Formalising Monotone Frameworks: A dependently typed implementation in Agda

by J.J. van Wijk
Supervisors: dr. J. Hage and dr. W.S. Swierstra

June 15, 2017

Abstract

Compiler builders use monotone frameworks to perform static analysis. Often, they omit proof of the domain of the function being a bounded semi lattice or only argue why their domain should be.

Unfortunately, mistakes in their argumentation could result in a non terminating static analysis. Since important software, such as compilers, often depends on the results of a static analysis, embedding a non terminating analysis causes such software to loop.

To assist programmers in their reasoning and to obtain machine verified proofs of termination, this thesis presents a verified implementation of embellished, extended and regular monotone frameworks in Agda. The implementation contains several algorithms to compute the least fixed point of a function that represents the flow of information for the static analysis on an input program. That program is written in a simplified procedural programming language.

To facilitate construction of termination proofs, we introduce a set of bounded semi lattice combinators which can be used to compose the domain of a transfer function. The bounded semi lattice constructed by the combinators includes a proof that the partial order is conversely well founded and thus implies the ascending chain condition.

Finally, we perform classical analyses on a intra-procedural and inter-procedural language.

Contents

1 Introduction

In a world where humanity becomes increasingly reliant on the correct behaviour of programs, the existence of program bugs, unintended or undesired program behaviour poses a threat to privacy as well as general safety. Programming languages have advanced type systems that can be used to prove the (non) existence of certain properties upon compilation of a program. These type systems also allow compiler based optimisations. Development in such advanced systems usually requires a resource investment and is often perceived as challenging. Hence, the majority of programs are written in languages without these guarantees. Furthermore, porting existing programs also comes with a price and thus a lot of old software is still being used today. Static analysis, the analysis of program code before execution, can reduce the gap between programming languages with advanced type systems and ones without. Some of the more recent static analyses are: automatic proving of termination or complexity [12], energy consumption analysis parameterized by devices [8] and DoS detection analysis with respect to regular expressions [25] all of which contribute to the solution of problems that society faces today.

A Monotone dataflow analysis framework, or Monotone Framework, a system developed by Kam and Ullman [14], is a system to analyse certain properties of code in some programming language. As such, we can use it to perform static analysis. It is built upon an algorithm first published by Kildall [15]. Kam and Ullman extend Kildall's work by adjusting his algorithm to allow a broader class of inputs by weakening the distributivity requirement to monotonicity. They also show that the desired solution for monotone frameworks differs from the maximal fixed point and show that the desired solution is undecidable in general.

Due to Rice's theorem, Monotone Frameworks can not precisely determine the absence of bugs, or non-intended behaviour, for all programs [22]. Because of this result, we restrict our solution to support just the properties that are finitely computable. Monotone frameworks have the ability to interpret program code abstractly by both abstracting over datatypes used in the program as well as abstracting over different execution

paths [5]. Evaluation of such a framework resembles verification of existence, or absence, of certain properties. The properties that can be verified by monotone frameworks are required to be determinable in a finite amount of steps to guarantee termination of the analysis.

We represent such properties using a lattice that satisfies the ascending chain condition and a monotone function. Tarski's fix point theorem then ensures us that all computations of monotone functions on the complete lattice will eventually stabilize, i.e. reach a fixed point.

People often prove the payload of the analysis being a lattice structure on paper. To avoid errors in such proofs, we can use machine verification. Section 2 briefly describes Agda, a dependently typed programming language which we will use as proof assistant to perform the machine verification. Tarksi's fixed point theorem is also described in section 2. To assist someone constructing a lattice, we provide some combinators in section 3. The combinators also combine proof of termination and other properties, hereby we alleviate a small burden on the programmers side.

We will then take a look at our approach to formalise monotone frameworks and the algorithms used to compute results in section 4. The algorithms considered are parallel iteration, chaotic iteration and maximal fixed point. The approach we take is based on, and we try to keep notation similar to, the description by F. Nielson, H.R. Nielson and C. Hankin [20]. Additionally, an example analysis is described. Furthermore, we show monotonicity is decidable when the domain is listable, spending computing power to avoid having to prove monotonicity.

Section 5 shows how to support programming languages that offer support for procedures in an *Embellished* monotone framework. We do this by showing that an embellished monotone framework can be represented as regular monotone framework. This way, when using the framework, we only have to proof simpler theorems and thus obtain the results of embellished frameworks for free.

To support analyses on languages that have dynamic type systems, we also formalize *Extended* monotone frameworks in section 6. Extended monotone frameworks, first published by L. Fritz and J. Hage [11], allow extension of the control flow graph during fix

point iteration and as such, forms a solution to the problem where the control flow graph is not staticly available. Our approach differs from theirs as we prove the version first for regular frameworks and then embellish it, to obtain extended-embellished frameworks, similar to how we obtained the results of embellished frameworks.

Finally, we look at other approaches that obtain similar results in section 7, discuss further work in section 8 and conclude in section 9. In this thesis an attempt is made to construct a formal dependently typed framework that contains sound, mechanically proven using Agda, algorithms to construct and perform analyses.

2 Preliminaries

2.1 Agda

Agda is a dependently typed functional programming language as well as a proof assistant. Agda was originally developed by Caterina Coquand in 1999 at Chalmers University. Ulf Norell rewrote Agda to version 2 in his PhD Thesis [21]. Agda is based on Per Martin-Löf's intuitionistic type theory as to have a solution to several paradoxes that occur in set theory. The Curry Howard correspondence ensures we can encode logical propositions inside Agda's type system [2]. An inhabitant of a type that resembles a proposition forms a proof that the proposition is sound. Agda's type system, being based on Martin Löf's type theory, does not support the axiom of the excluded middle. Classical proofs that use this axiom must be formulated in a constructive way.

Agda's syntax looks a lot like Haskell. It can be compiled to Haskell, OCaml as well as JavaScript.

Agda's type system gives us information about Agda's terms. The distinction between types and terms is less clear in Agda than in other type systems as Agda allows types to consist of terms and terms to consist of types. The type system is used to reason about the terms and is used by Agda's type and termination checker to verify properties like termination and general preservation of soundness, i.e. you are unable to use the type

system to prove unsound things. The type system is based upon Set theoretic concepts, but due to Russells paradox, types cannot just contain themselves as we would end up with an unsound type system. Hence, the predicative type system consists of linear ordered universes, i.e. types that contain types but not themselves, where the lowest universe is represented as `Set zero`. Because this type is used a lot it is abbreviated to `Set`. The type `Set` contains all the types that are independent from `Set` itself, for example `Bool` and `ℕ` are in `Set`, denoted `Bool ℕ : Set`. The type of polymorphic functions that have `Set n` in their type are contained in successor universes: `Set (suc n)`. Combinations of types of different universes are contained in the least upper bound, e.g. `Set zero → Set (suc zero) : Set (suc (zero ⊔ suc zero))`. Since the notation of `⊔` is commonly used in literature regarding monotone frameworks, we will refer to the Agda's `⊔` as `Level.⊔`. Members of `Set` can be constructed by using keywords as `record` and `data`. Furthermore, code can be organised using `module` and `import`.

Usually, programming languages come with a collection of predefined functions in the form of a prelude. Agda has no such thing. However, you are able to use compiler pragma's to define the usual data structures such as natural numbers and create a prelude yourself. This is already done for us in the Agda standard library [1].

2.2 Lattice Theory

To be able to formulate the finitely determinable properties, a monotone framework relies upon the codomain of the analysis being a bounded semi lattice.

A poset, abbreviation for partially ordered set, consists of a set $\mathbb{C}$ together with a binary relation `_⊑_ : ℂ → ℂ → Set` which represents a partial order on $\mathbb{C}$ requiring the following properties:

$$\forall x \in \mathbb{C} : x \sqsubseteq x \qquad \text{(reflexivity)}$$

$$\forall x, y \in \mathbb{C} : y \sqsubseteq x \wedge x \sqsubseteq y \Rightarrow x = y \qquad \text{(antisymmetry)}$$

$$\forall x, y, z \in \mathbb{C} : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z \qquad \text{(transitivity)}$$

A poset has a top element, which is also called supremum or maximum, if there exists an element $\top$ such that:

$$\top \in \mathbb{C} \land \forall x \in \mathbb{C} : x \sqsubseteq \top$$

When all possibly infinite sequences of the form:

$$a_0 \sqsubseteq a_1 \sqsubseteq \cdots \sqsubseteq a_k \sqsubseteq \cdots$$

eventually stabilize, i.e. :

$$\exists k \geq 0 : \forall j \geq k : a_j = a_k$$

it is said that the poset satisfies the Ascending Chain Condition (ACC).

Given $S \subseteq \mathbb{C}$, $x \in \mathbb{C}$ is an upperbound of S if and only if $\forall s \in S : s \sqsubseteq x$. Furthermore, $y \in \mathbb{C}$ is a least upperbound of S if and only if it is an upperbound of S and:

$$\forall c \in \mathbb{C} : (\forall s \in S : s \sqsubseteq c) \Rightarrow y \sqsubseteq c$$

The least upper bound of P, if it exists, is referred to as $\bigsqcup P$.

A join semi lattice, is a poset $(\mathbb{C}, \sqsubseteq, \sqcup)$ where $\sqcup$ is a binary total operator `⊔ : ℂ → ℂ → ℂ` such that `x ⊔ y = ⨆ {x , y}` . Dually, we can define a lowerbound and a greatest lower bound (or meet; $\sqcap$) to form a meet semi lattice.

A lattice is a poset $(\mathbb{C}, \sqsubseteq, \sqcup, \sqcap)$ such that $(\mathbb{C}, \sqsubseteq, \sqcup)$ is a join semi lattice and $(\mathbb{C}, \sqsubseteq, \sqcap)$ is a meet semi lattice. This lattice satisfies the following algebraic properties for all $a, b, c \in \mathbb{C}$:

$$(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c) \qquad \text{(associativity)}$$

$$a \sqcup b = b \sqcup a \qquad \text{(commutativity)}$$

$$a \sqcup a = a \qquad \text{(idempotency)}$$

$$a \sqcup (a \sqcap b) = a \qquad \text{(absorption)}$$

Dually, associativity, commutativity and idempotency laws also hold for $\sqcap$.

There are multiple ways to construct a (semi) lattice. We can provide a binary operator on some set such that the algebraic laws hold and prove them, or we can use a

binary relation together with the proof that for any two elements there exists a least upperbound and/or a greatest lowerbound. Both constructions are equivalent.

A complete lattice $(\mathbb{C}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a lattice such that: $\forall P \subseteq \mathbb{C} : \exists x \in \mathbb{C} : \bigsqcup P = x$. Since $\varnothing \subseteq \mathbb{C}$, a complete lattice can never be empty. Furthermore, there exists a greatest element or top: $\bigsqcup \mathbb{C} = \top$ and a lowest element or bottom: $\bigsqcup \varnothing = \bot$.

Now, as we will see later, it is not trivial to implement $\bigsqcap$ or $\bigsqcup$ in Agda because we represent $\mathbb{C}$ by some type and we cannot break this type down into subsets/types as Agda has no support for subtyping. However, a lattice that has a bottom and satisfies the ascending chain condition is a complete lattice [20]. Furthermore, it has been shown that satisfying ACC is equivalent to the partial order being conversely well founded. However, when speaking about well foundedness on any reflexive relation, usually the strict variant of the relation is used. This is because any non empty reflexive relation contains infinite chains: $x \sqsubseteq x \sqsubseteq \cdots$. Agda's standard library already has definitions for the order theoretic form of a lattice. Based on this definition, we represent a bounded semi lattice by a strict partial order, defined by $\sqcup$ or $\sqcap$, that is conversely well founded with a least element, i.e. $\bot$:

Agda

```
record BoundedSemiLattice a : Set (Level.suc a) where
  constructor boundedSemiLattice
  field
    ℂ : Set a -- Carrier type
    _⊔_ : ℂ → ℂ → ℂ -- Binary join
    _≟_ : (x y : ℂ) → Dec (x ≡ y) -- decidability of propositional equality
    ⊥ : ℂ -- Least element
    ⊥-isMinimal : (c : ℂ) -> ⊥ ⊑ c -- Proof that ⊥ is the least element
    ⊔-idem : (x : ℂ) → (x ⊔ x) ≡ x
    ⊔-comm : (x y : ℂ) → (x ⊔ y) ≡ (y ⊔ x)
    ⊔-cong₂ : {x y u v : ℂ} → x ≡ y → u ≡ v → (x ⊔ u) ≡ (y ⊔ v)
    ⊔-assoc : (x y z : ℂ) → ((x ⊔ y) ⊔ z) ≡ (x ⊔ (y ⊔ z))
    ⊐-isWellFounded : (x : ℂ) → Acc _⊐_ x
```

An element of type BoundedSemiLattice, must have inhabitants for all of its fields. Given an instance of such a record, we know that each field that represents some property

must have an inhabitant and thus the proposition it represents must be true.

