Psi'.\ ([\![A]\!] \text{ in } \Psi,\Psi') \supset ([\![B]\!] \text{ in } \Psi,\Psi')$$

That is, the meaning of A arrow B in Ψ is a function that, for any future extension of the context, maps the meaning of A in that extension to the meaning of B in that extension. In our type theory, we represent (a simply-typed version of) this logical relation as a datatype `sem`. The datatype constructor corresponding to the above clause would have the following type:

```
sem ⇐ (∀⇒ sem ⊃ sem)
```

However, for the argument to go through, we must ensure that the context extension `Ψ'` consists only of variables of a specific type `neu`, so we use a bounded context quantifier below.

We represent the semantics by the datatypes `neu` and `sem` in Figure 3. The type `neu` (neutral terms) consists of variables or neutral terms applied to semantic arguments (`napp`); these are the standard neutral proofs in natural deduction. A `sem` (semantic term) is either a neutral term or a semantic function. A semantic function of type `(∀≤ neuC (sem ⊃ sem))` is a computational function that works in any extension of the context consisting entirely of `neu` variables.

We define reification first, via two mutually recursive functions, `reifyn` (for neutral terms) and `reify` (for semantic terms). It is typical in logical relations arguments to use two independent contexts, one for the syntax and one for the semantics. Thus, we parametrize these functions by two contexts, one consisting for `neu` variables for the semantics, and the other consisting of `exp` variables for the syntax. We will write `Ψs` for the former and `Ψe` for the latter.

In the type of `reify`, we must name one of these contexts, because each context scopes over two disconnected parts of the type. We choose to name the semantic context and let the expression context be the ambient one. The outer $\forall_\Rightarrow$ thus binds the expression context, whereas we use the full binding form $\forall$`c` for the semantic context. The type of `reify` then says that, under some condition expressed by the type `var2var`, `reify` maps semantics in the semantic context (recall that `[ Ψ ]* A` stands for $\Box$ `(Ψ ⇒* A)`) to expressions (in the ambient expression context). The type `var2var C1 Ψ1 C2` means that every variable of type `C1` in `Ψ1` maps to a variable of type `C2` in the ambient context. It is defined in a library as follows:

```
var2var : VarType -> Vars -> VarType -> Type
var2var C1 Ψ1 C2 = ([ Ψ1 ]* (C1 #)) ⊃ (C2 #)
```

Even though `reify` is given a precise type describing the scoping of variables, its code is as simple as one could want. To reify neutral terms: The reification of a variable is the variable given in the substitution. The reification of an application is the application of the reifications. To reify semantic terms: The reification of a function `(slam · φ)` is the λ-abstraction of the reification of an instance of φ. In the recursive call, the expression context is extended with a new `exp` variable (which is bound by the `lam`) and the semantic context is extended with a new `neu` variable. We instantiate the semantic function φ, which anticipates extensions of the context, with this one-variable extension (`[ x ]` constructs a singleton list), and apply it to the variable. The library function `extendv2v` makes the "parallel" extension of a `var2var` in the obvious way, mapping the one new variable to the other:

```
extendv2v : {D1 D2 : VarType} -> (Ψs : Vars)
       -> ∀⇒ (var2var D1 Ψs D2)
             ⊃  D2 ⇒ (var2var D1 (D1 :: Ψs) D2)
extendv2v Ψs Ψe σ (i0)   = i0
extendv2v Ψs Ψe σ (iS i) = iS (σ i)
```

The neutral-to-semantic coercion is reified recursively, and we disallow `sem` variables from the context.

To define evaluation, we first define an auxiliary function `appsem` that applies one semantic term to another. This requires a case-analysis of the function term: when it is an `slam` (i.e. the application is a β-redex), we apply the embedded computational function, choosing the nil context extension, and letting the argument be `s2`. When the function term is neutral, we make a longer neutral term.

The type of `eval` is symmetric to `reify`, except the environment that we carry along in the induction maps expression variables to semantic *terms* rather than just variables. The type `evalenv Ψs` means that every expression variable in the ambient context is mapped to a semantic value in `Ψs`. A variable is evaluated by looking it up; an application is evaluated by combining the recursive results with semantic application. A `lam` is evaluated to an `slam` whose body φ has the type indicated in the figure. When given a context extension `Ψ'` and an argument `s'` in that extension, φ evaluates the original body `e` in an extended substitution. The new substitution `σ'` maps the λ-bound variable `i0` to the provided semantic value, and defers to σ on all other variables. However, σ provides values in `Ψs`, which must be weakened into the extension `Ψ'`. Fortunately, the bounded quantifier provides sufficient evidence to show that weakening can be performed in this case, because `sem`'s can be weakened with `neu` variables.

Normalization is defined by composing evaluation and reification. We define a normalizer for closed λ-terms as follows:

```
emptyv2v : ⊡ (var2var neuC [] expC)
emptyv2v ()

emptyevalenv : ⊡ (evalenv [])
emptyevalenv ()

norm : ⊡ (exp ⊃ exp)
norm e = reify [] [] emptyv2v (eval [] [] emptyevalenv e)
```

Our type system has verified the scope-correctness of this code, proving that it maps closed terms to closed terms. Amusingly, Agda accepts the termination of this evaluator for the untyped λ-calculus, provided that we have told it to ignore its issues with our universe itself—a nice illustration of the need for the positivity check on datatypes.

Comparison. The type `sem` is a truly mixed datatype: the premise `(∀≤ neuC (sem ⊃ sem))` uses both $\Rightarrow$ and $\supset$ (recall that there is a $\Rightarrow$ buried in the definition of $\forall\leq$). Because it uses $\supset$ in a recursive datatype, it is not representable in LF. Because it uses $\Rightarrow$, it would not even be representable in Delphin/Beluga extended with standard recursive types (that did not interact with the LF part of the language). Despite the fact that our implementation enforces strong invariants about the scope of variables, the code is essentially as simple as the FreshML version described by Shinwell et al. (2003), aside from the need to pass the contexts `Ψe` and `Ψs` along. Invariants about variable scoping can be proved in Pure FreshML (Pottier, 2007), but we would like to enforce these invariants within a type system, not using an external specification logic. Relative to a direct implementation in Agda, our framework provides the weakening function needed in the final case of `eval` for free.

4. Structural Properties

The structural properties are implemented by instantiating a generic traversal for `< Ψ > A`. The generic traversal has the following type:

```
map : (A : Type) {Ψ Ψ' : Vars}
    -> (Co A Ψ Ψ')  ->  < Ψ > A  ->  < Ψ' > A
```

This should be read as follows: for every `A Ψ Ψ'`, under the condition `Co A Ψ Ψ'`, there is a map from terms of type `A` in `Ψ` to terms of type `A` in `Ψ'`.

`Co : Type -> Vars -> Vars -> Set` is a *variable relation*, a type-indexed family of relations between two contexts. `Co` is in fact a parameter to the generic `map`; it provides: (1) a variable or term in `Ψ'` for each variable in `Ψ` that the traversal runs into; and (2) enough information to keep the traversal going inductively. We will instantiate `Co` with a specific relation for each traversal; e.g., for weakening with a variable of type `D`, `Co` will relate `Ψ` to `(Ψ ,, D)` under appropriate conditions on `D` and `A`.

For expository purposes, we present a slightly simplified version of the traversal first; the generalization is described with weakening below.