Since our representation of a lattice (`BoundedSemiLattice`) satisfies the ascending chain condition, we know that (`ℂ` , `_⊔_`) actually forms a complete lattice. For any `ℂ` and `_⊔_` for which we construct a `BoundedSemiLattice` , we know that it is complete. The meet operation is not encoded in our definition however, so it does not represent a complete lattice. Furthermore, it seems that because we use the accessibility predicate, we do not have enough information to reconstruct the meet operation. The information must obviously be there upon proving the term, but we can not use the proof to enumerate all possible ascending chains, we only captured that all of them eventually terminate. Consequently, there is not enough information to invert the order of the lattice. The reason we choose this approach is because we now have a minimal set of requirements for a user as input. Also, the term that represents the proof of wellfoundedness seems lightweight, in the sense that we do not have to propagate a big term that shows all possible chains terminate through our program or keep it in memory [10].

From `⊔` and `≟` we derive other operators such as `⊑` , `⊑?` , `⊏` , `⊏?` , `⊒` , `⊒?` , `⊐` , `⊐?` . Since `⊔` and `≟` give precisely enough information to define these operators and properties we would not want to ask anyone who constructs a bounded semi lattice using this record to also define these operators (as that would be redundant work).

Agda allows you to create dependent records in which fields of the record are constructed of which the types are dependent on earlier fields. Thus we could have a definition:

Agda

```
record Example a : Set where
  field
    ℂ : Set a
    _⊔_ : Op₂ ℂ
  _⊑_ : Rel ℂ a
  x ⊑ y = (x ⊔ y) ≡ y
  field
    ⊥         : ℂ           -- Least element
    ⊥-isMinimal : (c : ℂ) -> ⊥ ⊑ c    -- Proof that ⊥ is the least element
```

We can now use `⊑` in types of later defined record fields such as `⊥-isMinimal`. However, if we would like to use `⊑` as a value when constructing the fields, we have a problem. This is because Agda considers the order of record fields relevant in the definition but irrelevant when instantiating, i.e. the intermediate values such as `⊑` are not brought into scope upon construction of such a value. However, we can manually bring them into scope by importing them from a separate module. This way, when constructing a value of such a record we can still access the dependent values by importing the correct modules and mimicking the hierarchy of the created record.

Given the join operation (`⊔`), decidable propositional equality (`≟`) and their algebraic laws, we can define the derivatives and prove properties about them.

The standard library already defined the properties shown below:

Agda

```
Transitive : ∀ {a ℓ} {A : Set a} → Rel A ℓ → Set _
Transitive {a} {ℓ} {A} _~_ = {i j k : A} → i ~ j → j ~ k → i ~ k

Reflexive : ∀ {a ℓ} {A : Set a} → Rel A ℓ → Set _
Reflexive _~_ = ∀ {x} → x ~ x

Antisymmetric : ∀ {a ℓ₁ ℓ₂} {A : Set a} → Rel A ℓ₁ → Rel A ℓ₂ → Set _
Antisymmetric _≈_ _≤_ = {x y : A} → x ≤ y → y ≤ x → x ≈ y

Asymmetric : ∀ {a ℓ} {A : Set a} → Rel A ℓ → Set _
Asymmetric _<_ = ∀ {x y} → x < y → ¬ (y < x)
```

Furthermore, the equational reasoning system supplied by the standard library can be used to enhance human interpretation. The types of terms that are to be proven transitively equal, using `_≡⟨_⟩_` are reflected in the code and surrounded by `begin` and `∎` to denote a proof. For example, we can see that parts from the type signature in `begin a ≡⟨ reason ⟩ b ∎ : a ≡ b` are reflected in the term. The equational reasoning system is used to show that `⊑` obeys the relational properties transitivity, reflexivity and antisymmetry with respect to propositional equality.

Agda

```
⊑-trans : Transitive _⊑_
⊑-trans {i} {j} {k} i⊔j≡j j⊔k≡k =
  begin
  i ⊔ k ≡⟨ ⊔-cong₂ refl (sym j⊔k≡k) ⟩
  i ⊔ (j ⊔ k) ≡⟨ sym (⊔-assoc i j k) ⟩
  (i ⊔ j) ⊔ k ≡⟨ ⊔-cong₂ i⊔j≡j refl ⟩
  j ⊔ k ≡⟨ j⊔k≡k ⟩
  k ∎

⊑-reflexive : Reflexive _⊑_
⊑-reflexive = ⊔-idem _

⊑-antisym : Antisymmetric _≡_ _⊑_
⊑-antisym {x} {y} x⊔y≡y y⊔x≡x =
  begin
  x
  ≡⟨ sym y⊔x≡x ⟩
  (y ⊔ x)
  ≡⟨ ⊔-comm y x ⟩
  (x ⊔ y)
  ≡⟨ x⊔y≡y ⟩
  y ∎
```

For ⊏ , ⊐ etc, we can show transitivity and asymmetry and several other properties of the operators are proven:

Agda

```
-- properties about ⊔ and ⊑
⊔-on-⊑ : {a b c d : ℂ} → a ⊑ b → c ⊑ d → (a ⊔ c) ⊑ (b ⊔ d)
⊔-on-left-⊑ : {a b c : ℂ} → a ⊑ c → b ⊑ c  → a ⊔ b ⊑ c
⊔-on-right-⊑ : {a b c : ℂ} → a ⊑ b → a ⊑ b ⊔ c
⊔-on-right-⊒ : {a b c : ℂ} → a ⊒ b → a ⊒ c → a ⊒ b ⊔ c
right-⊔-on-⊑ : {a b : ℂ} → a ⊑ (b ⊔ a)
left-⊔-on-⊑ : {a b : ℂ} → a ⊑ (a ⊔ b)
⊔-monotone-right : {x : ℂ} → Monotone _⊏_ (_⊔ x)
⊔-monotone-left : {x : ℂ} → Monotone _⊏_ (_⊔_ x)

-- properties about ⊑ and ≡
⊑-cong : (f : ℂ → ℂ) → (Monotone _⊏_ f) → {a b : ℂ} → a ⊑ b → f a ⊑ f b
≡⇒⊑ : {a b : ℂ} → a ≡ b → a ⊑ b
⋢⇒≠ : {a b : ℂ} → ¬ (a ⊑ b) → ¬ a ≡ b
⊑-split-left :  {a b c : ℂ} → a ⊔ b ⊑ c → a ⊑ c
⊑-split-right :  {a b c : ℂ} → a ⊔ b ⊑ c → b ⊑ c

-- properties about ⊐ and ⊏
⊏-asymmetric : Asymmetric _⊏_   -- ≡ ⊐-asymmetric
⊏-trans : Transitive _⊏_
⊐-trans : Transitive _⊐_
⊔-over-⊔ : ∀{a b c d} → (a ⊔ b) ⊔ (c ⊔ d) ≡ (a ⊔ c) ⊔ (b ⊔ d)
```

Note that we do not provide explicit proofs for the properties above here. Instead they, and other upcoming proofs, can be found in the source code [24]. When browsing the source code, we recommend to take a loot at `Index.agda` and start from there.

Furthermore, we extended the equational reasoning system with `_⊏⟨_⟩_` that proves `a ⊑ b` and can be chained transitively.

We can define `⨆ = foldr _⊔_ ⊥ : List ℂ → ℂ` and prove several properties:

Agda

```
x⊑⨆x : (x : ℂ) → (xs : List ℂ) → x ∈ xs → x ⊑ ⨆ xs
x⊒⨆ : (x : ℂ) → (xs : List ℂ) → All (x ⊒_) xs → x ⊒ ⨆ xs
⨆⊏⨆-pointwise : (xs ys : List ℂ) → Pointwise.Rel _⊏_ xs ys → ⨆ xs ⊑ ⨆ ys
⨆-mono : Monotone₂ _list⊑_ _⊏_ ⨆
```

The term `x⊑⨆x` is a proof that any element inside a finite list is at least as small as the lowest upper bound (`⨆`) of this list. Also, a property (`x⊒⨆`) is defined that says if some element is bigger than or equal to any element in the list, then this element is also bigger than or equal to the join of this list. Finally, we show that for any two lists `xs` and `ys` that `⨆ xs ⊑ ⨆ ys` whenever `xs` is pointwise `⊑` related to `ys` . Finally, we show that `⨆` is monotone with respect to `⊆` and `⊑` .

2.3 Tarski's fixed point theorem

In 1955, Alfred Tarski formulated his theory on fixed points, He showed the existence of a fixed point of any monotone function for any complete lattice. Additionally, the set of fixed points also forms a complete lattice with a least and greatest element. The fixed point theorem is formalized in this chapter.

A function f is monotone on some relation $\sqsubseteq$ when distribution of the function over the relation does not change the order of the elements:

Agda

```
Monotone : ∀{a ℓ} -> {ℂ : Set a} -> Rel ℂ ℓ ->
           (f : ℂ -> ℂ) -> Set (a Level.⊔ ℓ)
Monotone _⊑_ f = ∀ x y → x ⊑ y → f x ⊑ f y
```

To find a solution for a monotone framework, we need to compute a fixed point for each of the transfer functions. We are interested in the least fixed point because we do not want to include superfluous information. For example, $\{'a','b','c'\}$ is a fixed point for function $f\,x = x \cup \{'a','b'\}$. However, $\{'a','b'\}$ is also a fixed point but does not contain the superfluous information about $'c'$.

To find the least fixed point we can form a chain by iteratively applying f, starting with $\bot$:

$$(f_i \circ \cdots \circ f_1 \circ f_0)\bot \stackrel{\text{def}}{=} f^i \bot$$

Now, whenever we apply an additional f, we know that

$$f^i x \sqsubseteq f(f^i x)$$

And because of the ascending chain condition, or well foundedness of $\sqsupset$, we know that eventually (there exists some x) such that:

$$f^i x = f(f^i x) \stackrel{\text{def}}{=} l_0$$

which is a fixed point. Suppose that l_0 is this point. To prove that it is the least fixed point we assume that e is another fixed point and then prove inductively:

$$\bot \sqsubseteq e \qquad \text{(base)}$$

$$\forall c \in \mathbb{C} : c \sqsubseteq e \Rightarrow fc \sqsubseteq fe \Rightarrow fc \sqsubseteq fe \sqsubseteq e \Rightarrow fc \sqsubseteq e \qquad \text{(inductive)}$$

We can keep on applying f until we reach l_0 and obtain: $l_0 \sqsubseteq e$. To define Tarski's fixed point theorem in Agda, we will make use of the following types in which we assume $\mathbb{C}$ is the domain of a lattice:

Agda

```
IsFixedPoint : (c : ℂ) -> Set a
IsFixedPoint c = c ≡ f c

IsExtensivePoint : ℂ -> Set a
IsExtensivePoint c = c ⊑ f c

IsReductivePoint : ℂ -> Set a
IsReductivePoint c = f c ⊑ c
```

Any `element : ℂ` that satisfies `IsFixedPoint element` can be packed together with the proof:

Agda

```
record FixedPoint : Set a where
 constructor fp
 field
  element : ℂ
  isFixedPoint : IsFixedPoint element
```

Likewise, the least fixed point can be described:

Agda

```
IsLeastFixedPoint : (c : ℂ) -> Set a
IsLeastFixedPoint c = IsFixedPoint c × ((e : FixedPoint) -> c ⊑ FixedPoint.element e)

record LeastFixedPoint : Set a where
 constructor lfp
 field
  element : ℂ
  isLeastFixedPoint : IsLeastFixedPoint element
```

Then, we use these properties to describe a computation that will, given an ascending chain condition, eventually reach a fixed point.

Agda

```
-- To find our least fixed point, we need to start at an extensive point.
-- ⊥ is such a point.
fp-base : IsExtensivePoint ⊥
fp-base = ⊥-isMinimal (f ⊥)

-- if we have an extensive point c, then due to monotonicity f c is also extensive
fp-step : ∀{c} -> IsExtensivePoint c → IsExtensivePoint (f c)
fp-step = isMonotone
```

We use the above two cases in the function below to compute the fixed point.

Agda

```
-- given an extensive point, we find a fixed point by iteratively applying f
l₀-isFixedPoint : {c : ℂ} -> c ⊑ f c -> FixedPoint
l₀-isFixedPoint {c} x with c ≟ f c
l₀-isFixedPoint {c} x | yes p = fp c p
l₀-isFixedPoint {c} x | no ¬p = l₀-isFixedPoint (fp-step x)
```

To show that `l₀` is the least fixed point, we assume `e` is a fixed point, and then inductively show that `c ⊑ e`.

Agda

```
-- we can show that ⊥ ⊑ e
lfp-base : ⊥ ⊑ e
lfp-base = ⊥-isMinimal e

-- and inductively, c ⊑ e ⇒ f c ⊑ f e ⊑ e
lfp-step : {c : ℂ} -> c ⊑ e -> f c ⊑ e
lfp-step x = ⊑-trans (isMonotone x) (fixed⇒reductive p)
```

When combining the above induction cases with `l₀-isFixedPoint'` we get:

Agda

```
l₀-isLeastFixedPoint : {c : ℂ} -> c ⊑ f c
                     -> ((e : FixedPoint) -> c ⊑ element e )
                     -> LeastFixedPoint
l₀-isLeastFixedPoint {c} x x₁ with c ≟ f c
l₀-isLeastFixedPoint {c} x x₁ | yes p = lfp c (p , x₁)
l₀-isLeastFixedPoint {c} x x₁ | no ¬p =
  l₀-isLeastFixedPoint (fp-step x) (λ e → lfp-step e (x₁ e))
```

However, Agda is unable to determine that the above function terminates for any input. To see that recursive functions terminate, the termination checker poses a requirement on the arguments that they must become strictly smaller each recursive call. The standard library has some definitions in place, namely the accessibility predicate, which we can use to show Agda that our computation of a fixed point terminates when our lattice satisfies the ascending chain condition.