4.1 Compatibility

We account for the requirement on Co with the notion of *compatibility*. Suppose that Co and Contra are variable relations. We say that Co and Contra are compatible iff there is a term

```
compat : ({A : Type} {Ψ Ψ' : Vars}
           -> Co A Ψ Ψ' -> Compat Co Contra A Ψ Ψ')
```

where Compat is defined as follows:

```
Compat : Type -> Vars -> Vars -> Set
Compat (D #) Ψ Ψ' = (D ∈ Ψ) -> (D ∈ Ψ')
Compat (D⁺ D) Ψ Ψ' =
   ({A : Type} -> (c : In∑ (D ⇐ A)) -> Co A Ψ Ψ')
  × ({ch : _} -> (▷ D {ch}) ∈ Ψ  ->  < Ψ' > D⁺ D)
Compat 1⁺ Ψ Ψ' = Unit
Compat 0⁺ Ψ Ψ' = Unit
Compat (A ⊗ B) Ψ Ψ' =  Co A Ψ Ψ' × Co B Ψ Ψ'
Compat (A ⊕ B) Ψ Ψ' =  Co A Ψ Ψ' × Co B Ψ Ψ'
Compat (A ⊃ B) Ψ Ψ' =  Contra A Ψ' Ψ × Co B Ψ Ψ'
Compat (list A) Ψ Ψ' =  Co A Ψ Ψ'
Compat (Ψ0 ⇒* A) Ψ Ψ'  =  Co A (Ψ + Ψ0) (Ψ' + Ψ0)
Compat (□ A) Ψ Ψ' =  Unit
Compat (∀c τ) Ψ Ψ' =  (Ψ0 : _) -> Co (τ Ψ0) Ψ Ψ'
Compat (∀≤ D A) Ψ Ψ' =
   (Ψ0 : _) -> AllEq Ψ0 D -> Co A (Ψ + Ψ0) (Ψ' + Ψ0)
Compat (∃c τ) Ψ Ψ'    =  (Ψ0 : _) -> Co (τ Ψ0) Ψ Ψ'
Compat (∀⇒ A) Ψ Ψ' =
   (Ψ0 : _) -> Co A (Ψ + Ψ0) (Ψ' + Ψ0)
Compat (∃⇒ A) Ψ Ψ' =
   (Ψ0 : _) -> Co A (Ψ + Ψ0) (Ψ' + Ψ0)
Compat ([ Ψ ]* A) _ _ =  Unit
```

Compat imposes certain conditions on Co and Contra. For example, for variable types D #, it says that Co (D #) Ψ Ψ' induces a map from variables of type D in Ψ to variables in Ψ', For defined atoms D⁺ D, Compat says that Co (D⁺ D) Ψ Ψ' induces a map from variables in Ψ to terms[3] in Ψ', and that Co A Ψ Ψ' holds for every premise of every constant inhabiting D. In all other cases, Compat provides enough information to keep the induction going in map below. This amounts to insisting that Co (or Contra) holds on the subexpressions of a type in all appropriate contexts. For example, the condition for Ψ0 ⇒* A is that Co holds for A in the contexts extended with Ψ0. As discussed in Section 2, ∀⇒ and ∃⇒ and [_]* are actually built-in types, so they can be handled specially by the structural property tactics, though their semantics is the same as if they were derived forms.

In the usual monadic traversals of syntax (Altenkirch and Reus, 1999), Co _ Ψ Ψ' is taken to be (D : VarType) -> D ∈ Ψ -> < Ψ' > D—i.e. a realization of every variable in Ψ as a term in Ψ'. In our setting, this does not suffice to define a traversal, because (1) it does not provide for the contravariant flip necessary to process the domains of computational functions and (2) it does not allow us to express a *conditional* traversal, where conditions on the types ensure that the traversal will only find certain variables, and thus that only those variables need realizations. Compatibility ensures that Co provides enough information Contra to process the contravariant positions to the left of a computational arrow. Additionally, it permits conditional traversals: below, we will instantiate Co so that it is uninhabited for certain A.

4.2 Map

Suppose that Co and Contra are compatible, and assume a function

```
map' : (A : Type) {Ψ Ψ' : Vars}
     -> (Contra A Ψ Ψ') -> < Ψ > A -> < Ψ' > A
```

[3] In retrospect, bind may have been a better name than map, because Co maps variables to terms, as in monadic traversals (Altenkirch and Reus, 1999).

```
map : (A : Type) {Ψ Ψ' : Vars}
    -> (Co A Ψ Ψ') -> < Ψ > A -> < Ψ' > A
map (D⁺ Dat) co (▷ x) = (snd (compat co) x)
map (Dat #)  co x = ((compat co) x)
map (A ⊃ B) co e =
  \ y -> (map B (snd (compat co))
              (e (map' A (fst (compat co)) y)))
map 1⁺ co t = _
map 0⁺ co ()
map (A ⊗ B) {Ψ} {Ψ'} co  (e1 , e2) =
  (map A (fst (compat co)) e1 ,
   map B (snd (compat co)) e2)
map (A ⊕ B) co (Inl e1) =
  Inl (map A (fst (compat co)) e1)
map (A ⊕ B) co (Inr e1) =
  Inr (map B (snd (compat co)) e1)
map (list A) co [] = []
map (list A) co (x :: xs) =
  map A (compat co) x :: map (list A) co xs
map (Ψ0 ⇒* A) co e = map A (compat co) e
map (∀c τ)    co e =
  \ Ψ0 -> map (τ Ψ0) ((compat co) Ψ0) (e Ψ0)
map (∀≤ D A)  co e =
  \ Ψ0 ev -> map A ((compat co) Ψ0 ev) (e Ψ0 ev)
map (∃c τ)    co (Ψ0 , e) =
  Ψ0 , map (τ Ψ0) ((compat co) Ψ0) e
map (D⁺ Dat)  co (_·_ {A} c e) =
  c · map A (fst (compat co) c) e
map (∀⇒ A)   co e =
  \ Ψ0 -> map A ((compat co) Ψ0) (e Ψ0)
map (∃⇒ A)   co (Ψ0 , e) =
  Ψ0 , map A ((compat co) Ψ0) e
map (□ A)     co e = e
map ([ Ψ ]* A) co e = e
```

Figure 4. Map

that is the equivalent of map for the Contravariant positions.

Then we implement map in Figure 4. In the first and second cases, the compatibility of Co induces the map on variables that we need. In the third case, we pre-compose the function with map' and post-compose with map. In all other cases, map simply commutes with constructors, or stops early if it hits a boxed term.

4.3 Exchange/Contraction

Exchange and contraction are implemented by one instantiation of map. In this case, we take

```
Co A Ψ Ψ' = Contra A Ψ Ψ' = (Ψ ⊆ Ψ' × Ψ' ⊆ Ψ)
```

where ⊆ means every variable in one context is in the other. It is simple to show that these relations are compatible, because Co (a) provides the required action on variables directly and (b) ignores its type argument, so the compatibility cases for the type constructors are easy. Exchange is defined by instantiating the generic map with Co, where map' is taken be map itself, which works because Co = Contra.

4.4 Strengthening

Next, we define a traversal that strengthens away variables that, based on type information, cannot possibly occur. The invariant for strengthening is (essentially) the following:

```
Co : Type -> Vars -> Vars -> Set
Co A Ψ Ψ' = Σ \(D : VarType) ->
             Σ \(i : D ∈ Ψ) ->
              Check(irrel (un▷ D) A) × Id Ψ' (Ψ - i)
```

Here i, a pointer into the initial context Ψ is the variable to be strengthened away; the propositional equality constraint repre-

sented by the Identity says that the final context Ψ' is the initial context with i removed. The type Check(irrel (un▷ D) A) computes to Unit when strengthening is possible, and Void when it is not. Here un▷ simply peels off the injection of a defined atom into a VarType.

The crucial property of irrel is that Check(irrel (un▷ D) (D^+ D)) computes to Void. This forbids strengthening a variable of type D out of a term of type D. This is necessary because we cannot satisfy the usual compatibility condition for (D^+ D), which would require mapping all variables—including the variable-to-be-strengthened i—to a term of type D that does not mention i.

More generally, Check(irrel (un▷ D) A) means that variables of type D can never be used to construct terms of type A, which ensures that strengthening never runs into variables of the type being strengthened. The function irrel D A is defined by traversing the graph structure of types (i.e., it unrolls the definitions of defined atoms) and checks Sums.not (DefinedAtoms.eq D Dat) for each defined atom Dat it finds.