Agda

```
-- code from Agda stdlib
-- The accessibility predicate: x is accessible if everything which is
-- smaller than x is also accessible (inductively).
data Acc {a ℓ} {A : Set a} (_<_ : Rel A ℓ) (x : A) : Set (a ⊔ ℓ) where
  acc : (rs : ∀ y → y < x → Acc _<_ y) → Acc _<_ x

-- if all elements are accessible, then _<_ is well-founded.
Well-founded : ∀ {a ℓ} {A : Set a} → Rel A ℓ → Set _
Well-founded _<_ = ∀ x → Acc _<_ x
```

Given an element of the accessibility predicate, we can pattern match on it and use the resulting accessibility predicate to continue hereby letting Agda know the argument is decreasing. Our fixed point computation hence becomes:

Agda

```
l₀-isLeastFixedPoint : {c : ℂ} -> c ⊑ f c
                     -> ((e : FixedPoint) -> c ⊑ element e )
                     -> Acc _⊐_ c -> LeastFixedPoint
l₀-isLeastFixedPoint {c} p q (acc g) with c ≟ (f c)
l₀-isLeastFixedPoint {c} p q (acc g) | yes r = lfp c (r , q)
l₀-isLeastFixedPoint {c} p q (acc g) | no ¬r =
  l₀-rec (fp-step p) (λ e → lfp-step e (q e)) (g (f c) (p , ¬r))
```

Which we can invoke by starting from $\bot$:

Agda

```
-- We obtain the fixed point by starting from ⊥
-- application of l₀-isLeastFixedPoint with initial values
l₀-lfp : LeastFixedPoint
l₀-lfp = l₀-isLeastFixedPoint fp-base lfp-base (⊒-isWellFounded ⊥)

-- the actual least fixed point without proof
l₀ : ℂ
l₀ = LeastFixedPoint.element l₀-lfp

-- synonym for l₀ which we use throughout the rest of our code.
solveLeastFixedPoint : ℂ
solveLeastFixedPoint = l₀
```

3 Lattice combinators

This section describes several constructions to create bounded semi lattices. Some of these constructions are combinators that combine lattices or other structures to form a new lattice with the ascending chain condition. This way, when using such a semi lattice, the need to prove certain combinatorial properties is satisfied as they arise by construction. As a notational convention, each bounded semi lattice structure or combinator is annotated with `ᴸ`.

3.1 Unit

A trivial example of a bounded semi lattice is a domain with just a single element which is represented in Agda's standard library by `⊤` with element `tt`. In our solution we immediately lift `⊤` to any universe level `a`.

The proofs of `⊤` being a bounded semi lattice are straightforward, but nevertheless are shown to provide an easy example of how to define a new bounded semi lattice. Note that operators such as `⊑` and `⊒` are dependent on the definition of `⊔` and that we have to import the `Operators` submodule to use these definitions and their

associated properties.

Agda

```
Unitᴸ  : BoundedSemiLattice a
Unitᴸ  = boundedSemiLattice ℂ _⊔_ _≟_ ⊥ ⊥-isMinimal ⊔-idem ⊔-comm
                            ⊔-cong₂ ⊔-assoc ⊐-isWellFounded
 where
  ℂ : Set a
  ℂ = Level.Lift ⊤
  open Algebra.FunctionProperties {A = ℂ} _≡_
  _⊔_ : Op₂ ℂ
  x ⊔ y = Level.lift tt
  _≟_ : Decidable {A = ℂ} _≡_
  x ≟ y = yes refl
  ⊥ : ℂ
  ⊥ = Level.lift tt
  open Operators ℂ ⊥ _⊔_ _≟_
  ⊥-isMinimal : (c : ℂ) -> ⊥ ⊑ c
  ⊥-isMinimal c = refl
  ⊔-idem : Idempotent _⊔_
  ⊔-idem = const refl
  ⊔-comm : Commutative _⊔_
  ⊔-comm = const₂ refl
  ⊔-cong₂ : _⊔_ Preserves₂ _≡_ ⟶ _≡_ ⟶ _≡_
  ⊔-cong₂ = const₂ refl
  ⊔-assoc : Associative _⊔_
  ⊔-assoc = const₃ refl
  ⊐-isWellFounded : Well-founded _⊐_
  ⊐-isWellFounded (Level.lift tt) = acc (λ{(Level.lift tt) (a , b) → ⊥-elim (b a)})
```

3.2 Booleans

The next step is to create bounded semi lattices for domains with two elements such as `Bool`. There are two sensible orders that we can impose on `Bool`: either `false ⊏ true` or `true ⊏ false`. Therefore we can make two possible lattice instances wich we refer to as `Mayᴸ` and `Mustᴸ` where `⊔` corresponds to `∨` and `∧` respectively. The proof

of these structures being lattices is already defined in the standard library, to make them bounded we show that $\sqsupset$ is well founded by traversing all possible chains and showing that we eventually reach absurd.

Agda

```
⊐-isWellFounded : Well-founded _⊐_
⊐-isWellFounded false = acc (λ{ false (a , b) → ⊥-elim (b a)
                              ; true x → acc (λ{ false (a , b) → ⊥-elim (b a)
                                               ; true (a , b) → ⊥-elim (b a)})})
⊐-isWellFounded true = acc (λ{ false (a , b) → ⊥-elim (b a)
                             ; true (a , b)  → ⊥-elim (b a)})
```

3.3 $\leq$ ON NATURAL NUMBERS WITH ω

We can also define a bounded semi lattice for $\mathbb{N} \cup \{\omega\}$ and the $\leq$ relation. Note that the easy way to define this in Agda is by defining a meet semi lattice and extend this with a top element. In our definition for a BoundedSemiLattice we chose to use notation commonly used in literature to describe a join semi lattice. The identical structure can be used to describe a meet semi lattice.

Agda supports renaming of record fields, thus when using a bounded semi lattice that forms a meet semi lattice, we can rename the fields:

Agda

```
open BoundedSemiLattice L renaming (_⊔_ to _⊓_; ⊥ to ⊤; ⊥-isMinimal to ⊤-isMaximal; ..)
```

The partial order $(\mathbb{N} \cup \{\omega\} , \_\leq\_)$ forms such a meet semi lattice. For the natural numbers, a lot of theorems including wellfoundedness of $\leq$ are already defined in the standard library which we reuse with a little bit of adjustments to fit to the following datatype:

Agda

```
data ℕ∞ : Set where
  ω : ℕ∞
  nat : (n : ℕ) → ℕ∞
```

We can then form `⊓` and `⊤` as:

Agda

```
_⊓_ : Op₂ ℂ
ω ⊓ y = y
x ⊓ ω = x
nat n ⊓ nat m = nat (n ℕ⊓ m)

⊤ : ℂ
⊤ = ω
```

The well-foundedness and other properties are derived from the standard library.

3.4 Product

Given two bounded semi lattices L and R, the lattice constructed by the cartesian product, i.e. $L \times R$, and the product order, i.e. $(a, b) \sqsubseteq (c, d)$ if and only if $a \sqsubseteq c$ and $b \sqsubseteq d$, forms a bounded semi lattice.

Agda

```
_×ᴸ_ : ∀{α β} → BoundedSemiLattice α → BoundedSemiLattice β
       → BoundedSemiLattice (α Level.⊔ β)
```

We use the fact that `R.⊐-isWellFounded` and `L.⊐-isWellFounded` and the following lemma to construct the well foundedness proof:

Agda

```
lemma-wf : ∀{x y} -> Acc L._⊐_ x -> Acc R._⊐_ y -> Acc _⊐_ (x , y)
```

The lemma is implemented using case analysis where we show that no matter if we expand on L, R or both we always eventually reach $(L.\top, R.\top)$ after which there is no successor thus we reach absurd.

Furthermore, we provide some additional properties for elements of the new lattice.

Agda

```
⊑-by-× : ∀{x y z w} → x L.⊑ y → z R.⊑ w → (x , z) ⊑ (y , w)
×-⊔ : ∀{a b c d} → (a , b) ⊔ (c , d) ≡ (a L.⊔ c , b R.⊔ d)
```

We can use the binary product to form the n-ary product, in which the domain and order is the same for all positions in the tuple.

Agda

```
N-ary-×ᴸ : BoundedSemiLattice a → (n : ℕ) → BoundedSemiLattice a
N-ary-×ᴸ L zero = Unitᴸ
N-ary-×ᴸ L (suc zero) = L
N-ary-×ᴸ L (suc n) = L ×ᴸ (N-ary-×ᴸ L n)
```

3.5 Vector

An advantage of using a vector instead of nested tuples is that we can use already defined Agda functions such as lookup to access or manipulate the data. As vectors, i.e. finite lists of a fixed length, are equivalent to our n-ary products, we can use the equivalence to create a bounded semi lattice with a vector as domain. Let us first show Agda this equivalence between these two types by providing two conversion functions and showing that application of one after the other is an inverse operation i.e. $(f \circ g)\, x \equiv x$. Note that functions in Agda are total, so given these functions we have proof that for every term in one type there is exactly one element in the other type.

Agda

```
to-tuple : Vec ℂ n -> BoundedSemiLattice.ℂ (N-ary-×ᴸ L n)
to-vec : BoundedSemiLattice.ℂ (N-ary-×ᴸ L n) → Vec ℂ n

right-inverse-of : (x : BoundedSemiLattice.ℂ (N-ary-×ᴸ L n)) -> to-tuple (to-vec x) ≡ x
left-inverse-of : (x : Vec ℂ n) -> to-vec (to-tuple x) ≡ x
```

Above functions are then combined in a record type named $\leftrightarrow$ which represents the equivalence. In fact, it could be useful to construct a bounded semi lattice for any equivalence relation to another already defined bounded semi lattice. It turns out that using such an approach can be easier as we can omit more difficult proofs of accessibility and decidability.

Note that even though we show an equivalence between two types, it might not necessarily be the case that the bounded semi lattice you are trying to construct has an

equivalent order. For example, the product order is different from the lexicographical order.

Finally, we can use the equivalence relation to construct a bounded semi lattice for vectors using the product order:

Agda

```
fromBijectionᴸ :
 (A : Set a) → (L : BoundedSemiLattice b) → A ↔ L.𝕔 → BoundedSemiLattice a

Vecᴸ : BoundedSemiLattice a → (n : ℕ) -> BoundedSemiLattice a
Vecᴸ L n =
 fromBijectionᴸ (Vec (BoundedSemiLattice.𝕔 L) n) (N-ary-×ᴸ L n) Vecᴸ↔N-ary-×ᴸ
```

For the implementation of fromBijection we refer to the source code [24].

3.6 Powerset

In Agda, a powerset $\mathcal{P}(A)$ can be represented for finite domains (A) by mapping elements to natural numbers and then doing a lookup in a vector of booleans. A lookup resulting in true implies containment. Now at least two particular orders can be of interest for a powerset. The powerset where $\varnothing$ is represented as $\bot$ and A , the entire domain, is represented by $\top$ and $\sqcup \equiv \cup$ is called a powerset ordered by inclusion. The dual is a powerset ordered by exclusion. Analyses often are described as may or must analysis. Typically a may analysis uses a powerset lattice ordered by inclusion, whereas a must analysis uses a powerset ordered by exclusion.

Agda

```
𝒫ᴸ : BoundedSemiLattice _
𝒫ᴸ = Vecᴸ Mayᴸ n

𝒫ᴸ-by-inclusion : BoundedSemiLattice _
𝒫ᴸ-by-inclusion = 𝒫ᴸ

𝒫ᴸ-by-exclusion : BoundedSemiLattice _
𝒫ᴸ-by-exclusion = Vecᴸ Mustᴸ n
```

3.7 Total function space

Even though we do not assume that the domain ($\mathbb{C}$) of our lattice is finite, we can impose an order on total functions or mappings from other domains (A) that are finite to $\mathbb{C}$.

We can implement a total function space combinator in at least two ways:

1. by converting a function to a lookup in a vector.
2. by embedding the total function in Agda and using regular application.

Obviously, method two is preferred since it is more elegant but also does not require tabulation. However, an attempt for the second method resulted in difficulties regarding proving wellfoundedness. The first method is more easy to proof, because we already have a definition for a vector lattice. Thus, to create a total function space lattice we only have to show an equivalence between functions and vectors. We can only show this equivalence in Agda assuming function extensionality. Since Agda's propositional equality (≡) is intensional we must postulate this. Note that when compiling Agda to Haskell, execution will stop upon encountering a postulate. Therefore, we must make sure that we only make use of a postulate when proving properties that are irrelevant upon compilation.

Function extensionality is defined in the standard library, representing that two functions are extensionally equal when equality is preserved upon application for everything in the domain ignoring how the property is computed.

Agda

```
Extensionality : (a b : Level) → Set _
Extensionality a b = {A : Set a} {B : A → Set b}
  {f g : (x : A) → B x} →  (∀ x → f x ≡ g x) → f ≡ g
```

The equivalence TFS↔Vec$^{\llcorner}$, is to . to be shown here and can be found in the source code [24].

We can convert between the function and vector representation by:

Agda

```
mkVec : (A → L.ℂ) → Vec L.ℂ n
mkVec f = Data.Vec.map (λ x → f (from ⟨$⟩ x)) (allFin n)

mkFun : Vec L.ℂ n → A → L.ℂ
mkFun v x = lookup (to ⟨$⟩ x) v
```

The proof that they are inverses of each other is done using function extensionality:

Agda

```
right-inverse : (x : Vec L.ℂ n) → mkVec (mkFun x) ≡ x
right-inverse x = begin
 mkVec (mkFun x)
 ≡⟨⟩
 Data.Vec.map (λ x₁ → lookup (to ⟨$⟩ (from ⟨$⟩ x₁)) x) (tabulate (λ x₁ → x₁))
 ≡⟨ sym (tabulate-∘ (λ x₁ → lookup (to ⟨$⟩ (from ⟨$⟩ x₁)) x) Function.id)  ⟩
 tabulate (λ x₁ → lookup (to ⟨$⟩ (from ⟨$⟩ x₁)) x)
 ≡⟨ tabulate-allFin (λ x₁ → lookup (to ⟨$⟩ (from ⟨$⟩ x₁)) x) ⟩
 Data.Vec.map (λ x₁ → lookup (to ⟨$⟩ (from ⟨$⟩ x₁)) x) (allFin n)
 ≡⟨ map-cong (λ x₁ → cong (flip lookup x) (right-inverse-of x₁)) (allFin n) ⟩
 Data.Vec.map (λ x₁ → lookup  x₁ x) (allFin n)
 ≡⟨ map-lookup-allFin x ⟩
 x ∎
```

Agda

```
left-inverse : (f : A → L.ℂ) → mkFun (mkVec f) ≡ f
left-inverse f = fun-ext (λ x → begin
 mkFun (mkVec f) x
 ≡⟨⟩
 lookup (to ⟨$⟩ x) (Data.Vec.map (λ x → f (from ⟨$⟩ x)) (allFin n))
 ≡⟨ lookup-map (to ⟨$⟩ x) (λ x → f (from ⟨$⟩ x)) (allFin n) ⟩
 f (from ⟨$⟩ (lookup (to ⟨$⟩ x) (allFin n)))
 ≡⟨ cong (f $_) (subst (λ y → from ⟨$⟩ y ≡ x)
    (sym (lookup∘tabulate Function.id (to ⟨$⟩ x)))
    (left-inverse-of x)) ⟩
 f x
 ∎)
```

We capture this information in the equivalence `TFS↔Vecᴸ`, which we then use to create

the complete lattice by:

Agda

```
_-[_]→_ : BoundedSemiLattice (β Level.⊔ α)
_-[_]→_ = fromBijectionᴸ {β Level.⊔ α} {β} (A → L.𝕃) (Vecᴸ L n) TFS↔Vecᴸ
```

We also provide some properties to convert between different representations.

Agda

```
$-⊔ : (f g : A → L.𝕃) → (x : A) → (f ⊔ g) x ≡ f x L.⊔ g x
$-⊑ : (f g : A → L.𝕃) → (x : A) → f ⊑ g → f x L.⊑ g x
$-⊑' : (f g : A → L.𝕃) → ((x : A) → f x L.⊑ g x) → f ⊑ g
```

The downside of this solution is that we use tabulation to implement `⊔` and thus the solution is more computationally expensive than the other approach. We implement the other solution as well although the proof of well-foundedness is postulated.

3.8 Duality

Given an element `⊤`, such that `(a : 𝕃) → ⊤ ⊒ a`, a bounded semi lattice can be turned into another bounded semi lattice with inversed order by swapping `⊔` for `⊓` and `⊥` for `⊤`. We encounter difficulties when trying to define the dual semi lattice as our definition for the ascending chain condition does not contain the information of the chains itself, even though this information was present at the moment the proof was constructed. Thus, we have forgotten what possible chains can be formed but we know that all of them become strictly smaller and eventually terminate. The benefit of this approach is that it results in concise and perhaps easier to complete proofs but because we we cannot just enumerate over our domain (`𝕃`) it seems impossible without information of the actual chains to invert the accessibility proof.

3.9 Z-Top

In order to do the constant propagation analysis we use the lattice `ℤ⊤` which consist of the integers together with `⊤` and `⊥`. Here, `⊤` represents the case where we have

encountered conflicting information for a particular variable and thus cannot conclude that it is constant whereas $\bot$ represents the case where there is no information available at all.

Agda

```
data ℤ⊤⊥ : Set where
  top : ℤ⊤⊥
  in-ℤ : (i : ℤ) → ℤ⊤⊥
  bot : ℤ⊤⊥
```

we can then define $\sqcup$ as:

Agda

```
_⊔_ : Op₂ ℂ
top ⊔ y = top
y ⊔ top = top
in-ℤ i ⊔ in-ℤ j with i ≟ℤ j
in-ℤ i ⊔ in-ℤ j | yes p = in-ℤ i
in-ℤ i ⊔ in-ℤ j | no ¬p = top
in-ℤ i ⊔ bot = in-ℤ i
bot ⊔ a = a
```

and formulate the ascending chain condition:

Agda

```
-- top is accessible because there does not exist anything greater than top.
acc-top : Acc _⊐_ top
acc-top = acc (λ{ y (a , b) → ⊥-elim (b a)})

-- everything greater than i is accessible because
acc-i : ∀{i} → Acc _⊐_ (in-ℤ i)
acc-i {i} = acc
(λ{ -- top is accessible
    top x → acc-top
    -- there is no i ≠ j such that j is greater than i
    ; (in-ℤ i₁) (a , b) → ⊥-elim (b (cong in-ℤ (lemma a)))
    -- bot is not greater than i
    ; bot (a , b) → ⊥-elim (b a)})

-- everything greater than bot is accessible because
acc-bot : Acc _⊐_ bot
acc-bot = acc
 (λ{ -- top is accessible
    top (a , b) → acc-top
    -- all i are accessible
    ; (in-ℤ i) (a , b) → acc-i
    -- bot is not greater than bot
    ; bot (a , b) → ⊥-elim (b a)})
```

4 Monotone frameworks

A Monotone framework is a generalization of types of source code analysis for a given programming language. An instance of a monotone framework thus represents some static analysis. Certain commonalities with other static analyses are described in the framework. Analysis specific elements determine the instance.

A monotone framework requires a program written in some programming language as input. According to the programming language, we assign labels to program points. We then form the control flow graph, which consists of all possible paths in which infor-

mation can flow and generally is equivalent to possible execution paths. The nodes in the control flow graph thus consist of program points (or labels) and the edges between the nodes correspond to control flow. We then pick an initial point by label and assign initial values for the analysis. We determine what information flows through the control flow graph by specifying, for each label, a transfer function. The input to the transfer function is the join over the value of the predecessors to the program point and represents the information of possible execution paths up until the program point. The transfer function computes the information after the program point. For a much more in depth understanding of monotone frameworks we refer you to J.B. Kam and J.D. Ullman [14] or to F. Nielson, H.R. Nielson and C. Hankin in their book Principles of Program Analysis [20]. In our formalisation, we try to use an notation equivalent to Nielson et al where possible.

We define a monotone framework to be:

Agda

```
record MonotoneFramework a m : Set (Level.suc a) where
  field
    L : BoundedSemiLattice a -- Lattice instance
    ℱ : Label -> ℂ -> ℂ  -- Set of transfer functions indexed by label
    F : Graph n           -- Control flow graph
    E : List⁺ Label       -- Extremal labels
    ι : ℂ                 -- Extremal value
    -- The proof that all elements in the function space are monotone.
    ℱ-isMonotone : (l : Fin n) → Monotone _⊑_ (ℱ l)
```

To compute the result of an analysis, we compute two values at each program point:

- Context: `analysis○ ℓ′ = ⨆ { analysis● ℓ | ℓ ∈ predecessors F ℓ′ }`
- Effect: `analysis● ℓ′ = ℱ ℓ′ (analysis○ ℓ′)`

We are looking for a fixed point in a vector structure x such that:

Agda

```
∀ ℓ → lookup ℓ x ≡ ⨆ { ℱ ℓ  (analysis● ℓ) | ℓ ∈ predecessors F ℓ′ }
```

As context and effect are mutually dependent, we solve the equation by using fix point iteration.

4.1 Algorithms

4.1.1 Parallel iteration

There are several ways to compute a fixed point over a vector of lattices. The parallel version is a straightforward but naive approach. Each iteration, the results of the previous iteration are provided individually to each of the elements. Let `V×` denote the lattice formed by: `Vecᴸ L n ×ᴸ Vecᴸ L n` , then:

Agda

```
ιE : Label → ℂ
iE ℓ = if ⌊ ℓ′ ∈? E ⌋ then ι else ⊥

transfer-parallel : V×.ℂ → V×.ℂ
transfer-parallel (entry , exit) =
 let entry' =
  𝕍.map (λ ℓ′ → ιE ℓ′ ⊔ ⨆ (𝕃.map (flip lookup exit) (predecessors F ℓ′))) (allFin n)
 in (entry' , (tabulate ℱ 𝕍.⊛ entry'))
```

The algorithm can be used as a monotone function to compute a fixed point by using Tarski's theorem. Assume we have proofs of above functions being monotone, then:

Agda

```
-- parallel iteration solves the fixed point equation by
-- reusing results from the previous iteration.
parallel-iteration : ℂ
parallel-iteration =
  solveLeastFixedPoint transfer-parallel transfer-parallel-isMonotone
```

Parallel iteration can also be computed using the total function space lattice. In fact, this approach is a lot more concise. Instead of iterating a function that has a tuple of vectors in its domain, we now iterate a function that is quantified over all labels.

Agda

```
ParallelTotalFunctionSpace : BoundedSemiLattice a
ParallelTotalFunctionSpace = Label -[ n , Function.Inverse.id ]→ L

-- σ represents the previous iteration
parallel-tfs : P.𝐶 → P.𝐶
parallel-tfs σ ℓ′ = ιE ℓ′ ⊔ ⨆ (𝕃.map (λ ℓ → ℱ ℓ (σ ℓ)) (predecessors F ℓ′))
```

We then obtain the solution by:

Agda

```
parallel-tfs○ : V.𝐶
parallel-tfs○ = tabulate solveLeastFixedPoint
-- where tabulate replicates the function
-- solveLeastFixedPoint for every index of the vector.

parallel-tfs● : V.𝐶
parallel-tfs● = tabulate ℱ ⊛ parallel-tfs○
-- where ⊛ is pointwise application.
```

Note that we use tabulation to solve the least fixed point for each individual element. While this is an interesting and concise approach, it is computationally even worse than the other variant of parallel iteration without memoization.

4.1.2 Chaotic iteration

Chaotic iteration is an improvement over the naive approach as we propagate changes inside the vector as well. This is done using a fold. For each label `l`, we update the value for `l` by applying `l`'s transfer function to the result obtained from all previously computed labels. This way, transfer functions computing new information for an index in the vector can use the new information computed for previous indices. However, the order in which the functions do this is defined by the order of the vector. It can be the case that certain orders are more efficient than others.

Agda

```
transfer-chaotic : Vec ℂ n × Vec ℂ n → Vec ℂ n × Vec ℂ n
transfer-chaotic x =
  𝕍.foldr (λ x₁ → _)
  (λ{ ℓ′ (entry , exit) →
     (let entry' = ιE ℓ′ ⊔ ⨆ (𝕃.map (flip lookup exit) (predecessors F ℓ′))
      in (entry [ ℓ′ ]≔ entry' , exit [ ℓ′ ]≔ ℱ ℓ′ entry'))})
  x
  (allFin n)
```

Identical to the parallel version we can define the fixed point:

Agda

```
-- chaotic iteration solves the fixed point equation by
-- reusing results from the current iteration.
chaotic-iteration : ℂ
chaotic-iteration =
  solveLeastFixedPoint transfer-chaotic transfer-chaotic-isMonotone
```

4.1.3 Worklist algorithm (MFP)

Killdall introduced an algorithm to compute a fixed point for a directed graph model of program flow structure [15]. This algorithm is then modified by Kam and Ullman to a more general version that weakens the distributivity constraint to a monotonicity constraint. The algorithm is referred to as Maximal Fixed Point in literature even though the resulting fixed point being maximal depends on the shape of the semi lattice. Here, we use the algorithm with a bounded join semi lattice to compute the least fixed point. The algorithm uses a work list which contains values that have changed. Any value that depends on a changed value is to be updated. The algorithm keeps updating values until a fixed point is reached. The algorithm is described in pseudocode below, taken from Nielson, Nielson and Hankin [20].