To account for contravariance, we must define strengthening simultaneously with weakening by irrelevant assumptions, which is similar. About 250 lines of Agda code shows that these two relations together are compatible. Their traversals are then defined by instantiating map twice, mutually recursively—each is passed to the other as map' for the contravariant recursive calls.

4.5 Weakening

In addition to weakening by irrelevant types (e.g. weakening a nat with an exp), we can weaken by types that do not appear to the left of a computational arrow (e.g., weakening an exp with an exp).

For a simple version of weakening, the variable relation is similar to strengthening, but uses a different computed condition, and flips the role of Ψ and Ψ' (now Ψ' is bigger):

```
Co : Type -> Vars -> Vars -> Set
Co A Ψ Ψ' = Σ \(D : VarType) ->
            Σ \(i : D ∈ Ψ) ->
              Check(canWeaken (un▷ D) A) × Id Ψ' (Ψ - i)
```

The function canWeaken is a different graph traversal than before: this time, we check irrel (un▷ D) A for the left-hand side of each computational arrow A ⊃ B. Weakening can then be defined using strengthening in contravariant positions, as irrel is exactly the condition that strengthening requires.

This suffices for a simple version of weakening. However, we can be more clever, and observe that types of the form ∀⇒ A are *always* weakenable, because their proofs are explicitly parametrized over arbitrary extensions of the context. Similarly, ∀≤ C A is weakenable with any context composed entirely of C's. Capitalizing on this observation requires a slight generalization of the traversal described above: computationally, weakening ∀⇒ A does not recursively traverse the proof of A, like map usually does, but stops the traversal and instantiates the context quantifier appropriately. Thus, our actual implementation of map is parametrized so that, for each type A, either it is given sufficient information to transform A directly (a function < Ψ > A -> < Ψ' > A), or it has enough information to continue recursively, as in the compatibility conditions described above. We use the former only for weakening the quantifiers (map < Ψ - i > (∀⇒ A) to < Ψ > (∀⇒ A)). We refer the reader to our Agda code for details. All told, weakening takes about 210 lines of Agda code to define and prove compatible.

4.6 Substitution

Substitution is similar to weakening and strengthening. Its invariant has the same form, using a condition canSubst (un▷ D) A. This condition ensures two things: (1) that D is irrelevant to the left-hand-sides of any computational arrow, so that substitution can be defined using weakening-with-irrelevant-assumptions in the contravariant position, and (2) that D is weakenable with all variable types bound by A, so that the term being plugged in for the variable can be weakened as substitution goes under binders. Substitution takes about 220 lines to define and prove compatible.

5. Related Work

We have provided comparisons with several other systems throughout the paper: Relative to LF-based systems such as Twelf (Pfenning and Schürmann, 1999), Delphin (Poswolsky and Schürmann, 2008), and Beluga (Pientka, 2008), our framework permits definitions that mix binding and computation; this is essential for defining the datatype sem in the NBE example. Relative to FreshML (Pottier, 2007; Shinwell et al., 2003), our framework enforces invariants about variable scoping in the type system. Such invariants can be proved in Pure FreshML (Pottier, 2007), but we would like to enforce these invariants within a type system, not using an external specification logic.

Aydemir et al. (2008) provide a nice overview of various techniques that are used to implement variable binding, including named, de Bruijn, {locally / globally} {named /nameless}, and weak higher-order abstract syntax (Bucalo et al., 2006; Despeyroux et al., 1995). More recently, Chlipala (2008) has advocated the use of parametric higher-order abstract syntax. We have chosen well-scoped de Bruijn indices (Altenkirch and Reus, 1999; Bellegarde and Hook, 1994; Bird and Paterson, 1999) for our Agda implementation, a simple representation that makes the pronoun structure of variables explicit. It would be interesting to investigate whether any benefits can be obtained by implementing our universe with a different representation. Relative to these techniques for representing binding, the advantage of our framework is that it provides datatype-generic implementations of the structural properties, including substitution. Both the Hybrid frameworks (Ambler et al., 2002; Capretta and Felty, 2007; Momigliano et al., 2007) and Hickey et al. (2006)'s work describe languages-within-a-language for specifying data with binding. However, to the best of our knowledge, these logical frameworks do not make the computational functions of the meta-language available for use in the framework (except inasmuch as they are used to represent binding itself). In contrast, our universe includes both ⇒ and ⊃.

In this work, we have created a universe of contextual types in Agda. Contextual types appear in Miller and Tiu's work (Miller and Tiu, 2003), as well as in contextual modal type theory (Nanevski et al., 2007). Miller and Tiu's self-dual ∇ connective is closely related to ⇒, also capturing the notion of a scoped constant. However, the ∇ proof theory adopts a logic-programming-based distinction between propositions and types, and ∇ binds a scoped term constant in a proposition. In our setting, ⇒ allows the meaning of certain propositions (defined atoms) to vary.

Fiore et al. (1999) and Hofmann (1999) give semantic accounts of variable binding. In a sense, the present paper gives a semantics for our type theory, where binding is represented by an indexed inductive definition. However, this is not an entirely satisfactory story, because it does not account for the datatype-generic definition of the structural properties. It would be interesting to explore a semantic characterization of the conditions under which substitution is definable (e.g., certain objects $(D \Rightarrow A)$ are exponentials, whereas others are not).

6. Conclusion

In this paper, we have constructed a logical framework supporting datatypes that mix binding and computation: Our framework is implemented as a universe in the dependently typed programming language Agda. Binding is represented in a pronominal manner, so the

type system can be used to reason about the scoping of variables. Our implementation provides datatype-generic implementations of the structural properties (weakening, subordination-based strengthening, exchange, contraction, and substitution). We have used the framework to program a number of examples, including a scope-correct version of the normalization-by-evaluation challenge problem discussed by Shinwell et al. (2003). We believe that these examples demonstrate the viability of our approach for simply-typed programming.

We hope also to have clarified the gap between LF-based systems for programming with binding, such as Twelf, Delphin, and Beluga, and a generic dependently typed programming language like Agda. For simply-typed programming, the benefits of the LF-based systems that we were unable to mimic include: (1) the ability to write pronominal variables with a named syntax; and (2) a convenient syntax for applying the structural properties. For example, the syntax of weakening and strengthening is relatively heavy in our setting. In Twelf, weakening is silent, and strengthening (including `strengthen?` used in the η-contraction example) is marked by saying which variables do occur, using a non-linear higher-order pattern. In our Agda implementation, weakening must be marked explicitly, and strengthening requires one to enumerate those variables that do not occur instead. However, the more convenient syntax seems within reach for a standalone implementation of our framework; e.g., weakening could be implemented using a form of coercive subtyping.

Of course, one way in which all of the LF-based systems outpace ours is that they support dependent types, which are crucial for representing logics and for mechanizing metatheory. Our most pressing areas of future work are to investigate a dependently typed extension of our universe, and to address the termination issues that we have deferred here. One key issue for the dependently typed version will be the equational behavior of the structural properties, which we have not yet investigated. We would hope that they have the right behavior up to propositional equality (otherwise there is a bug in the code presented here), but it remains to be seen whether we can get Agda's definitional equality to mimic the equations proved automatically by, e.g., Twelf. That said, the fact that the `map` function defined in Section 4 commutes with all term constructors definitionally in Agda gives us some hope in this regard.

Acknowledgements

We thank Noam Zeilberger for discussions about this work.

References

M. Abbott, T. Altenkirch, and N. Ghani. Containers: constructing strictly positive types. *Theoretic Computer Science*, 342(1):3–27, 2005.

T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl*, 2003.

T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *CSL 1999: Computer Science Logic*. LNCS, Springer-Verlag, 1999.