Pseudocode

```
W := nil;
for all (ℓ,ℓ′) ∈ F do
  W := cons((ℓ,ℓ′),W);
for all ℓ in F or E do
  if ℓ ∈ E then Analysis[ℓ] := ι
           else Analysis[ℓ] := ⊥;

while W ≠ nill do
  ℓ := fst(head(W));
  ℓ′ := snd(head(W));
  W := tail(W);
  if f[ℓ](Analysis[ℓ]) ⋢ Analysis[ℓ′] then
    Analysis[ℓ′] := Analysis[ℓ′] ⊔ f(Analysis[ℓ]);
    for all ℓ′′ with (ℓ′,ℓ′′) ∈ F do
      W := cons((ℓ′,ℓ′′),W);

for all ℓ ∈ F do
  MFP○(ℓ) := Analysis[ℓ];
  MFP●(ℓ) := f(Analysis[ℓ]);
```

A straightforward implementation of the maximal fixed point algorithm in functional programming languages is to use the worklist as an argument to a function and check if the head of the list causes an update and if so, append all dependent values to the worklist and continue:

Agda

```
mfp₁ : (x : ℂ) → (workList : List Edge) → ℂ
mfp₁ x [] = x
mfp₁ x ((l₁ , l₂) ∷ workList) with f l₁ x ⊑? lookup l₂ x
mfp₁ x ((l₁ , l₂) ∷ workList) | yes p = mfp₁ x workList
mfp₁ x ((l₁ , l₂) ∷ workList) | no ¬p =
 mfp₁ (x [ l₂ ]≔ lookup l₂ x ⊔ f l₁ x) (lookup l₂ (adjacencyList F) 𝕃.++ workList)
```

First, the proofs in this paper and in the Agda code are about the functional version of MFP as above. Second, for now we just consider computation of the values before applying the transfer function, i.e. `MFP○`. Note that we encounter the same problem

as with Tarksi's fixed point theorem as the termination checker cannot deduce that our seemingly increasing recursive argument, i.e. the worklist, is actually decreasing. MFP terminates because the worklist will not contain an edge with the same value (as vector) twice, i.e. once one update has propagated through the program, applying the same update again does not affect the result. It is for this reason, the worklist strictly decrements. Hence, we must show Agda that the worklist is actually decreasing. Because our lattice satisfies the ascending chain condition and thus is conversely well founded, we know that all elements strictly greater than x are accessible. What remains for us to show is that the new value for ℓ' is strictly greater. Therefore, we can represent the result as a vector, which is also a lattice and thus it satisfies the ascending chain condition. The ascending chain condition for this vector structure is what we will use to show Agda MFP terminates.

Agda

```
transfer-mfp : (x : ℂ) → Acc _⊐_ x → (workList : List Edge) → ℂ
transfer-mfp x x₁ [] = x
transfer-mfp x x₁ ((l₁ , l₂) ∷ workList) with f l₁ x L.⊑? lookup l₂ x
transfer-mfp x x₁ ((l₁ , l₂) ∷ workList) | yes p = transfer-mfp x x₁ workList
transfer-mfp x (acc rs) ((l₁ , l₂) ∷ workList) | no ¬p =
 transfer-mfp x' (rs x' x⊏x') (lookup l₂ (adjacencyList F) 𝕃.++ workList)
  where x' : ℂ
        x' = x [ l₂ ]≔ f l₁ x ⊔ lookup l₂ x

        x⊑x' : x ⊑ x'
        x⊑x' = ⊑-by-element n L x l₂ (f l₁ x ⊔ lookup l₂ x) L.left-⊔-on-⊑

        x≠x' : ¬ x ≡ x'
        x≠x' x≡x' = contradiction
           (L.begin
           ℱ l₁ (lookup l₁ x) L.⊔ lookup l₂ x
           L.≡⟨ sym (lookup∘update l₂ x (f l₁ x ⊔ lookup l₂ x)) ⟩
           lookup l₂ x'
           L.≡⟨ sym (cong (lookup l₂) x≡x') ⟩
           lookup l₂ x
           L.∎) ¬p

        x⊏x' : x ⊏ x'
        x⊏x' = x⊑x' , x≠x'

maximal-fixed-point : ℂ
maximal-fixed-point = transfer-mfp ⊥ (⊐-isWellFounded ⊥) F
```

Unfortunately, this definition of MFP contains no proof that the result is a fixed point. We can solve this by either manually constructing the proof for this MFP function or by rewriting the MFP algorithm into a structure that resembles a fixed point operation much more clearly. For example, we can imagine a function `mfp-monotone-step : ℂ → ℂ`, that when evaluated to the fixed point has the same result as mfp with equal time complexity.

Let us take a look at manually proving that our mfp computation results in a fixed point.

When we look at Tarski's theorem, we see that the fix point we compute is a fix point for a single domain. In this case, our domain is structured in a specific manner that we have to formally describe in order to use Tarski's theorem. Perhaps a more easy approach is to apply the structure to the fixed point theorem to obtain different definitions for more specific fixed points.

In the case of some vector structure like mfp, we could use a fix point definition such as:

Agda

```
IsFixedPoint : Vec 𝕃 n → Set a
IsFixedPoint c = (ℓ′ : Fin n) →
 lookup ℓ′ initial ⊔ ⨆ (𝕃.map (flip f c) (predecessors F ℓ′)) ≡ lookup ℓ′ c
```

That encodes that for each position ℓ' in the vector, the value at ℓ' is equal to the maximal value of the transfer function applied to all predecessors, with respect to the flow, and some initial value. Given a definition for a fixed point, we also obtain other definitions and properties similar to the more basic fixed point definition. Note that by including the initial value we can find a fixed point for any value in the domain although they now are relative to the initial values. Therefore, the least fixed point can change when using different initial values.

Agda

```
IsReductivePoint : V.𝕃 → Set a
IsReductivePoint c =
 (ℓ ℓ′ : Label) → (ℓ , ℓ′) ∈ F → lookup ℓ′ initial ⊔ f ℓ c ⊑ lookup ℓ′ c

fixed⇒reductive : (c : V.𝕃) → IsFixedPoint c → IsReductivePoint c
```

We then show that the result of the maximal fixed point algorithm is always a least fixed point according to the above definition by proving the following theorem:

Agda

```
mfp :
 -- for all vectors x
    (x : V.ℂ)
 -- that are below or equal to all other fixed points
  → (K : ((y : FixedPoint) → x V.⊑ fp y))
 -- and of which all greater values are accessible
  → Acc V._⊐_ x
 -- and is above or equal to the initial value
  → initial V.⊑ x
 -- and for all work lists
  → (workList : List Edge)
 -- that have all of their elements originating from the flow graph
  → ((e : Edge) → e ∈ workList → e ∈ F)
 -- and such that all two labels that form an edge in the flow graph are
 -- either contained in the work list or the value at ℓ′ in x is bigger or equal to
 -- the transfer function applied over the value at ℓ in x.
  → (I : ((ℓ ℓ′ : Label) → (ℓ , ℓ′) ∈ F →
     ((ℓ , ℓ′) ∈ workList) ⊎ (lookup ℓ′ x ⊒ ℱ ℓ (lookup ℓ x))))
 -- and such that for all ℓ′ the value at ℓ′ in x is less or equal to the maximal value
 -- of the transfer function applied over all predecessors and the initial value at ℓ′.
 -- i.e. we stay below our definition of the fixed point
  → (J : ((ℓ′ : Label) →
     lookup ℓ′ x ⊑ lookup ℓ′ initial ⊔ ⨆ (𝕃.map (flip f x) (predecessors F ℓ′))))
 -- there exists a fixed point, such that it is smaller than all other fixed points
  → Σ[ c ∈ FixedPoint ] ((y : FixedPoint) → fp c V.⊑ fp y)
```

We can find arguments that satisfy above parameters by taking the entire control flow graph as initial worklist and for x we can take the vector consisting of initial values. We then show that

1. the initial point is below the fixed point (by definition)
2. everything above the initial point is accessible
3. all edges in the worklist are in the flow graph
4. all edges in the flow graph are in the worklist

5. each initial value is less or equal to the maximum value of itself and an irrelevant value

Agda

```
maximal-fixed-point : Σ[ c ∈ FixedPoint ] ((y : FixedPoint) → fp c V.⊑ fp y)
maximal-fixed-point = mfp initial initial⊑fp (V.⊐-isWellFounded initial)
  V.⊑-reflexive F (λ e x → x) (λ ℓ ℓ′ x → inj₁ x) (λ ℓ′ → ⊔-on-right-⊑ ⊑-reflexive)
```

The proof for `mfp` is done by case analysis on the work list.

If there is an element on top of the work list we consider if the application of the transfer function yields a bigger value. If it does not, we continue with the rest of the worklist. If the value is bigger, we update the vector:

Agda

```
x' : V.ℂ
x' = x [ ℓ′ ]≔ f ℓ x ⊔ lookup ℓ′ x
```

And consecutively we must show that for `x'` and the new smaller `worklist` the invariants (I , J , K , etc) still hold. To accomplish this, we prove the following lemmas:

Agda

```
-- for each element ℓ, the transfer function for ℓ applied to x is smaller
-- or equal to the transfer applied to the value at x'
fx⊑fx′-pointwise : (ℓ : Label) → f ℓ x ⊑ f ℓ x'
-- The lowest upper bound of the vector x is smaller than
-- the lowest upper bound of x'
⨆fx⊑⨆fx' : (ℓ′′ : Label) → ⨆ (𝕃.map (flip f x) (predecessors F ℓ′′))
                          ⊑ ⨆ (𝕃.map (flip f x') (predecessors F ℓ′′))
-- K : x' is still less than any other fixed point
x'⊑fp : (y : FixedPoint) → x' V.⊑ fp y

-- I : we show that for all edges (ℓ′′ , ℓ′′′) that are not in the worklist:
-- lookup l₄ x' ⊒ ℱ l₃ (lookup l₃ x')
⋄x'⊒fx' : (l₃ l₄ : Label) → (l₃ , l₄) ∈ F → (l₃ , l₄) ∈
 outgoing F ℓ′ 𝕃.++  workList ⊎ (lookup l₄ x' ⊒ ℱ l₃ (lookup l₃ x'))

-- J : we show using monotonicity of the transfer function that
-- the new x' is below the fixed point.
x'⊑⨆f : (ℓ′′ : Label) → lookup ℓ′′ x' ⊑ lookup ℓ′′ initial ⊔
 ⨆ (𝕃.map (flip f x') (predecessors F ℓ′′ ))
```

When the worklist is empty, we know by invariant I that `(ℓ , ℓ′) ∈ F → lookup ℓ′ x ⊒ f ℓ x` and therefore it must be that `∀ ℓ′ → lookup ℓ′ x ⊒ ⨆ { f ℓ x | (ℓ,ℓ′) ∈ F}` . Due to antisymmetry with J we know that the result must be a fixed point and it must be the least one due to K .

4.2 While language

To provide an example analysis in our framework we obviously need some language to perform this analysis on. For this purpose, we will make use of the While language [20]. We create a pure description of it and a description with additional information in the form of labels and unique variables. The language is described in Agda parameterized by label and variable types. The reason being that we want to represent `Var` , the type of variables, as `Fin n` and as `String` . The While language consists of arithmetic expressions, boolean expressions which can be composed of some trivial operations. It

also features simple while loops, conditionals and sequential statements and manipulation of program variables by use of the assignment operation. Since the primary goal is to provide an example for explanatory purposes, only a small subset of expressions is supported. There is no support for functions.

Description of arithmetic and boolean expressions:

Agda

```
module Common where
  data AExpr {a} (Ident : Set a) : Set a where
    var : Ident → AExpr Ident
    lit : ℤ → AExpr  Ident
    _plus_ : AExpr Ident → AExpr Ident → AExpr Ident
    _min_ : AExpr Ident → AExpr Ident → AExpr Ident
    _mul_ : AExpr Ident → AExpr Ident → AExpr Ident

  data BExpr {a} (Ident : Set a) : Set a where
    true : BExpr Ident
    false : BExpr Ident
    not : BExpr Ident → BExpr Ident
    _and_ : BExpr Ident → BExpr Ident → BExpr Ident
    _or_ : BExpr Ident → BExpr Ident → BExpr Ident
    _gt_ : AExpr Ident → AExpr Ident → BExpr Ident
```

Then we can describe the (unlabeled and labeled) statements in the language as :

Agda

```
module Unlabeled where
  AExpr : Set
  AExpr = Common.AExpr String

  BExpr : Set
  BExpr = Common.BExpr String

  data Stmt : Set where
    _:=_ : (v : String) → (e : AExpr) → Stmt
    skip :  Stmt
    _seq_ : (s₁ : Stmt) → (s₂ : Stmt) → Stmt
    if_then_else_ : (c : BExpr) → (t : Stmt)
                  → (f : Stmt) → Stmt
    while_do_ : (c : BExpr) → (b : Stmt) → Stmt

module Labeled where
  data Stmt' {a} {b} (Lab : Set a) (Var : Set b) : Set (a Level.⊔ b) where
    _:=_ : (v : Var) → (e : Common.AExpr Var) → (l : Lab) → Stmt' Lab Var
    skip : (l : Lab) → Stmt' Lab Var
    _seq_ : (s₁ : Stmt' Lab Var) → (s₂ : Stmt' Lab Var) → Stmt' Lab Var
    if_then_else_ : (Common.BExpr Var × Lab)
                  → (t : Stmt' Lab Var) → (f : Stmt' Lab Var) → Stmt' Lab Var
    while_do_ : (Common.BExpr Var × Lab) → (b : Stmt' Lab Var) → Stmt' Lab Var
```

To make things concise, different types of expressions are represented as Blocks following the approach taken by Nielson, Nielson et al [20]. By doing such a transformation we lose some information but make it a bit easier to describe an analysis. For example, a static analysis counting the number of while statements becomes impossible by this approach.