S. Ambler, R. L. Crole, and A. Momigliano. Combining higher order abstract syntax with tactical theorem proving and (co)induction. In *International Conference on Theorem Proving in Higher-Order Logics*, pages 13–30, London, UK, 2002. Springer-Verlag.

B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–15, 2008.

F. Bellegarde and J. Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 23(2–3):287–311, 1994.

U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *IEEE Symposium on Logic in Computer Science*, 1991.

R. S. Bird and R. Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

A. Bucalo, M. Hofmann, F. Honsell, M. Miculan, and I. Scagnetto. Consistency of the theory of contexts. *Journal of Functional Programming*, 16 (3):327–395, May 2006.

V. Capretta and A. Felty. Combining de Bruijn indices and higher-order abstract syntax in Coq. In *Proceedings of TYPES 2006*, volume 4502 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2007.

A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2008.

J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138, Edinburgh, Scotland, 1995. Springer-Verlag.

P. Dybjer and A. Filinski. Normalization and partial evaluation. In *Applied Semantics: International Summer School, APPSEM 2000*, volume 2395 of *Lecture Notes in Computer Science*, pages 137–192. Springer-Verlag, September 2000.

M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *IEEE Symposium on Logic in Computer Science*, 1999.

J. Hickey, A. Nogin, X. Yu, and A. Kopylov. Mechanized meta-reasoning using a hybrid HOAS/de Bruijn representation and reflection. In *ACM SIGPLAN International Conference on Functional Programming*, pages 172–183, New York, NY, USA, 2006. ACM.

M. Hofmann. Semantical analysis of higher-order abstract syntax. In *IEEE Symposium on Logic in Computer Science*, 1999.

D. R. Licata, N. Zeilberger, and R. Harper. Focusing on binding and computation. In *IEEE Symposium on Logic in Computer Science*, 2008.

P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. Rose and J. Shepherdson, editors, *Logic Colloquium*. Elsevier, 1975.

C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 15(1), 2004.

D. Miller and A. F. Tiu. A proof theory for generic judgments: An extended abstract. In *IEEE Symposium on Logic in Computer Science*, pages 118–127, 2003.

A. Momigliano, A. Martin, and A. Felty. Two-level hybrid: A system for reasoning using higher-order abstract syntax. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, 2007.

A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 2007. To appear.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *International Conference on Automated Deduction*, pages 202–206, 1999.

B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 371–382, 2008.

A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. In *European Symposium on Programming*, 2008.

F. Pottier. Static name control for FreshML. In *IEEE Symposium on Logic in Computer Science*, 2007.

M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *ACM SIGPLAN International Conference on Functional Programming*, pages 263–274, August 2003.

R. Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, 1999.

A. Agda Overview

In this section, we review Agda's syntax, we show a simple example of well-scoped de Bruijn indices, and we give a simple example of a universe. We refer the reader to the Agda Wiki (`http://wiki.portal.chalmers.se/agda/`) for more introductory materials.

A.1 Well-scoped de Bruijn indices in Agda

We review the representation of well-scoped de Bruijn indices as an indexed inductive definition (Altenkirch and Reus, 1999; Bellegarde and Hook, 1994; Bird and Paterson, 1999). A Agda data types are introduced as follows:

```
data List (A : Set) : Set where
  []   : List A
  _::_ : A -> List A -> List A
```

`Set` classifies Agda classifiers, like the kind `type` in ML or Haskell. Mixfix constructors are declared by using `_` in an identifier; e.g., `::` can now be used infix as in `Zero :: (Zero :: [])`.

Functions are defined by pattern-matching:

```
append : {A : Set} -> List A -> List A -> List A
append [] ys = ys
append (x :: xs) ys = x :: (append xs ys)
```

The curly-braces mark an implicit dependent function space; applications to implicit arguments are inferred. For example, we do not explicitly apply `append` to the `Set` argument `A`.

Indexed datatypes are defined using a notation similar to GADTs in GHC. For example, we define a datatype $\in$ representing indices into a list:

```
data _∈_ {A : Set} : A -> List A -> Set where
  i0 : {x : A}   {xs : List A} -> x ∈ (x :: xs)
  iS : {x y : A} {xs : List A} -> y ∈ xs -> y ∈ (x :: xs)
```

For any `Set A`, and terms `x` and `xs` of type `A` and `List A`, there is a type `x ∈ xs`. The first constructor, `i0`, creates a proof of `x ∈ (x :: xs)`—i.e. `x` is the first element of the list. The second constructor `iS`, creates a proof of `x ∈ (y :: xs)` from a proof that `x` is in the tail.

As a simple example of dependent pattern matching, we define an n-ary version of `iS`:

```
skip : {A : Set} (xs : List A) {ys : List A} {y : A}
     -> y ∈ ys -> y ∈ (append xs ys)
skip []       i = i
skip (x :: xs) i = iS (skip xs i)
```

We use an implicit-quantifier for all arguments but the list `xs`; explicit-quantifiers are written with parentheses instead of curly-braces. The fact that this code type-checks depends on the computational behavior of `append`; e.g., in the first case, the expression `append [] ys` reduces to `ys`, so we can return the index `i` unchanged.

Well-scoped syntax for the untyped λ-calculus is defined as follows:

```
data Term (Γ : List Unit) : Set where
  ▷   : <> ∈ Γ  ->  Term Γ
  Lam : Term (<> :: Γ)  ->  Term Γ
  App : Term Γ  ->  Term Γ  ->  Term Γ
```

The type `Unit` is defined to be the record type with no fields, with inhabitant written `<>`. We represent variables as indices into a list Γ containing elements of the one-element type `Unit`. (Such lists are isomorphic to natural numbers, but this illustrates the pattern for variables of more than one types.) The constructor $\triangleright$ makes a term from an index into Γ, which represents a variable. The body of `Lam` can refer to all of the variables in Γ, as well as a new bound variable represented by extending Γ to `(<> :: Γ)`. The K combinator $\lambda x.\lambda y.x$ is represented as follows: `Lam (Lam (▷ (iS i0)))`. The values of `Term Γ` correspond exactly to the λ-terms with free variables in Γ.

A.2 Universes

A universe is specified by a inductive datatype of *codes* for types, along with a function mapping each code to a `Set`. For example, the universe of finite sums and products is specified as follows:

```
data Type : Set where
  0⁺  : Type
  1⁺  : Type
  _⊗_ : Type -> Type -> Type
  _⊕_ : Type -> Type -> Type

Element : Type -> Set
Element 0⁺         = Void
Element 1⁺         = Unit
Element (τ1 ⊗ τ2) = (Element τ1) × (Element τ2)
Element (τ1 ⊕ τ2) = Either (Element τ1) (Element τ2)
```

In the right-hand side of `Element`, we write `A × B` for the Agda pair type, `Either A B` for the Agda sum type, etc.

Datatype-generic programs are implemented by recursion over the codes; e.g, every product and sum can be converted to a string:

```
_^_ = string-append

show : (τ : Type) -> Element τ -> String
show 0⁺ ()
show 1⁺ <> = "<>"
show (τ1 ⊗ τ2) (e1 , e2) =
  "< " ^ (show τ1 e1) ^ " , " ^ (show τ2 e2) ^ " >"
show (τ1 ⊕ τ2) (Inl e) = "left( " ^ (show τ1 e) ^ " )"
show (τ1 ⊕ τ2) (Inr e) = "right( " ^ (show τ2 e) ^ " )"
```

In the first clause, the empty parentheses are a refutation pattern, telling Agda to check that the type in question (in this case `Element 0⁺`) is uninhabited, and allowing the programmer to elide the right-hand side.

As another example, we will often view booleans as a two-element universe, with only `True` inhabited:

```
data Bool : Set where
  True  : Bool
  False : Bool

Check : Bool -> Set
Check True = Unit
Check False = Void
```

Because Agda implements extensionality for `Unit` (there is only one record with no fields), terms of type `Check True` can be left implicit and inferred.