Agda

```
data Block' {a} {b} (Lab : Set a) (Var : Set b) : Set (a Level.⊔ b) where
  skip : (l : Lab) → Block' Lab Var
  _:=_ : (x : Var) → (a : Common.AExpr Var) → (l : Lab) → Block' Lab Var
  bExpr : (c : Common.BExpr Var) → (l : Lab) → Block' Lab Var
```

We fold over the above datatype using an algebra computing different properties and

invariants and transforming the unlabeled program to a labelled one. Finally, a While program can be represented as:

Agda

```
record WhileProgram : Set₁ where
  field
    n : ℕ
    Var* : Bag String
  m : ℕ
  m = length (Util.Bag.toList Var*)
  Lab : Set
  Lab = Fin n
  Var : Set
  Var = Fin m
  AExpr : Set
  AExpr = Common.AExpr Var
  BExpr : Set
  BExpr = Common.BExpr Var
  Stmt : Set
  Stmt = Stmt' Lab Var
  Block : Set
  Block = Block' Lab Var
  field
    blocks : Vec Block n
    labelledProgram : Stmt
```

Various properties are computed for an unlabeled program such as the set of variables, number of variables, number of labels, etc. To comfortably work with the language in the monotone framework we also define some functions:

Agda

```
-- The initial label (entry point) of a statement
init : Stmt → Lab

-- The non empty set of final labels a statement can end
final : Stmt → List⁺ Lab

-- The set of labels for a statement
labels : Stmt → List Lab

-- The control flow graph, represented by a list of label pairs.
flow : Stmt → List (Lab × Lab)

-- Reversed flow
flowᴿ : Stmt → List (Lab × Lab)
```

4.2.1 Live variables

As example, we show how to compute and construct a termination proof of a live variable analysis for the While language.

Live Variable analysis is an analysis that can be used to remove dead code from a program, dead code being code that is known not to influence the outcome of the program. The analysis starts at the end of the program, and then we determine what variables can be live at the exit of each program point starting with, depending on the application, the empty set.

We can determine some variable is alive if it used in some expression. Therefore, at each program point we check what variables are used and mark them live. Furthermore, if a variable x is defined, we remove it from the live variables as each use of x refers to this assignment, assuming we do not make use of some earlier defined x to define the new x.

Because at each program point we remove some variables and add some variable, this type of analysis can be regarded as a kill-gen analysis.

Agda

```
-- fv is a function that returns all free variables for some expression
fv : (BExpr | AExpr) → 𝒫 Var*

kill : Block → 𝒫 Var*
kill (skip l) = ⊥
kill ((x := a) l) = [ x ]
kill (bExpr c l) = ⊥

gen : Block → 𝒫 Var*
gen (skip l) = ⊥
gen ((x := a) l) = fv a
gen (bExpr c l) = fv c
```

The transfer function can then be defined, for each label assuming Block is the block of the label, as:

Agda

```
transfer-function : Block → 𝒫 Var* → 𝒫 Var*
transfer-function b x = (x - (kill b)) ∪ gen b
```

Using these building blocks, we can form the monotone framework and perform the analysis:

Agda

```
live-variables : Stmt → MonotoneFramework _
live-variables program = record
  { L = 𝒫ᴸ-by-inclusion 3
  ; ℱ = transfer-functions
  ; F = flowᴿ program
  ; E = final program
  ; ι = ⊥
  ; ℱ-isMonotone = _ -- postulate
  }

analysis : ℂ
analysis = mfp-result live-variables
```

Note that live variable analysis is a backward analysis, which we perform by using the reversed flow: `flowᴿ` and by starting from the final labels.

4.2.2 Available expressions

Another classical example is Available expression analysis. This analysis is interesting because it uses forward flow. It is also a kill - gen analysis. It computes at every program point what subexpressions are available.

Agda

```
available-expressions : MonotoneFramework _
available-expressions = record
    { L = 𝒫ᴸ-by-exclusion (length AExpr⋆)
    ; 𝓕 = transfer-functions
    ; F = flow labelledProgram
    ; E = [ init labelledProgram ]
    ; ι = ⊥
    ; 𝓕-isMonotone = transfer-monotone
    }
```

Note that we now make use of the normal flow and start at the initial label of our program.

4.2.3 Constant propagation

Constant propagation is different from the other two analyses as it is not an analysis that can be performed distributively. Furthermore, it makes use of the total function space combinator, as we perform the analysis for each variable i.e. the lattice we work with is: `Fin m → 𝕀⊤⊥ᴸ` .

Agda

```
𝑨CP : AExpr → (Fin m → BoundedSemiLattice.ℂ ℤ⊤⊥ᴸ) →  BoundedSemiLattice.ℂ ℤ⊤⊥ᴸ
𝑨CP (var x) ĝ = ĝ x
𝑨CP (lit n) ĝ = in-ℤ n
𝑨CP (x plus y) ĝ = 𝑨CP x ĝ plusℤ 𝑨CP y ĝ
𝑨CP (x min y) ĝ = 𝑨CP x ĝ minℤ 𝑨CP y ĝ
𝑨CP (x mul y) ĝ = 𝑨CP x ĝ mulℤ 𝑨CP y ĝ
```

Given an expression and an environment, i.e. a mapping from variables to their current values, ACP tries to compute the expression using the environment. Instead of computing the actual computation at runtime, we interpret an abstraction. Once any value in the computation reaches $\top$, the entire expression becomes $\top$.

We can use ACP to form the transfer function. Information is propagated through the control flow graph and upon reaching an assignment we try to compute the value using the information we have and then update this variable in the current environment.

Agda

```
transfer-functions : Lab → ℂ → ℂ
transfer-functions l x = case lookup l blocks of (λ
  { (Labeled.skip l₁) → x
  ; ((x₁ Labeled.:= a) l₁) → λ m' → case m' FinP.≟ x₁ of (λ
        { (yes p) → 𝑨CP a x
        ; (no ¬p) → x m'
        })
  ; (Labeled.bExpr c l₁) → x
  })
```

Which we use to create the monotone framework:

Agda

```
constant-propagation : MonotoneFramework _
constant-propagation = record
  { L = Fin m -[ m , Inverse.id ]→ ℤ⊤⊥ᴸ
  ; 𝓕 = transfer-functions
  ; F = flow labelledProgram
  ; E = Data.List.[ init labelledProgram ]
  ; ι = λ x → top
  ; 𝓕-isMonotone = transfer-monotone
}
```

4.3 Decidability of monotonicity

In the case that $\mathbb{C}$ is listable we can also use brute force to show monotonicity of a function since $\sqsubseteq$ is decidable.

Agda

```
module _ {a} {ℓ} {ℂ : Set a} (_⊑_ : Rel ℂ ℓ) (_⊑?_ : Decidable _⊑_)
         (f : ℂ -> ℂ) (ls : Listable ℂ) where
decidable-monotonicity : Dec (Monotone _⊑_ f)
decidable-monotonicity with
 all? (λ x → all? (λ y → x ⊑? y →-dec f x ⊑? f y) (Listable.all ls)) (Listable.all ls)
decidable-monotonicity | yes p = yes
 (λ {x} {y} q → (lookup (lookup p (Listable.complete ls x)) (Listable.complete ls y)) q)
decidable-monotonicity | no ¬p = no
 (λ ⊑-isMonotone → ¬p (tabulate (λ _ → tabulate (λ _ x⊑y → ⊑-isMonotone x⊑y))))
```

5 Embellished monotone frameworks

More realistic programming languages often support functions or procedures, i.e. functions that modify state. We describe the changes we have to make to a monotone framework to perform static analysis of such languages. Luckily for us, the changes we need to introduce are all related to the language itself or to the representation of the data.

An embellished framework can be turned into a monotone framework which allows us to reuse existing algorithms and their properties.

5.1 While-Fun language

We add a statement that can call a function using parameters to obtain a result which is bound to a variable.

Agda

```
data Stmt : Set where
  call : (name : String) → (arguments : List AExpr) → (result : String) → Stmt
  ...
```

Furthermore, we add a definition for a declaration, as the procedures need to be defined somewhere, and a program definition that contains a list of declarations and a main program as entry point.

Agda

```
data Decl : Set where
  proc_(_,_)_end : (name : String) → (arguments : List String) →
                   (result : String) → (body : Stmt) → Decl
data Program : Set where
  begin_main-is_end : (declarations : List Decl) → (main : Stmt) → Program
```

We call the resulting language While-Fun.

The changes to the language propagate to the blocks, to make as much information available for analysis purposes as possible.

Agda

```
data Block' .. : Set .. where
  call : .. → Block' ..
  return : .. → Block' ..
  entry : .. → Block' ..
  exit : .. → Block' ..
  ...
```

Note that we add four different block types: call and return identify positions at the

calling site, whereas entry and exit identify positions in the function. Multiple calls to a function thus share entry and exit but have different call and return labels. By doing this we follow the same approach as taken by Nielson et al [20]. As for the While language, we define some functions that help us identify the correct flow for analyses.

Agda

```
init⋆ : Program → Lab
final⋆ : Program → List Lab
flow⋆ : Program → List Edge
```

We assume the program to be verified is a valid `While-Fun` program. A program is valid when all functions referred to are defined, the defined functions have unique names and all variables are unique to avoid shadowing. We thus modify our previous definition for `WhileProgram` to only consist of valid programs accompanied by the proofs of validity:

Agda

```
record WhileProgram : Set₁ where
  field
    Fun* : Bag String
  k : ℕ
  k = length (Util.Bag.toList Fun*)
  Fun : Set
  Fun = Fin k
  Decl : Set
  Decl = Decl' Lab Var Fun
  Program : Set
  Program = Program' Lab Var Fun
  field
    functions : Vec Decl k
  ...
```

5.2 Embellished flow

When considering information flow, information before a function call should propagate to after the function call. To accomplish this, we require the transfer function at the return label to have two arguments. The first one is the information before the

program call, and the second is information from inside the program call. This way, the combination of information can be made analysis specific. In order to achieve such behaviour we model the type of the transfer function as dependent on the type of block.

To lift an analysis to an embellished framework we do need to know what program points are return or call labels. Note that for reversed flow these two interchange.

We let Agda know this by a function:

Agda

```
data EmbellishedBlock (n : ℕ) : Set where
  other call : EmbellishedBlock n
  return : (c : Fin n) → EmbellishedBlock n

labelType : Label → EmbellishedBlock n
```

Now, we can make our transfer function binary at return labels by

Agda

```
data Arity : Set where
  unary : Arity
  binary : Arity

arityToType : ∀{a} → Arity → Set a → Set a
arityToType unary 𝕃 = 𝕃 → 𝕃
arityToType binary 𝕃 = 𝕃 → 𝕃 → 𝕃

arity : ∀{n} → (Fin n → EmbellishedBlock n) → Fin n → Arity
arity f x with f x
arity f x | return c = binary
arity f x | _ = unary

ℱ : (ℓ : Label) -> arityToType (arity labelType ℓ) (BoundedSemiLattice.𝕃 L)
```

Furthermore, the monotonicity constraint is dependent on the arity as well, for binary transfer functions the constraint becomes:

Agda

```
BiMonotone = ∀{x y z w} -> x ⊑ y → z ⊑ w -> f x z ⊑ f y w
```

Note that we can also solve the monotonicity problem by changing the domain of return labels, thus making the domain of the transfer function dependent on label type instead of arity resulting in the use of heterogeneous vectors. For return labels we would then transform the domain to `L ×ᴸ L`, allowing us to use the regular definition of monotonicity albeit on different domains.

5.3 Context

Because we abstract the programs execution paths to a flow graph, we lose some information. More precisely we lose the coupling between (call, entry) and (exit, return). This means that in our analysis it is possible for information to flow from one function call into the return site of some other function call. To prevent such poisoning we only consider *valid* paths as defined by Nielson, Nielson and Hankin [20]. We do this by adding context to our lattice in the form of a call string. A call string is a finite list of call labels of functions which we assume are on top of the call stack at some program point. Since we cannot use a call string that is infinitely long, we only consider call strings of length less than some number `k`. For `k ≡ 2`, the call string [1,3] represents information for some program point with a call stack of [1,3,2,4] but also for [1,3,3] i.e. the information is aggregated. We represent call strings by a bounded list. Adding items to a maximal bounded list, i.e. a list of length `k`, will cause a shift such that the last item on the list will be dropped.

Therefore, we represent context as:

Agda

```
Δ : Set
Δ = BoundedList (Fin n) k
```

Given an analysis for a regular monotone framework `(L, 𝓕, F, E, ι)` and some additional information (for the newly added language statements), we can lift the analysis to an embellished one by considering context and using the total function space combinator. Note that we have to show finiteness of a bounded list.

Agda

```
L̂ : BoundedSemiLattice a
L̂ = Δ -[ .. ]→ L
```

To get the new extremal values for this lattice we start by using ι for the empty bounded list, representing the initial empty call stack, and $\bot$ for all other lists.

Agda

```
î : ℂ
î (zero , nil) = ι
î (suc n₁ , cons' x xs x₁) = L.⊥
```

We can then transform the possible binary transfer function $\mathcal{F}$ to the always unary function $\mathcal{F}$:

Agda

```
ℱ : Fin n → ℂ → ℂ
ℱ ℓ l̂ δ′ with labelType ℓ | ℱ ℓ
-- we just propagate information
ℱ ℓ l̂ δ′ | other | f = f (l̂ δ′)
-- ..
ℱ ℓ l̂ δ′ | call | f =
 f (L.⨆ (𝕃.map (λ δ → if ⌊ cons ℓ δ ≟⟨ _≟_ ⟩ δ′ ⌋ then l̂ δ else L.⊥ ) allCallStrings≤k ))
-- here at program point ℓ for call string δ′, we use the transfer
-- function with information for δ′, but also all callstrings that
-- have ℓc on top of the call stack, as they may return to ℓ.
ℱ ℓ l̂ δ′ | return ℓc | f = f (l̂ δ′) (l̂ (cons ℓc δ′))
```

The proof of $\mathcal{F}$ being monotone can be found in the source code [24]. Finally, we can use this information to create a new monotone framework:

Agda

```
asMonotoneFramework : MonotoneFramework a
asMonotoneFramework = record
  { n = n
  ; L = L̂
  ; 𝓕 = 𝓕̂
  ; F = F
  ; E = E
  ; ι = ι̂
  ; 𝓕-isMonotone = 𝓕̂-isMonotone
  }
```

By constructing an analysis using this embellishment, a user only has to create the simple lattice `L` and show monotonicity for the simpler transfer function $\mathcal{F}$.

5.4 Constant propagation example

Constant propagation can now also be described for an embellished framework. We take the easy route by assuming/proving all variables in the program are unique, thus there can be no shadowing. Then constant propagation is a lot simpler. Nevertheless, it gives an example of how the framework can be used.

We create a function that shows what labels are call and return blocks:

Agda

```
embellishedType : Fin n → EmbellishedBlock n
embellishedType l with lookup l blocks
embellishedType l | call c name r a r₁ = call
embellishedType l | return c name r a r₁ = return c
embellishedType l | _ = other
```

We can use this function in the type of the tranfer function, to allow tranfer functions corresponding to return blocks to require two arguments: `beforeCall` and `afterCall`. The function `afterCall` contains information, or the mapping it represents is updated by information, that passed through the referenced function call including the function's return value. If we want to know the value for a constant `v` such that it is

the return value of a function, we need the information that has passed through the function, in which the variable is assigned. We consider the function to be pure and have no effect on any other variable, so then we can use the information from before the call.

Agda

```
transfer-emb : (ℓ : Fin n) → arityToType (arity embellishedType ℓ) ℂ
transfer-emb ℓ with lookup ℓ blocks
transfer-emb ℓ | call c name r a r₁ = id
transfer-emb ℓ | return  cℓ name rℓ args retvar =
λ beforeCall afterCall v →
 (case v FinP.≟ retvar of
   (λ{ (yes p) → afterCall
  ; (no ¬p) → beforeCall})
 ) v
transfer-emb ℓ | entry name arguments result ln body lx = id
transfer-emb ℓ | exit name arguments result ln body lx = id
transfer-emb ℓ | _ = transfer-function ℓ -- non embellished version
```

We then form an EmbellishedMonotoneFramework:

Agda

```
constant-propagation-embellished : EmbellishedMonotoneFramework _
constant-propagation-embellished = record
  { n = n
  ; L = L
  ; k = 2
  ; labelType = embellishedType
  ; 𝓕 = transfer-emb
  ; F = flow⋆ labelledProgram
  ; E = Data.List.[ init⋆ labelledProgram ]
  ; ι = λ x → top
  ; 𝓕-isMonotone = postulate
  }
```

6 Extended monotone frameworks

We have shown strong guarantees on the output of the MFP algorithm on interprocedural languages. However, for dynamicly typed languages as Python, PHP etc. a lot of information about values is propagated during runtime and thus, is inaccessible during static analysis defined using monotone frameworks. In such languages, the control flow of a program can depend on values and is not statically available. Fritz et al published a variant on the MFP algorithm, the extended algorithm, such that the control flow graph is dependent on the property space but claims it still computes a least fixed point according to some definition. He then uses the algorithm for a type inference analysis for Python. The algorithm is then used by Van der Hoek for an object-sensitive type analysis for PHP [23].

The extended algorithm is described below as described by Van der Hoek (for details on used functions, see [23, p. 19]):

Pseudocode

```
step 1: initialisation
IF := ∅
W := nil
for ℓ ∈ E do
  A[ℓ, Λ] := ι
  for ((ℓ, δ), (ℓ′, δ′)) ∈ next ℓ Λ ∅
    W := cons ((ℓ , δ) , (ℓ′ , δ′)) W

step 2: iteration
while W ≠ nil do
 ((ℓ, δ), (ℓ′, δ′)) ← head W
  W := tail W
  if (ℓ, δ) ∈ returnPoint(IF) then
    ℓc := IF ℓr
    effect := f ℓc ℓr (A[ℓc, δ], A[ℓ, δ])
  else
    effect := f ℓ δ (A[ℓ, δ])
  if effect ⊄ A[ℓ′, δ′] then
    A[ℓ′, δ′] := A[ℓ′, δ′] ⊔ effect
    IF := φ ℓ′ δ′ A[ℓ′, δ′] ∪ IF
    for ((ℓ′, δ′), (ℓ′′, δ′′)) ∈ next ℓ′ δ′ IF do
      W := cons ((ℓ′, δ′), (ℓ′′, δ′′)) W

step 3: presentation
for all ℓ , δ do
MFP○(ℓ, δ) := A[ℓ, δ]
if (ℓ, δ) ∈ returnPoints IF then
  ℓc := IF ℓr
  MFP●(ℓ, δ) := f ℓc ℓr (A[ℓc, δ], A[ℓ, δ])
else
  MFP●(ℓ, δ) := f ℓ δ (A[ℓ, δ])
```

Van der Hoek models a stack and a heap and then uses the extended algorithm to form an Object Sensitive Type analysis for PHP. In an attempt to add support for analyses such as Fritz's or Van der Hoek's we start out by adapting the MFP algorithm to allow extension of the control flow graph during execution of the algorithm.

The extended algorithm is explicitly written for embellished frameworks and thus keeps track of information regarding interprocedural flow. However, the analysis being interprocedural seems to be irrelevant for the extension of flow. Therefore, a non embellished variant of the extended algorithm is considered first. Note that by changing the algorithm this way, we can still use it later on for embellished frameworks. As embellished frameworks are instances of regular monotone frameworks, we hope to show identical results for embellished-extended and extended frameworks. The final goal is to show that any embellished or monotone framework can be turned into an extended one.

Instead of representing the Control Flow Graph as a list of edges we now represent it as a set of edges, because then we can use the powerset by inclusion lattice. Therefore the flow graph is an element of the lattice `F^L = 𝒫^L-by-inclusion (n * n)` .

Since we factor out the interprocedural part, we only have to consider the `next` function and it turns out that we also require this function to be monotone.

These changes result in a new definition of an extended framework:

Agda

```
record ExtendedFramework a : Set (Level.suc a) where
 field
  n : ℕ
  L : BoundedSemiLattice a
 open BoundedSemiLattice L
 Label : Set
 Label = Fin n
 CFG : Set
 CFG = Subset (n * n)
 Fᴸ : BoundedSemiLattice _
 Fᴸ = 𝒫ᴸ-by-inclusion (n * n)
 field
  ℱ : Label -> 𝕃 -> 𝕃  -- Set of transfer functions indexed by label
  -- function that given information provides new edges for the control flow graph.
  next : Label → 𝕃 → CFG
  E : List Label   -- Extremal labels
  ι : 𝕃            -- Extremal value
  ℱ-isMonotone : (ℓ : Fin n) → Monotone _⊑_ (ℱ ℓ)
  next-isMonotone :
   (ℓ : Fin n) → Monotone₂ _⊑_ (BoundedSemiLattice._⊑_ Fᴸ) (next ℓ)
```

The initial control flow graph (`initial-F`) is now formed by the union of pointwise application of the next function on initial values. Note that this is different from Van der Hoek's version, as he always starts from `next ℓ ∅` . Any edges in `next ℓ ∅` are still present in `next ℓ ι` , because of monotonicity.

Agda

```
initial-F : CFG
initial-F =
 F.⨆ (Data.List.map (λ ℓ → next ℓ (lookup ℓ initial)) (Data.Vec.toList (allFin n)))
```

We then update the MFP algorithm to include flow extension:

Agda

```
module Standard where
mfp-extended : (x : V.ℂ) → (workList : List (Label × Label))
 → (F : CFG)  → Σ[ F̂ ∈ CFG ] V.ℂ
mfp-extended x [] F = F , x
mfp-extended x ((ℓ , ℓ′) ∷ workList) F = case-nonemptyworklist (f ℓ x ⊑? lookup ℓ′ x)
 where
  v = f ℓ x ⊔ lookup ℓ′ x
  F′ = F F.⊔ next ℓ′ v
  F′⊒F : F′ F.⊒ F
  F′⊒F = F.⊔-on-right-⊑ F.⊑-reflexive
  x′ = x [ ℓ′ ]≔ v
  case-nonemptyworklist : Dec (f ℓ x ⊑ lookup ℓ′ x) → Σ[ F̂ ∈ CFG ] V.ℂ
  case-nonemptyworklist (yes p) = mfp-extended x workList F
  case-nonemptyworklist (no ¬p) =
   mfp-extended x′
    (set-to-list (F′ Util.Subset.- F) 𝕃.++ outgoing (set-to-list F) ℓ′ 𝕃.++ workList)
    F′

mfp : Σ[ F̂ ∈ CFG ] V.ℂ
mfp = mfp-extended initial (set-to-list initial-F) initial-F
```

The first change to note is that we now use a local function `case-nonemptyworklist`. The function has nothing to do with the algorithm itself, but allows us to reuse the definition of `v`, `F′` and several other properties in the two possible branches of `case-nonemptyworklist`. In the recursive call when $f\ \ell\ x \not\sqsubseteq \mathtt{lookup}\ \ell'\ x$, all edges newly added by the next function are placed at the front of the worklist. The case that next `(ℓ , ℓ′)` adds an entirely different edge `(ℓ′′ , ℓ′′′)` is included. Furthermore, all outgoing edges from ℓ' that were already in `F` are added to the worklist as they might require an update identical to the MFP algorithm.

To let Agda know the algorithm terminates, we are now also required to show that our increasing argument `F′` will reach a point such that it cannot increase anymore. Since `F′` is a member of `Fᴸ`, we can use the `⊐-isWellFounded` property of `Fᴸ` to show termination.

Agda

```
module WithTermination where
mfp-extended :
-- for all vectors x
   (x : V.ℂ)
-- of which all greater values are accessible
 → Acc V._⊐_ x
-- and for all work lists
 → (workList : List (Label × Label))
-- and for all control flow graphs
 → (F : CFG)
-- of which all greater values are accessible
 → Acc F._⊐_ F
-- there exists a control flow graph F̂ such that we obtain an x.
 → Σ[ F̂ ∈ CFG ] V.ℂ
```

Since the fixed point is now relative to a Control Flow Graph, the definition needs to be altered.

Agda

```
IsFixedPoint : CFG → V.ℂ → Set a
IsFixedPoint F x =
 ((ℓ′ : Label) → (lookup ℓ′ initial ⊔
   ⨆ (𝕃.map (flip f x) (predecessors (set-to-list F) ℓ′)) ≡ lookup ℓ′ x ))
 × F ≡ initial-F F.⊔
       F.⨆ (𝕃.map (λ ℓ → next ℓ (lookup ℓ x)) (Data.Vec.toList (allFin n)))
```

We reach a fixed point whenever additional application of the transfer function of predecessors does not result in new information and such that additional application of the next function for any label using the current values for `x` does not contribute to the control flow graph.

Supplementary to the MFP proof we have to prove the following inductively, to obtain a least fixed point:

1. Given `F̂`, `x` is less or equal to any other fixed point for `F̂`
2. `F` stays above or equal to `initial-F` We start out using `initial-F`, so the

base case trivially holds. The inductive case is valid because of transitivity.

3. `F` contains all information supplied by the next function for each label F starts out using some initial flow which we obtain from the next function for initial values, whenever we have to update information and the information for some label `ℓ′` has changed, we update F to include this information.

4. `F` stays below our definition of the fixed point. The initial point is below the fixed point, as it includes the initial point.

   Since `F′ ≡ F F.⊔ next ℓ′ (f ℓ x ⊔ lookup ℓ′ x)` , we have to show

   ```
   lemma-A : F ⊑ initial-F F.⊔
   F.⨆ (𝕃.map (λ ℓ′′ → next ℓ′′ (lookup ℓ′′ x′)) (toList (allFin n)))
   ```

   and

   ```
   lemma-B : next ℓ′ (f ℓ x ⊔ lookup ℓ′ x)
   ⊑
   initial-F F.⊔ F.⨆
   (𝕃.map (λ ℓ′′ → next ℓ′′ (lookup ℓ′′ x′)) (toList (allFin n)))
   ```

   `lemma-A` can be proven by the induction hypothesis for `F` and using monotonicity on `next` and `lookup` to show that

   ```
   F.⨆ (𝕃.map (λ ℓ → next ℓ (lookup ℓ x)) (toList (allFin n)))
   ⊑ F.⨆ (𝕃.map (λ ℓ → next ℓ (lookup ℓ x′)) (toList (allFin n)))
   ```

   `lemma-B` is proven by showing that `ℓ′ ∈ allFin n` and thus `next ℓ′ (f ℓ x ⊔ lookup ℓ′ x) ∈ ⨆ ..`

The above information is used in the following theorem:

Agda

```
mfp-extended :
-- for all vectors x
   (x : V.ℂ)
-- that are above or equal to the initial value
 → initial V.⊑ x
-- and of which all greater values are accessible
 → Acc V._⊐_ x
-- and for all work lists
 → (workList : List (Label × Label))
-- and for all control flow graphs
 → (F : CFG)
-- that are above or equal to the initial control flow graph
 → initial-F F.⊑ F
-- and of which all greater values are accessible
 → Acc F._⊐_ F
-- such that x is less or equal to any fixedpoint y for any F̂
-- that is greater or equal to F.
 → ((F̂ : CFG) → F̂ F.⊒ F → (y : FixedPoint F̂) → x V.⊑ fp y)
-- and such that anything that is in the worklist originated from the flow graph
 → ((ℓ ℓ′ : Label) → (ℓ , ℓ′) listϵ workList → (ℓ , ℓ′) setϵ F)
-- and such that everything not in the worklist is above its predecessors
 → ((ℓ ℓ′ : Label) → (ℓ , ℓ′) setϵ F → ¬ (ℓ , ℓ′) listϵ workList →
   lookup ℓ′ x ⊒ 𝓕 ℓ (lookup ℓ x))
-- and such that F contains all new control flow added by the next function for
-- the current value.
 → ((ℓ : Label) → F F.⊒ next ℓ (lookup ℓ x))
-- and such that for all ℓ′ the value at ℓ′ in x is less or equal to the maximal
-- value of the transfer function applied over all predecessors of ℓ′ using
-- the current flow and the initial value at ℓ′.
 → ((ℓ′ : Label) → lookup ℓ′ x ⊑ lookup ℓ′ initial ⊔
   ⨆ (𝕃.map (flip 𝑓 x) (predecessors (set-to-list F) ℓ′)))
-- and such that the control flow graph remains true to the control flow graph
-- defined by the least upper bound of the next function on all labels.
 → F F.⊑ initial-F F.⊔
   F.⨆ (𝕃.map (λ ℓ′′ → next ℓ′′ (lookup ℓ′′ x)) (Data.Vec.toList (allFin n)))
-- there exists a control flow graph F̂ such that x is a fixed point,
-- and all other fixed points under F̂ are bigger or equal to x.
 → Σ[ F̂ ∈ CFG ] Σ[ x ∈ FixedPoint F̂ ] ((y : FixedPoint F̂) → fp x V.⊑ fp y)
```

Note that, we rely on a bijection: `(Fin m × Fin n) ↔ Fin (m * n)` which we postulated. Thus, without the explicit proof, we lose computability.

By removing the flow from an Embellished Framework, we can make two functions that convert the Embellished Framework to either a Monotone Framework when the full Control Flow Graph is given, or an Extended Framework when the next function and proof of monotonicity is supplied:

Agda

```
record EmbellishedMonotoneFramework a : Set (Level.suc a) where
 ..
 asExtendedFramework : (next : Label → ℂ → Subset (n * n)) → ((ℓ : Fin n)
                     → Monotone₂ _⊑_ (𝒫._⊑_) (next ℓ)) → ExtendedFramework a
 asExtendedFramework next next-mono =
  record
  { n = n
  ; L = L̂
  ; 𝓕 = 𝓕̂
  ; next = next
  ; E = E
  ; ι = ι̂
  ; 𝓕-isMonotone = 𝓕̂isMonotone
  ; next-isMonotone = next-mono
  }
```

7 Related work

David Darais and David Van Horn have worked on a similar project in which they provide a framework that allows derivation of correct abstract interpreters by construction. Their framework is monadically composed of constructive galois connections [6,7]. Denis Firsov et al have worked on a Listable type, that easily allows you to construct data from finite sets; perhaps it is worthwhile to extend this work to incorporate complete lattices by also deriving some order [9,10]. J. Knoop et al worked on machine checkable abstract interpretation based interprocedural data flow analysis in the theorem prover

Athena. Their methodology facilitates construction of correct analyses as they separate generic proofs from analysis specific proofs which is similar to what we attempt to do using the lattice combinators [16]. David Cachera et al provide a similar framework for Coq with a constraint based analysis for OCaml. They argue that these static analysis frameworks reduce the gap between analysis proven on paper using abstract interpretation versus the analysis computed on a machine. While proving properties of these analysis for toy languages might be trivial, doing the same for real-life languages might require a lot more engineering [4].

Kahl and Al-hassy formalized order theoretic concepts in their work Relational Algebraic Theories in Agda (RATH). Their work includes definitions for complete lower semilattices and Galois connections. They also provide dualization techniques but report that 'even with 52GB of heap (on a machine with 64GB of RAM), the current development version of Agda still runs out of heap space after a few days when checking this.' [13].

The CompCert project aims to investigate possibilities of compiler verification. The project resulted in the Compcert C compiler, for which about 90% of the algorithms are proved correct. Several compiler optimisations are applied to the C compiler which are based on several static analyses. The Compcert C compiler uses Kildall's algorithm to apply constant propagation, dead code elimination and common subexpression elimination for which the static analyses and algorithms are verified using Coq [17].

8 Further research and extensions

The parallel fixed point algorithm using the Total Function Space combinator and Tarski's least fixed point theorem seems definitionally very elegant (opposed to computational). One can wonder whether such elegant structures exist for the other worklist algorithms since a lot of steps in the proof of obtaining a fixed point are shared between them. Perhaps, it can be worthwhile to extend the lattice product combinator to a dependent product combinator parameterized by some constraints. Perhaps such a combinator can be used to show preservation of a fixed point. Given a fixed point for some data structure, can we make use of the combinator to find a new fixed point for a sub do-

main using more specialized constraints. It might be that smart data structures can be chosen using the normal fixed point theorem such that more complicated algorithms, for determining the least fixed point, become obsolete while maintaining equivalent output, performance and complexity.

As lattices can be represented using Hasse diagrams, the proof of some graph object being a lattice can also be automatically determined.

Another extension to monotone framework is the ability to deal with object sensitive analysis. In which a call on some receiver object is identified by putting the receivers objects allocation sites in the context [19]. Since no algorithmic, in respect to finding the least fixed point, changes are required to support such an analysis, the framework can be used to apply object sensitive analysis for some target object oriented language such as Van der Hoek's analysis [11, 23].

When considering the use case of an dependently typed verified compiler, dependently typed attribute grammars can play an important role in the specification of the analysis. For example, when assigning labels to program blocks we need to know in advance the amount of program blocks (`n`) the program has as we need to construct a member of `Fin n` . This also holds for assigning variables where we need uniqueness constraints and for functions calls where each referenced functions needs to be defined. To more easily propagate information, or proofs, to the right places, the use of an attribute grammar as described by Middelkoop et al could help a lot [18]. Additionally, one could look into smart updating of an analysis where an update to the target language has effects on the least fixed point without recomputing the whole analysis [3].

9 Conclusion

The Monotone Dataflow Analysis Framework initially presented by Kam and Ullman, Embellished Frameworks and part of Extended Frameworks are formalized in Agda. Proofs of termination and proofs of obtaining a least fixed point are provided for several algorithms. The resultings theorem can be seen as a specialized instance of Tarski's fixed point theorem. Using the formalisation in Agda, one can obtain proofs for the ascending

chain condition and other properties by composition of bounded semi lattices. We have supplied several instances including the total function space, n-ary products and the powerset. As the Agda code can be compiled, some groundwork is laid down possibly contributing to verified compilers and automated verification of software in general.

Acknowledgements

First, I would like to thank dr. J. Hage and dr. W.S. Swierstra for their immense patience, dedication and support to put up with my flaws. Additionally, I am in debt for their guidance and knowledge they provided throughout my study at Utrecht University and during my thesis and so I wish them `T : Luck .`

Second, I would like to thank my parents and family for their unconditional support, encouragement and superb caretaking. Without them this would not have been possible.

Thank you

References

[1] Nils Anders Danielsson, Ulf Norell, et al. Agda Standard Library. `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Libraries.StandardLibrary`, 2016. [Online].

[2] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda – a functional language with dependent types.

[3] J Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars*. PhD thesis, Utrecht University, 2015.

[4] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Applied semantics: Selected topics extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56 – 78, 2005.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[6] David Darais and David Van Horn. Constructive galois connections.

[7] David Darais and David Van Horn. Mechanically verified calculational abstract interpretation. *arXiv preprint arXiv:1507.03559*, 2015.

[8] Enrico Eugenio and Agostino Cortesi. *WiFi-Related Energy Consumption Analysis of Mobile Devices in a Walkable Area by Abstract Interpretation*, pages 27–39. Springer International Publishing, Cham, 2017.

[9] Denis Firsov and Tarmo Uustalu. Dependently typed programming with finite sets. 2014.

[10] Denis Firsov, Tarmo Uustalu, and Niccolo Veltri. Variations on noetherianness. 2014.

[11] Levin Fritz and Jurriaan Hage. Cost versus precision for approximate typing for python. In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2017, pages 89–98, New York, NY, USA, 2017. ACM.

[12] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, et al. Analyzing program termination and complexity automatically with aprove. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

[13] Wolfram Kahl. Relation-algebraic theories in agda. Technical report, Department of Computing and Software, McMaster University, 2017.

[14] John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[15] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[16] J. Knoop, G.C. Necula, W. Zimmermann, Alexandru Sălcianu, and Konstantine Arkoudas. Proceedings of the fourth international workshop on compiler optimization meets compiler verification (cocv 2005) machine-checkable correctness proofs for intra-procedural dataflow analyses. *Electronic Notes in Theoretical Computer Science*, 141(2):53 – 68, 2005.

[17] Xavier Leroy et al. The compcert verified compiler. *Development available at http://compcert. inria. fr*, 2009, 2004.

[18] Arie Middelkoop, Atze Dijkstra, and S Doaitse Swierstra. Dependently typed attribute grammars. In *Symposium on Implementation and Application of Functional Languages*, pages 105–120. Springer, 2010.

[19] Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.

[20] Flemming Nielson, Hanne R Nielson, and Chris Hankin. Principles of program analysis, 2015.

[21] Ulf Norell. Towards a practical programming language based on dependent type theory, 2007.

[22] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[23] Henk Erik Van der Hoek and Jurriaan Hage. Object-sensitive type analysis of php. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, pages 9–20. ACM, 2015.

[24] J.J. van Wijk. Monotone frameworks in Agda. `https://github.com/jornvanwijk/monotoneframeworks-agda`, 2017. [Online].

[25] Valentin Wüstholz, Oswaldo Olivo, Marijn JH Heule, and Isil Dillig. Static detection of dos vulnerabilities in programs that use regular expressions.

Appendix A: Postulates

Agda

```
postulate
-- transfer function of chaotic and parallel evaluation being monotone
 transfer-chaotic-isMonotone :
  Monotone (BoundedSemiLattice._⊑_ (Vecᴸ m L)) transfer-chaotic
 transfer-parallel-isMonotone :
  Monotone (BoundedSemiLattice._⊑_ (Vecᴸ m L)) transfer-parallel
-- proof that the transfer functions of live variable analysis
--in our example are monotone
 transfer-functions-monotone :
 (l : Lab) → Monotone _⊑_ (transfer-functions l)
-- function extensional equality
 Extensionality :
 {A : Set a} {B : A → Set b} {f g : (x : A) → B x} → (∀ x → f x ≡ g x) → f ≡ g
 -- bijection used for extended frameworks
 ℱ×ℱ↔ℱ : (Fin n × Fin n) ↔ Fin (n * n)
 -- call strings are finite
 isBijectiveToFin : ∀{a k} → {A : Set a} →
  Σ[ n ∈ ℕ ] A ↔ Fin n → Σ[ r ∈ ℕ ] BoundedList A k ↔ Fin r
```